Pyomo Documentation

Release 6.10.1

Pyomo Development Team

Jun 04, 2026

CONTENTS

1 Getting Started **2**
- 1.1 Installation . . . 2
  - 1.1.1 Using CONDA . . . 2
  - 1.1.2 Using PIP . . . 2
  - 1.1.3 Optional Dependencies . . . 2
  - 1.1.4 Installation with Cython . . . 3
- 1.2 Using Solvers with Pyomo . . . 3
- 1.3 Pyomo Overview . . . 4
  - 1.3.1 Mathematical Modeling . . . 4
  - 1.3.2 Overview of Modeling Components and Processes . . . 6
  - 1.3.3 Abstract Versus Concrete Models . . . 6
  - 1.3.4 Simple Models . . . 7

2 How-To Guides **15**
- 2.1 Interrogating Models . . . 15
  - 2.1.1 Accessing Variable Values . . . 15
  - 2.1.2 Accessing Parameter Values . . . 17
  - 2.1.3 Accessing Duals . . . 17
  - 2.1.4 Accessing Slacks . . . 19
- 2.2 Manipulating Pyomo Models . . . 19
  - 2.2.1 Repeated Solves . . . 19
  - 2.2.2 Changing the Model or Data and Re-solving . . . 23
  - 2.2.3 Fixing Variables and Re-solving . . . 24
  - 2.2.4 Extending the Objective Function . . . 26
  - 2.2.5 Activating and Deactivating Objectives . . . 27
  - 2.2.6 Activating and Deactivating Constraints . . . 27
- 2.3 Solver Recipes . . . 27
  - 2.3.1 Accessing Solver Status . . . 27
  - 2.3.2 Display of Solver Output . . . 28
  - 2.3.3 Sending Options to the Solver . . . 28
  - 2.3.4 Specifying the Path to a Solver . . . 29
  - 2.3.5 Warm Starts . . . 29
  - 2.3.6 Solving Multiple Instances in Parallel . . . 29
  - 2.3.7 Changing the temporary directory . . . 30
- 2.4 Working with Abstract Models . . . 30
  - 2.4.1 Instantiating Models . . . 30
  - 2.4.2 Managing Data in AbstractModels . . . 32
  - 2.4.3 The `pyomo` Command . . . 64
  - 2.4.4 `BuildAction` and `BuildCheck` . . . 65
- 2.5 Debugging Models . . . 69

2.6 Contributing to Pyomo 69
2.6.1 Contribution Requirements 70
2.6.2 Review Process 72
2.6.3 Where to put contributed code 73
2.6.4 Submitting a new Contributed Package 74
2.6.5 Working on Forks and Branches 76

3 Explanations 79
3.1 Pyomo Philosophy 79
3.1.1 Abstract Models 79
3.1.2 Pyomo Component Design 80
3.1.3 Pyomo Expressions 80
3.1.4 Model Transformations 103
3.2 Modeling in Pyomo 103
3.2.1 Math Programming 103
3.2.2 Dynamic Optimization with pyomo.DAE 133
3.2.3 Generalized Disjunctive Programming 150
3.2.4 MPEC 163
3.2.5 Pyomo Network 163
3.2.6 Units Handling in Pyomo 172
3.3 Solvers 173
3.3.1 Persistent Solvers 173
3.3.2 GDPopt logic-based solver 177
3.3.3 PyROS Solver 182
3.3.4 MindtPy Solver 216
3.3.5 MC++ Interface 225
3.3.6 Multistart Solver 226
3.3.7 Trust Region Framework Method Solver 227
3.3.8 PyNumero 232
3.3.9 z3 SMT Sat Solver Interface 243
3.4 Analysis in Pyomo 243
3.4.1 Generating Alternative (Near-)Optimal Solutions 243
3.4.2 Community Detection for Pyomo models 251
3.4.3 Pyomo.DoE 262
3.4.4 Infeasibility Diagnostics 276
3.4.5 Incidence Analysis 278
3.4.6 MPC 297
3.4.7 Parameter Estimation 309
3.4.8 Sensitivity Toolbox 343
3.4.9 NLP Initialization 346
3.5 Modeling Utilities 349
3.5.1 “Flattening” a Pyomo model 349
3.5.2 Latex Printing 352
3.5.3 Nonlinear Preprocessing Transformations 354
3.5.4 Model Scaling Transformation 361
3.5.5 aslfunctions 362
3.6 Developer Utilities 372
3.6.1 The Pyomo Configuration System 372
3.6.2 Deprecation and Removal of Functionality 380
3.7 Experimental features 381
3.7.1 The Kernel Library 381
3.7.2 Future Solver Interface Changes 387

4 Reference Guides 398

4.1 Topical Reference . . . 398
4.1.1 AML Library Reference . . . 398
4.1.2 Expression Reference . . . 398
4.1.3 Solver Interfaces . . . 400
4.1.4 Model Data Management . . . 403
4.1.5 APPSI . . . 405
4.1.6 The Kernel Library API Reference . . . 409
4.2 Development Principles . . . 413
4.2.1 Backwards Compatibility . . . 414
4.2.2 Dependency Management . . . 415
4.2.3 Miscellaneous Conventions . . . 416
4.3 Accessing preview features . . . 418
4.3.1 Preview capabilities through `pyomo.__future__` . . . 418
4.4 Common Warnings/Errors . . . 418
4.4.1 Warnings . . . 418
4.4.2 Errors . . . 420
4.5 Related Packages . . . 420
4.5.1 Modeling Extensions . . . 420
4.5.2 Solvers and Solution Strategies . . . 421
4.5.3 Domain-Specific Applications . . . 421
4.6 Publications . . . 421
4.7 Bibliography . . . 421

5 Pyomo Resources **422**

6 Contributing to Pyomo **423**

7 Related Packages **424**

8 Citing Pyomo **425**

Bibliography **426**

About Pyomo

Pyomo is a Python-based open-source software package that supports a diverse set of optimization capabilities for formulating, solving, and analyzing optimization models.

A core capability of Pyomo is modeling structured optimization applications. Pyomo can be used to define general symbolic problems, create specific problem instances, and solve these instances using commercial and open-source solvers.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

Pyomo currently supports the following versions of Python:

- CPython: 3.10, 3.11, 3.12, 3.13, 3.14
- PyPy: 3

At the time of the first Pyomo release after the end-of-life of a minor Python version, Pyomo will remove testing for that Python version.

1.1.1 Using CONDA

We recommend installation with `conda`, which is included with the Anaconda distribution of Python. You can install Pyomo in your system Python installation by executing the following in a shell:

```
conda install -c conda-forge pyomo
```

Optimization solvers are not installed with Pyomo, but some open source optimization solvers can be installed with `conda` as well:

```
conda install -c conda-forge ipopt glpk
```

1.1.2 Using PIP

The standard utility for installing Python packages is `pip`. You can install Pyomo in your system Python installation by executing the following in a shell:

```
pip install pyomo
```

1.1.3 Optional Dependencies

Extensions to Pyomo, and many of the contributions in `pyomo.contrib`, often depend on additional third-party Python packages including but not limited to: matplotlib, networkx, numpy, openpyxl, pandas, pint, scipy, sympy, and xlrd.

A full list of optional dependencies can be found in Pyomo's `setup.py`. They can be displayed by running:

```
# Legacy format
python setup.py dependencies --extra optional
# Newer format - prints as a JSON
python -m pip install --dry-run --report - '.[optional]'
```

Pyomo extensions that require any of these packages will generate an error message for missing dependencies upon use.

When using *pip*, all optional dependencies can be installed at once using the following command:

```
pip install 'pyomo[optional]'
```

When using *conda*, many of the optional dependencies are included with the standard Anaconda installation.

You can check which Python packages you have installed using the command `conda list` or `pip list`. Additional Python packages may be installed as needed.

1.1.4 Installation with Cython

Users can opt to install Pyomo with cython initialized.

Note

This can only be done via `pip` or from source.

Installation via `pip` or from source is done the same way - using environment variables. On Linux/MacOS:

```
export PYOMO_SETUP_ARGS=--with-cython
pip install pyomo
```

On Windows:

```
# Via command prompt
set PYOMO_SETUP_ARGS=--with-cython
# Via powershell
$env:PYOMO_SETUP_ARGS="--with-cython"
pip install pyomo
```

From source (recommended for advanced users only):

```
export PYOMO_SETUP_ARGS=--with-cython
git clone https://github.com/Pyomo/pyomo.git
cd pyomo
# Use -e to install in editable mode
pip install .
```

1.2 Using Solvers with Pyomo

Pyomo supports modeling and scripting but does not install a solver automatically. For numerous reasons (including stability and managing intermittent dependency conflicts), Pyomo does not bundle solvers or have strict dependencies on any third-party solvers. The table below lists a subset of the solvers compatible with Pyomo that can be installed using `pip` or `conda`. It includes both commercial and open-source solvers – users are responsible for understanding the license requirements for their desired solver.

Table 1.1: Available Solvers through pip and conda

Solver	Pip	Conda	License Docs
cplex	`pip install cplex`	`conda install -c \ ibmdecisionoptimization cplex`	License Docs
CPoptimizer	`pip install cplex docplex`	`conda install -c \ ibmdecisionoptimization \ cplex docplex`	License Docs
cuOpt	CUDA-version dependent; see the official documentation.	CUDA-version dependent; see the official documentation.	License Docs
cyipopt	`pip install cyipopt`	`conda install -c conda-forge cyipopt`	License Docs
glpk	N/A	`conda install -c conda-forge glpk`	License Docs
Gurobi	`pip install gurobipy`	`conda install -c gurobi gurobi`	License Docs
HiGHS	`pip install highspy`	`conda install -c conda-forge highspy`	License Docs
KNITRO	`pip install knitro`	N/A	License Docs
MAiNGO	`pip install maingopy`	N/A	License Docs
PyMUMPS	`pip install pymumps`	`conda install -c conda-forge pymumps`	License Docs
SCIP (Command-line)	N/A	`conda install -c conda-forge scip`	(see SCIP project)
SCIP (Python)	`pip install pyscipopt`	`conda install -c conda-forge pyscipopt`	(see SCIP project)
XPRESS	`pip install xpress`	`conda install -c fico-xpress xpress`	License Docs

Note

We compiled this table of solvers to help you get started, but we encourage you to consult the official documentation of your desired solver for the most up-to-date and detailed information.

1.3 Pyomo Overview

1.3.1 Mathematical Modeling

This section provides an introduction to Pyomo: Python Optimization Modeling Objects. A more complete description is contained in the [PyomoBookIII] book. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with commercially available algebraic modeling languages (AMLs) such as [FGK02], [AIMMS], and [GAMS]. Pyomo's modeling objects are embedded within Python, a full-featured, high-level programming language that contains a rich set of supporting libraries.

Modeling is a fundamental process in many aspects of scientific research, engineering and business. Modeling involves the formulation of a simplified representation of a system or real-world object. Thus, modeling tools like Pyomo can be used in a variety of ways:

- *Explain phenomena* that arise in a system,
- *Make predictions* about future states of a system,
- *Assess key factors* that influence phenomena in a system,
- *Identify extreme states* in a system, that might represent worst-case scenarios or minimal cost plans, and
- *Analyze trade-offs* to support human decision makers.

Mathematical models represent system knowledge with a formalized language. The following mathematical concepts are central to modern modeling activities:

Variables

Variables represent unknown or changing parts of a model (e.g., whether or not to make a decision, or the characteristic of a system outcome). The values taken by the variables are often referred to as a *solution* and are usually an output of the optimization process.

Parameters

Parameters represents the data that must be supplied to perform the optimization. In fact, in some settings the word *data* is used in place of the word *parameters*.

Relations

These are equations, inequalities or other mathematical relationships that define how different parts of a model are connected to each other.

Goals

These are functions that reflect goals and objectives for the system being modeled.

The widespread availability of computing resources has made the numerical analysis of mathematical models a commonplace activity. Without a modeling language, the process of setting up input files, executing a solver and extracting the final results from the solver output is tedious and error-prone. This difficulty is compounded in complex, large-scale real-world applications which are difficult to debug when errors occur. Additionally, there are many different formats used by optimization software packages, and few formats are recognized by many optimizers. Thus the application of multiple optimization solvers to analyze a model introduces additional complexities.

Pyomo is an AML that extends Python to include objects for mathematical modeling. [PyomoBookI], [PyomoBookII], [PyomoBookIII], and [Pyomo-paper] compare Pyomo with other AMLs. Although many good AMLs have been developed for optimization models, the following are motivating factors for the development of Pyomo:

- *Open Source*

  Pyomo is developed within Pyomo's open source project to promote transparency of the modeling framework and encourage community development of Pyomo capabilities.

- *Customizable Capability*

  Pyomo supports a customizable capability through the extensive use of plug-ins to modularize software components.

- *Solver Integration*

  Pyomo models can be optimized with solvers that are written either in Python or in compiled, low-level languages.

- *Programming Language*

Pyomo leverages a high-level programming language, which has several advantages over custom AMLs: a very robust language, extensive documentation, a rich set of standard libraries, support for modern programming features like classes and functions, and portability to many platforms.

1.3.2 Overview of Modeling Components and Processes

Pyomo supports an object-oriented design for the definition of optimization models. The basic steps of a simple modeling process are:

- Create model and declare components
- Instantiate the model
- Apply solver
- Interrogate solver results

In practice, these steps may be applied repeatedly with different data or with different constraints applied to the model. However, we focus on this simple modeling process to illustrate different strategies for modeling with Pyomo.

A Pyomo *model* consists of a collection of modeling *components* that define different aspects of the model. Pyomo includes the modeling components that are commonly supported by modern AMLs: index sets, symbolic parameters, decision variables, objectives, and constraints. These modeling components are defined in Pyomo through the following Python classes:

Set

set data that is used to define a model instance

Param

parameter data that is used to define a model instance

Var

decision variables in a model

Objective

expressions that are minimized or maximized in a model

Constraint

constraint expressions that impose restrictions on variable values in a model

1.3.3 Abstract Versus Concrete Models

A mathematical model can be defined using symbols that represent data values. For example, the following equations represent a linear program (LP) to find optimal values for the vector x with parameters n and b, and parameter vectors a and c:

$$\begin{array}{lll} \min & \sum_{j=1}^n c_j x_j & \\ \mathrm{s.t.} & \sum_{j=1}^n a_{ij} x_j \geq b_i & \forall i = 1 \ldots m \\ & x_j \geq 0 & \forall j = 1 \ldots n \end{array}$$

Note

As a convenience, we use the symbol $\forall$ to mean “for all” or “for each.”

We call this an *abstract* or *symbolic* mathematical model since it relies on unspecified parameter values. Data values can be used to specify a *model instance*. The AbstractModel class provides a context for defining and initializing abstract optimization models in Pyomo when the data values will be supplied at the time a solution is to be obtained.

In many contexts, a mathematical model can and should be directly defined with the data values supplied at the time of the model definition. We call these *concrete* mathematical models. For example, the following LP model is a concrete instance of the previous abstract model:

$$\begin{array}{ll} \min & 2x_1 + 3x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{array}$$

The ConcreteModel class is used to define concrete optimization models in Pyomo.

Note

Python programmers will probably prefer to write concrete models, while users of some other algebraic modeling languages may tend to prefer to write abstract models. The choice is largely a matter of taste; some applications may be a little more straightforward using one or the other.

1.3.4 Simple Models

A Simple Concrete Pyomo Model

It is possible to get the same flexible behavior from models declared to be abstract and models declared to be concrete in Pyomo; however, we will focus on a straightforward concrete example here where the data is hard-wired into the model file. Python programmers will quickly realize that the data could have come from other sources.

Given the following model from the previous section:

$$\begin{array}{ll} \min & 2x_1 + 3x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{array}$$

This can be implemented as a concrete model as follows:

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()

model.x = pyo.Var([1,2], domain=pyo.NonNegativeReals)

model.OBJ = pyo.Objective(expr = 2*model.x[1] + 3*model.x[2])

model.Constraint1 = pyo.Constraint(expr = 3*model.x[1] + 4*model.x[2] >= 1)
```

Although rule functions can also be used to specify constraints and objectives, in this example we use the expr option that is available only in concrete models. This option gives a direct specification of the expression.

A Simple Abstract Pyomo Model

We repeat the abstract model from the previous section:

$$\begin{array}{lll} \min & \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall i = 1 \ldots m \\ & x_j \geq 0 & \forall j = 1 \ldots n \end{array}$$

One way to implement this in Pyomo is as shown as follows:

```
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.m = pyo.Param(within=pyo.NonNegativeIntegers)
model.n = pyo.Param(within=pyo.NonNegativeIntegers)

model.I = pyo.RangeSet(1, model.m)
model.J = pyo.RangeSet(1, model.n)

model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)

# the next line declares a variable indexed by the set J
model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals)

def obj_expression(m):
    return pyo.summation(m.c, m.x)

model.OBJ = pyo.Objective(rule=obj_expression)

def ax_constraint_rule(m, i):
    # return the expression for the constraint for i
    return sum(m.a[i,j] * m.x[j] for j in m.J) >= m.b[i]

# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

Note

Python is interpreted one line at a time. A line continuation character, \ (backslash), is used for Python statements that need to span multiple lines. In Python, indentation has meaning and must be consistent. For example, lines inside a function definition must be indented and the end of the indentation is used by Python to signal the end of the definition.

We will now examine the lines in this example. The first import line is required in every Pyomo model. Its purpose is to make the symbols used by Pyomo known to Python.

```
import pyomo.environ as pyo
```

The declaration of a model is also required. The use of the name `model` is not required. Almost any name could be used, but we will use the name `model` in most of our examples. In this example, we are declaring that it will be an abstract model.

```
model = pyo.AbstractModel()
```

We declare the parameters m and n using the Pyomo `Param` component. This component can take a variety of arguments; this example illustrates use of the `within` option that is used by Pyomo to validate the data value that is assigned to the parameter. If this option were not given, then Pyomo would not object to any type of data being assigned to these parameters. As it is, assignment of a value that is not a non-negative integer will result in an error.

```
model.m = pyo.Param(within=pyo.NonNegativeIntegers)
model.n = pyo.Param(within=pyo.NonNegativeIntegers)
```

Although not required, it is convenient to define index sets. In this example we use the `RangeSet` component to declare that the sets will be a sequence of integers starting at 1 and ending at a value specified by the the parameters `model.m` and `model.n`.

```
model.I = pyo.RangeSet(1, model.m)
model.J = pyo.RangeSet(1, model.n)
```

The coefficient and right-hand-side data are defined as indexed parameters. When sets are given as arguments to the `Param` component, they indicate that the set will index the parameter.

```
model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)
```

The next line that is interpreted by Python as part of the model declares the variable x. The first argument to the `Var` component is a set, so it is defined as an index set for the variable. In this case the variable has only one index set, but multiple sets could be used as was the case for the declaration of the parameter `model.a`. The second argument specifies a domain for the variable. This information is part of the model and will passed to the solver when data is provided and the model is solved. Specification of the `NonNegativeReals` domain implements the requirement that the variables be greater than or equal to zero.

```
# the next line declares a variable indexed by the set J
model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals)
```

Note

In Python, and therefore in Pyomo, any text after pound sign is considered to be a comment.

In abstract models, Pyomo expressions are usually provided to objective and constraint declarations via a function defined with a Python `def` statement. The `def` statement establishes a name for a function along with its arguments. When Pyomo uses a function to get objective or constraint expressions, it always passes in the model (i.e., itself) as the the first argument so the model is always the first formal argument when declaring such functions in Pyomo. Additional arguments, if needed, follow. Since summation is an extremely common part of optimization models, Pyomo provides a flexible function to accommodate it. When given two arguments, the `summation()` function returns an expression for the sum of the product of the two arguments over their indexes. This only works, of course, if the two arguments have the same indexes. If it is given only one argument it returns an expression for the sum over all indexes of that argument. So in this example, when `summation()` is passed the arguments `m.c, m.x` it returns an internal representation of the expression $\sum_{j=1}^{n} c_j x_j$.

```
def obj_expression(m):
    return pyo.summation(m.c, m.x)
```

To declare an objective function, the Pyomo component called `Objective` is used. The `rule` argument gives the name of a function that returns the objective expression. The default *sense* is minimization. For maximization, the `sense=pyo.maximize` argument must be used. The name that is declared, which is `OBJ` in this case, appears in some reports and can be almost any name.

```
model.OBJ = pyo.Objective(rule=obj_expression)
```

Declaration of constraints is similar. A function is declared to generate the constraint expression. In this case, there can be multiple constraints of the same form because we index the constraints by i in the expression $\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad \forall i = 1 \ldots m$, which states that we need a constraint for each value of i from one to m. In order to parametrize the expression by i we include it as a formal parameter to the function that declares the constraint expression. Technically, we could have used anything for this argument, but that might be confusing. Using an `i` for an i seems sensible in this situation.

```
def ax_constraint_rule(m, i):
    # return the expression for the constraint for i
    return sum(m.a[i,j] * m.x[j] for j in m.J) >= m.b[i]
```

Note

In Python, indexes are in square brackets and function arguments are in parentheses.

In order to declare constraints that use this expression, we use the Pyomo `Constraint` component that takes a variety of arguments. In this case, our model specifies that we can have more than one constraint of the same form and we have created a set, `model.I`, over which these constraints can be indexed so that is the first argument to the constraint declaration. The next argument gives the rule that will be used to generate expressions for the constraints. Taken as a whole, this constraint declaration says that a list of constraints indexed by the set `model.I` will be created and for each member of `model.I`, the function `ax_constraint_rule` will be called and it will be passed the model object as well as the member of `model.I`

```
# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

In the object oriented view of all of this, we would say that `model` object is a class instance of the `AbstractModel` class, and `model.J` is a `Set` object that is contained by this model. Many modeling components in Pyomo can be optionally specified as *indexed components*: collections of components that are referenced using one or more values. In this example, the parameter `model.c` is indexed with set `model.J`.

In order to use this model, data must be given for the values of the parameters. Here is one file that provides data (in AMPL "`.dat`" format).

```
# one way to input the data in AMPL format
# for indexed parameters, the indexes are given before the value

param m := 1 ;
param n := 2 ;

param a :=
1 1 3
1 2 4
;

param c:=
1 2
2 3
;

param b := 1 1 ;
```

There are multiple formats that can be used to provide data to a Pyomo model, but the AMPL format works well for our purposes because it contains the names of the data elements together with the data. In AMPL data files, text after a

pound sign is treated as a comment. Lines generally do not matter, but statements must be terminated with a semi-colon.

For this particular data file, there is one constraint, so the value of `model.m` will be one and there are two variables (i.e., the vector `model.x` is two elements long) so the value of `model.n` will be two. These two assignments are accomplished with standard assignments. Notice that in AMPL format input, the name of the model is omitted.

```
param m := 1 ;
param n := 2 ;
```

There is only one constraint, so only two values are needed for `model.a`. When assigning values to arrays and vectors in AMPL format, one way to do it is to give the index(es) and the the value. The line 1 2 4 causes `model.a[1,2]` to get the value 4. Since `model.c` has only one index, only one index value is needed so, for example, the line 1 2 causes `model.c[1]` to get the value 2. Line breaks generally do not matter in AMPL format data files, so the assignment of the value for the single index of `model.b` is given on one line since that is easy to read.

```
param a :=
 1 1 3
 1 2 4
 ;

 param c:=
 1 2
 2 3
 ;

 param b := 1 1 ;
```

Symbolic Index Sets

When working with Pyomo (or any other AML), it is convenient to write abstract models in a somewhat more abstract way by using index sets that contain strings rather than index sets that are implied by $1, \ldots, m$ or the summation from 1 to n. When this is done, the size of the set is implied by the input, rather than specified directly. Furthermore, the index entries may have no real order. Often, a mixture of integers and indexes and strings as indexes is needed in the same model. To start with an illustration of general indexes, consider a slightly different Pyomo implementation of the model we just presented.

```
# ___________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________

# abstract2.py


import pyomo.environ as pyo

model = pyo.AbstractModel()

model.I = pyo.Set()
model.J = pyo.Set()
```

(continues on next page)

(continued from previous page)

```
model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)

# the next line declares a variable indexed by the set J
model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals)


def obj_expression(model):
    return pyo.summation(model.c, model.x)


model.OBJ = pyo.Objective(rule=obj_expression)


def ax_constraint_rule(model, i):
    # return the expression for the constraint for i
    return sum(model.a[i, j] * model.x[j] for j in model.J) >= model.b[i]


# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

To get the same instantiated model, the following data file can be used.

```
# abstract2a.dat AMPL format

set I := 1 ;
set J := 1 2 ;

param a :=
1 1 3
1 2 4
;

param c:=
1 2
2 3
;

param b := 1 1 ;
```

However, this model can also be fed different data for problems of the same general form using meaningful indexes.

```
# abstract2.dat AMPL data format

set I := TV Film ;
set J := Graham John Carol ;

param a :=
TV  Graham 3
```

(continues on next page)

(continued from previous page)

```
TV John 4.4
TV Carol 4.9
Film Graham 1
Film John 2.4
Film Carol 1.1
;

param c := [*]
 Graham 2.2
 John 3.1416
 Carol 3
;

param b := TV 1 Film 1 ;
```

Solving the Simple Examples

Pyomo supports modeling and scripting but does not install a solver automatically. In order to solve a model, there must be a solver installed on the computer to be used. If there is a solver, then the `pyomo` command can be used to solve a problem instance.

Suppose that the solver named glpk (also known as glpsol) is installed on the computer. Suppose further that an abstract model is in the file named `abstract1.py` and a data file for it is in the file named `abstract1.dat`. From the command prompt, with both files in the current directory, a solution can be obtained with the command:

```
pyomo solve abstract1.py abstract1.dat --solver=glpk
```

Since glpk is the default solver, there really is no need specify it so the `--solver` option can be dropped.

> **Note**
>
> There are two dashes before the command line option names such as `solver`.

To continue the example, if CPLEX is installed then it can be listed as the solver. The command to solve with CPLEX is

```
pyomo solve abstract1.py abstract1.dat --solver=cplex
```

This yields the following output on the screen:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.07] Creating model
[    0.15] Applying solver
[    0.37] Processing results
Number of solutions: 1
Solution Information
Gap: 0.0
Status: optimal
Function Value: 0.666666666667
Solver results file: results.json
```

(continues on next page)

(continued from previous page)

```
[    0.39] Applying Pyomo postprocessing actions
[    0.39] Pyomo Finished
```

The numbers in square brackets indicate how much time was required for each step. Results are written to the file named `results.json`, which has a special structure that makes it useful for post-processing. To see a summary of results written to the screen, use the `--summary` option:

```
pyomo solve abstract1.py abstract1.dat --solver=cplex --summary
```

To see a list of Pyomo command line options, use:

```
pyomo solve --help
```

Note

There are two dashes before `help`.

For a concrete model, no data file is specified on the Pyomo command line.

CHAPTER

TWO

HOW-TO GUIDES

2.1 Interrogating Models

2.1.1 Accessing Variable Values

Primal Variable Values

Often, the point of optimization is to get optimal values of variables. Some users may want to process the values in a script. We will describe how to access a particular variable from a Python script as well as how to access all variables from a Python script and from a callback. This should enable the reader to understand how to get the access that they desire. The Iterative example given above also illustrates access to variable values.

One Variable from a Python Script

Assuming the model has been instantiated and solved and the results have been loaded back into the instance object, then we can make use of the fact that the variable is a member of the instance object and its value can be accessed using its `value` member. For example, suppose the model contains a variable named `quant` that is a singleton (has no indexes) and suppose further that the name of the instance object is `instance`. Then the value of this variable can be accessed using `pyo.value(instance.quant)`. Variables with indexes can be referenced by supplying the index.

Consider the following very simple example, which is similar to the iterative example. This is a concrete model. In this example, the value of `x[2]` is accessed.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# noiteration1.py

import pyomo.environ as pyo
from pyomo.opt import SolverFactory

# Create a solver
opt = SolverFactory('glpk')

#
# A simple model with binary variables and
```

(continues on next page)

(continued from previous page)

```
# an empty constraint list.
#
model = pyo.ConcreteModel()
model.n = pyo.Param(default=4)
model.x = pyo.Var(pyo.RangeSet(model.n), within=pyo.Binary)


def o_rule(model):
    return pyo.summation(model.x)


model.o = pyo.Objective(rule=o_rule)
model.c = pyo.ConstraintList()

results = opt.solve(model)

if pyo.value(model.x[2]) == 0:
    print("The second index has a zero")
else:
    print("x[2]=", pyo.value(model.x[2]))
```

Note

If this script is run without modification, Pyomo is likely to issue a warning because there are no constraints. The warning is because some solvers may fail if given a problem instance that does not have any constraints.

All Variables from a Python Script

As with one variable, we assume that the model has been instantiated and solved. Assuming the instance object has the name `instance`, the following code snippet displays all variables and their values:

```
>>> for v in instance.component_objects(pyo.Var, active=True):
...     print("Variable",v)
...     for index in v:
...         print ("   ",index, pyo.value(v[index]))
```

Alternatively,

```
>>> for v in instance.component_data_objects(pyo.Var, active=True):
...     print(v, pyo.value(v))
```

This code could be improved by checking to see if the variable is not indexed (i.e., the only index value is `None`), then the code could print the value without the word `None` next to it.

Assuming again that the model has been instantiated and solved and the results have been loaded back into the instance object. Here is a code snippet for fixing all integers at their current value:

```
>>> for var in instance.component_data_objects(pyo.Var, active=True):
...     if not var.is_continuous():
...         print ("fixing "+str(v))
...         var.fixed = True # fix the current value
```

Another way to access all of the variables (particularly if there are blocks) is as follows (this particular snippet assumes that instead of *import pyomo.environ as pyo from pyo.environ import ** was used):

```
for v in model.component_objects(pyo.Var, descend_into=True):
    print("FOUND VAR:" + v.name)
    v.pprint()

for v_data in model.component_data_objects(pyo.Var, descend_into=True):
    print("Found: " + v_data.name + ", value = " + str(pyo.value(v_data)))
```

2.1.2 Accessing Parameter Values

Accessing parameter values is completely analogous to accessing variable values. For example, here is a code snippet to print the name and value of every Parameter in a model:

```
>>> for parmobject in instance.component_objects(pyo.Param, active=True):
...     nametoprint = str(str(parmobject.name))
...     print ("Parameter ", nametoprint)
...     for index in parmobject:
...         vtoprint = pyo.value(parmobject[index])
...         print ("   ",index, vtoprint)
```

2.1.3 Accessing Duals

Access to dual values in scripts is similar to accessing primal variable values, except that dual values are not captured by default so additional directives are needed before optimization to signal that duals are desired.

To get duals without a script, use the `pyomo` option `--solver-suffixes='dual'` which will cause dual values to be included in output. Note: In addition to duals (`dual`) , reduced costs (`rc`) and slack values (`slack`) can be requested. All suffixes can be requested using the `pyomo` option `--solver-suffixes='.*'`

Warning

Some of the duals may have the value `None`, rather than `0`.

Access Duals in a Python Script

To signal that duals are desired, declare a Suffix component with the name "dual" on the model or instance with an IMPORT or IMPORT_EXPORT direction.

```
# Create a 'dual' suffix component on the instance
# so the solver plugin will know which suffixes to collect
instance.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)
```

See the section on Suffixes *Suffixes* for more information on Pyomo's Suffix component. After the results are obtained and loaded into an instance, duals can be accessed in the following fashion.

```
# display all duals
print("Duals")
for c in instance.component_objects(pyo.Constraint, active=True):
    print("   Constraint", c)
    for index in c:
        print("      ", index, instance.dual[c[index]])
```

The following snippet will only work, of course, if there is a constraint with the name `AxbConstraint` that has and index, which is the string `Film`.

```
# access one dual
print("Dual for Film=", instance.dual[instance.AxbConstraint['Film']])
```

Here is a complete example that relies on the file `abstract2.py` to provide the model and the file `abstract2.dat` to provide the data. Note that the model in `abstract2.py` does contain a constraint named `AxbConstraint` and `abstract2.dat` does specify an index for it named `Film`.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# driveabs2.py

import pyomo.environ as pyo
from pyomo.opt import SolverFactory

# Create a solver
opt = SolverFactory('cplex')

# get the model from another file
from abstract2 import model

# Create a model instance and optimize
instance = model.create_instance('abstract2.dat')

# Create a 'dual' suffix component on the instance
# so the solver plugin will know which suffixes to collect
instance.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

results = opt.solve(instance)
# also puts the results back into the instance for easy access

# display all duals
print("Duals")
for c in instance.component_objects(pyo.Constraint, active=True):
    print("   Constraint", c)
    for index in c:
        print("      ", index, instance.dual[c[index]])

# access one dual
print("Dual for Film=", instance.dual[instance.AxbConstraint['Film']])
```

Concrete models are slightly different because the model is the instance. Here is a complete example that relies on the file `concrete1.py` to provide the model and instantiate it.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# driveconc1.py

import pyomo.environ as pyo
from pyomo.opt import SolverFactory

# Create a solver
opt = SolverFactory('cplex')

# get the model from another file
from concrete1 import model

# Create a 'dual' suffix component on the instance
# so the solver plugin will know which suffixes to collect
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

results = opt.solve(model)  # also load results to model

# display all duals
print("Duals")
for c in model.component_objects(pyo.Constraint, active=True):
    print("   Constraint", c)
    for index in c:
        print("      ", index, model.dual[c[index]])
```

2.1.4 Accessing Slacks

The functions `lslack()` and `uslack()` return the upper and lower slacks, respectively, for a constraint.

2.2 Manipulating Pyomo Models

This section gives an overview of commonly used scripting commands when working with Pyomo models. These commands must be applied to a concrete model instance or in other words an instantiated model.

2.2.1 Repeated Solves

```
>>> import pyomo.environ as pyo
>>> from pyomo.opt import SolverFactory
>>> model = pyo.ConcreteModel()
>>> model.nVars = pyo.Param(initialize=4)
>>> model.N = pyo.RangeSet(model.nVars)
>>> model.x = pyo.Var(model.N, within=pyo.Binary)
>>> model.obj = pyo.Objective(expr=pyo.summation(model.x))
```

(continues on next page)

(continued from previous page)

```
>>> model.cuts = pyo.ConstraintList()
>>> opt = SolverFactory('glpk')
>>> opt.solve(model)

>>> # Iterate, adding a cut to exclude the previously found solution
>>> for i in range(5):
...     expr = 0
...     for j in model.x:
...         if pyo.value(model.x[j]) < 0.5:
...             expr += model.x[j]
...         else:
...             expr += (1 - model.x[j])
...     model.cuts.add( expr >= 1 )
...     results = opt.solve(model)
...     print ("\n===== iteration",i)
...     model.display()
```

To illustrate Python scripts for Pyomo we consider an example that is in the file `iterative1.py` and is executed using the command

```
python iterative1.py
```

Note

This is a Python script that contains elements of Pyomo, so it is executed using the `python` command. The `pyomo` command can be used, but then there will be some strange messages at the end when Pyomo finishes the script and attempts to send the results to a solver, which is what the `pyomo` command does.

This script creates a model, solves it, and then adds a constraint to preclude the solution just found. This process is repeated, so the script finds and prints multiple solutions. The particular model it creates is just the sum of four binary variables. One does not need a computer to solve the problem or even to iterate over solutions. This example is provided just to illustrate some elementary aspects of scripting.

```
# ____________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ____________________________________________________________________________________

# iterative1.py
import pyomo.environ as pyo
from pyomo.opt import SolverFactory


# Create a solver
opt = pyo.SolverFactory('glpk')

#
```

(continues on next page)

(continued from previous page)

```
# A simple model with binary variables and
# an empty constraint list.
#
model = pyo.AbstractModel()
model.n = pyo.Param(default=4)
model.x = pyo.Var(pyo.RangeSet(model.n), within=pyo.Binary)


def o_rule(model):
    return pyo.summation(model.x)


model.o = pyo.Objective(rule=o_rule)
model.c = pyo.ConstraintList()

# Create a model instance and optimize
instance = model.create_instance()
results = opt.solve(instance)
instance.display()

# Iterate to eliminate the previously found solution
for i in range(5):
    expr = 0
    for j in instance.x:
        if pyo.value(instance.x[j]) == 0:
            expr += instance.x[j]
        else:
            expr += 1 - instance.x[j]
    instance.c.add(expr >= 1)
    results = opt.solve(instance)
    print("\n===== iteration", i)
    instance.display()
```

Let us now analyze this script. The first line is a comment that happens to give the name of the file. This is followed by two lines that import symbols for Pyomo. The pyomo namespace is imported as `pyo`. Therefore, `pyo.` must precede each use of a Pyomo name.

```
# iterative1.py
import pyomo.environ as pyo
from pyomo.opt import SolverFactory
```

An object to perform optimization is created by calling `SolverFactory` with an argument giving the name of the solver. The argument would be `'gurobi'` if, e.g., Gurobi was desired instead of glpk:

```
# Create a solver
opt = pyo.SolverFactory('glpk')
```

The next lines after a comment create a model. For our discussion here, we will refer to this as the base model because it will be extended by adding constraints later. (The words "base model" are not reserved words, they are just being introduced for the discussion of this example). There are no constraints in the base model, but that is just to keep it simple. Constraints could be present in the base model. Even though it is an abstract model, the base model is fully specified by these commands because it requires no external data:

```
model = pyo.AbstractModel()
model.n = pyo.Param(default=4)
model.x = pyo.Var(pyo.RangeSet(model.n), within=pyo.Binary)


def o_rule(model):
    return pyo.summation(model.x)


model.o = pyo.Objective(rule=o_rule)
```

The next line is not part of the base model specification. It creates an empty constraint list that the script will use to add constraints.

```
model.c = pyo.ConstraintList()
```

The next non-comment line creates the instantiated model and refers to the instance object with a Python variable `instance`. Models run using the `pyomo` script do not typically contain this line because model instantiation is done by the `pyomo` script. In this example, the `create` function is called without arguments because none are needed; however, the name of a file with data commands is given as an argument in many scripts.

```
instance = model.create_instance()
```

The next line invokes the solver and refers to the object contain results with the Python variable `results`.

```
results = opt.solve(instance)
```

The solve function loads the results into the instance, so the next line writes out the updated values.

```
instance.display()
```

The next non-comment line is a Python iteration command that will successively assign the integers from 0 to 4 to the Python variable `i`, although that variable is not used in script. This loop is what causes the script to generate five more solutions:

```
for i in range(5):
```

An expression is built up in the Python variable named `expr`. The Python variable `j` will be iteratively assigned all of the indexes of the variable `x`. For each index, the value of the variable (which was loaded by the `load` method just described) is tested to see if it is zero and the expression in `expr` is augmented accordingly. Although `expr` is initialized to 0 (an integer), its type will change to be a Pyomo expression when it is assigned expressions involving Pyomo variable objects:

```
    expr = 0
    for j in instance.x:
        if pyo.value(instance.x[j]) == 0:
            expr += instance.x[j]
        else:
            expr += 1 - instance.x[j]
```

During the first iteration (when `i` is 0), we know that all values of `x` will be 0, so we can anticipate what the expression will look like. We know that `x` is indexed by the integers from 1 to 4 so we know that `j` will take on the values from 1 to 4 and we also know that all value of `x` will be zero for all indexes so we know that the value of `expr` will be something like

```
0 + instance.x[1] + instance.x[2] + instance.x[3] + instance.x[4]
```

The value of j will be evaluated because it is a Python variable; however, because it is a Pyomo variable, the value of `instance.x[j]` not be used, instead the variable object will appear in the expression. That is exactly what we want in this case. When we wanted to use the current value in the `if` statement, we used the `value` function to get it.

The next line adds to the constraint list called `c` the requirement that the expression be greater than or equal to one:

```
    instance.c.add(expr >= 1)
```

The proof that this precludes the last solution is left as an exercise for the reader.

The final lines in the outer for loop find a solution and display it:

```
    results = opt.solve(instance)
    print("\n===== iteration", i)
    instance.display()
```

Note

The assignment of the solve output to a results object is somewhat anachronistic. Many scripts just use

```
>>> opt.solve(instance)
```

since the results are moved to the instance by default, leaving the results object with little of interest. If, for some reason, you want the results to stay in the results object and *not* be moved to the instance, you would use

```
>>> results = opt.solve(instance, load_solutions=False)
```

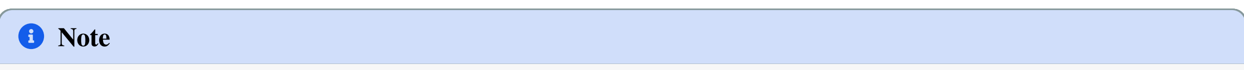
This approach can be useful if there is a concern that the solver did not terminate with an optimal solution. For example,

```
>>> results = opt.solve(instance, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     instance.solutions.load_from(results)
```

2.2.2 Changing the Model or Data and Re-solving

The `iterative1.py` example above illustrates how a model can be changed and then re-solved. In that example, the model is changed by adding a constraint, but the model could also be changed by altering the values of parameters. Note, however, that in these examples, we make the changes to the concrete model instances. This is particularly important for `AbstractModel` users, as this implies working with the `instance` object rather than the `model` object, which allows us to avoid creating a new `model` object for each solve. Here is the basic idea for users of an `AbstractModel`:

1. Create an `AbstractModel` (suppose it is called `model`)
2. Call `model.create_instance()` to create an instance (suppose it is called `instance`)
3. Solve `instance`
4. Change something in `instance`
5. Solve `instance` again



Note

Users of `ConcreteModel` typically name their models `model`, which can cause confusion to novice readers of documentation. Examples based on an `AbstractModel` will refer to `instance` where users of a `ConcreteModel` would typically use the name `model`.

If `instance` has a parameter whose name is `Theta` that was declared to be `mutable` (i.e., `mutable=True`) with an index that contains `idx`, then the value in `NewVal` can be assigned to it using

```
>>> instance.Theta[idx] = NewVal
```

or, more explicitly using the `set_value()` method:

```
>>> instance.Theta[idx].set_value(NewVal)
```

For a singleton parameter named `sigma` (i.e., if it is not indexed), the assignment can be made using

```
>>> instance.sigma = NewVal
```

or

```
>>> instance.sigma.set_value(NewVal)
```

Common Pitfalls: Updating Immutable Parameters

A common mistake is trying to update a parameter that was not declared as mutable. For example:

```
model.p = pyo.Param(initialize=10)  # mutable=False by default
model.p = 5                         # Raises TypeError
```

This will raise a `TypeError` because Pyomo has already "baked" the value `10` into the model's expressions. To allow updates, you **must** set `mutable=True` during declaration:

```
model.p = pyo.Param(initialize=10, mutable=True)
model.p.set_value(5)                # This works!
```

Note

While direct assignment (e.g., `model.p = 5`) works for mutable parameters, using `set_value()` is often clearer as it explicitly signals that you are updating a Pyomo component rather than just a Python attribute.

For more information about access to Pyomo parameters, see the section in this document on `Param` access *Accessing Parameter Values*. Note that for concrete models, the model is the instance.

2.2.3 Fixing Variables and Re-solving

Instead of changing model data, scripts are often used to fix variable values. The following example illustrates this.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
```

(continues on next page)

(continued from previous page)

```
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# iterative2.py

import pyomo.environ as pyo

# Create a solver
opt = pyo.SolverFactory('cplex')

#
# A simple model with binary variables and
# an empty constraint list.
#
model = pyo.AbstractModel()
model.n = pyo.Param(default=4)
model.x = pyo.Var(pyo.RangeSet(model.n), within=pyo.Binary)


def o_rule(model):
    return pyo.summation(model.x)


model.o = pyo.Objective(rule=o_rule)
model.c = pyo.ConstraintList()

# Create a model instance and optimize
instance = model.create_instance()
results = opt.solve(instance)
instance.display()

# "flip" the value of x[2] (it is binary)
# then solve again

if pyo.value(instance.x[2]) == 0:
    instance.x[2].fix(1)
else:
    instance.x[2].fix(0)

results = opt.solve(instance)
instance.display()
```

In this example, the variables are binary. The model is solved and then the value of `model.x[2]` is flipped to the opposite value before solving the model again. The main lines of interest are:

```
if pyo.value(instance.x[2]) == 0:
    instance.x[2].fix(1)
else:
    instance.x[2].fix(0)

results = opt.solve(instance)
```

This could also have been accomplished by setting the upper and lower bounds:

```
>>> if instance.x[2].value == 0:
...     instance.x[2].setlb(1)
...     instance.x[2].setub(1)
... else:
...     instance.x[2].setlb(0)
...     instance.x[2].setub(0)
```

Notice that when using the bounds, we do not set `fixed` to `True` because that would fix the variable at whatever value it presently has and then the bounds would be ignored by the solver.

For more information about access to Pyomo variables, see the section in this document on `Var` access *Accessing Variable Values*.

Note that

```
>>> instance.x.fix(1)
```

is equivalent to

```
>>> instance.x.value = 1
>>> instance.x.fixed = True
```

and

```
>>> instance.x.fix()
```

is equivalent to

```
>>> instance.x.fixed = True
```

2.2.4 Extending the Objective Function

One can add terms to an objective function of a `ConcreteModel` (or and instantiated `AbstractModel`) using the `expr` attribute of the objective function object. Here is a simple example:

```
>>> import pyomo.environ as pyo
>>> from pyomo.opt import SolverFactory

>>> model = pyo.ConcreteModel()

>>> model.x = pyo.Var(within=pyo.PositiveReals)
>>> model.y = pyo.Var(within=pyo.PositiveReals)

>>> model.sillybound = pyo.Constraint(expr = model.x + model.y <= 2)

>>> model.obj = pyo.Objective(expr = 20 * model.x)

>>> opt = SolverFactory('glpk')
>>> opt.solve(model)

>>> model.pprint()

>>> print ("------------- extend obj --------------")
```

(continues on next page)

(continued from previous page)

```
>>> model.obj.expr += 10 * model.y

>>> opt.solve(model)
>>> model.pprint()
```

2.2.5 Activating and Deactivating Objectives

Multiple objectives can be declared, but only one can be active at a time (at present, Pyomo does not support any solvers that can be given more than one objective). If both `model.obj1` and `model.obj2` have been declared using `Objective`, then one can ensure that `model.obj2` is passed to the solver as shown in this simple example:

```
>>> model = pyo.ConcreteModel()
>>> model.obj1 = pyo.Objective(expr = 0)
>>> model.obj2 = pyo.Objective(expr = 0)

>>> model.obj1.deactivate()
>>> model.obj2.activate()
```

For abstract models this would be done prior to instantiation or else the `activate` and `deactivate` calls would be on the instance rather than the model.

2.2.6 Activating and Deactivating Constraints

Constraints can be temporarily disabled using the `deactivate()` method. When the model is sent to a solver inactive constraints are not included. Disabled constraints can be re-enabled using the `activate()` method.

```
>>> model = pyo.ConcreteModel()
>>> model.v = pyo.Var()
>>> model.con = pyo.Constraint(expr=model.v**2 + model.v >= 3)
>>> model.con.deactivate()
>>> model.con.activate()
```

Indexed constraints can be deactivated/activated as a whole or by individual index:

```
>>> model = pyo.ConcreteModel()
>>> model.s = pyo.Set(initialize=[1,2,3])
>>> model.v = pyo.Var(model.s)
>>> def _con(m, s):
...    return m.v[s]**2 + m.v[s] >= 3
>>> model.con = pyo.Constraint(model.s, rule=_con)
>>> model.con.deactivate()   # Deactivate all indices
>>> model.con[1].activate()  # Activate single index
```

2.3 Solver Recipes

2.3.1 Accessing Solver Status

After a solve, the results object has a member `Solution.Status` that contains the solver status. The following snippet shows an example of access via a `print` statement:

```
results = opt.solve(instance)
#print ("The solver returned a status of:"+str(results.solver.status))
```

The use of the Python `str` function to cast the value to a be string makes it easy to test it. In particular, the value 'optimal' indicates that the solver succeeded. It is also possible to access Pyomo data that can be compared with the solver status as in the following code snippet:

```
from pyomo.opt import SolverStatus, TerminationCondition

#...

if (results.solver.status == SolverStatus.ok) and (results.solver.termination_condition␣
↪== TerminationCondition.optimal):
     print ("this is feasible and optimal")
elif results.solver.termination_condition == TerminationCondition.infeasible:
     print ("do something about it? or exit?")
else:
     # something else is wrong
     print (str(results.solver))
```

Alternatively,

```
from pyomo.opt import TerminationCondition

...

results = opt.solve(model, load_solutions=False)
if results.solver.termination_condition == TerminationCondition.optimal:
    model.solutions.load_from(results)
else:
    print ("Solution is not optimal")
    # now do something about it? or exit? ...
```

2.3.2 Display of Solver Output

To see the output of the solver, use the option `tee=True` as in

```
results = opt.solve(instance, tee=True)
```

This can be useful for troubleshooting solver difficulties.

2.3.3 Sending Options to the Solver

Most solvers accept options and Pyomo can pass options through to a solver. In scripts or callbacks, the options can be attached to the solver object by adding to its options dictionary as illustrated by this snippet:

```
optimizer = pyo.SolverFactory['cbc']
optimizer.options["threads"] = 4
```

If multiple options are needed, then multiple dictionary entries should be added.

Sometimes it is desirable to pass options as part of the call to the solve function as in this snippet:

```
results = optimizer.solve(instance, options={'threads' : 4}, tee=True)
```

The quoted string is passed directly to the solver. If multiple options need to be passed to the solver in this way, they should be separated by a space within the quoted string. Notice that `tee` is a Pyomo option and is solver-independent,

while the string argument to `options` is passed to the solver without very little processing by Pyomo. If the solver does not have a "threads" option, it will probably complain, but Pyomo will not.

There are no default values for options on a `SolverFactory` object. If you directly modify its options dictionary, as was done above, those options will persist across every call to `optimizer.solve(...)` unless you delete them from the options dictionary. You can also pass a dictionary of options into the `opt.solve(...)` method using the `options` keyword. Those options will only persist within that solve and temporarily override any matching options in the options dictionary on the solver object.

2.3.4 Specifying the Path to a Solver

Often, the executables for solvers are in the path; however, for situations where they are not, the SolverFactory function accepts the keyword `executable`, which you can use to set an absolute or relative path to a solver executable. E.g.,

```
opt = pyo.SolverFactory("ipopt", executable="../ipopt")
```

2.3.5 Warm Starts

Some solvers support a warm start based on current values of variables. To use this feature, set the values of variables in the instance and pass `warmstart=True` to the `solve()` method. E.g.,

```
instance = model.create()
instance.y[0] = 1
instance.y[1] = 0

opt = pyo.SolverFactory("cplex")

results = opt.solve(instance, warmstart=True)
```

Note

The Cplex and Gurobi LP file (and Python) interfaces will generate an MST file with the variable data and hand this off to the solver in addition to the LP file.

Warning

Solvers using the NL file interface (e.g., "gurobi_ampl", "cplexamp") do not accept warmstart as a keyword to the solve() method as the NL file format, by default, includes variable initialization data (drawn from the current value of all variables).

2.3.6 Solving Multiple Instances in Parallel

Building and solving Pyomo models in parallel is a common requirement for many applications. We recommend using MPI for Python (mpi4py) for this purpose. For more information on mpi4py, see the mpi4py documentation (https://mpi4py.readthedocs.io/en/stable/). The example below demonstrates how to use mpi4py to solve two pyomo models in parallel. The example can be run with the following command:

```
mpirun -np 2 python -m mpi4py parallel.py
```

```
# ___________________________________________________________________________________
#
```

(continues on next page)

(continued from previous page)

```
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# parallel.py
# run with mpirun -np 2 python -m mpi4py parallel.py
import pyomo.environ as pyo
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
assert (
    size == 2
), 'This example only works with 2 processes; please us mpirun -np 2 python -m mpi4py␣
↪parallel.py'

# Create a solver
opt = pyo.SolverFactory('cplex_direct')

#
# A simple model with binary variables
#
model = pyo.ConcreteModel()
model.n = pyo.Param(initialize=4)
model.x = pyo.Var(pyo.RangeSet(model.n), within=pyo.Binary)
model.obj = pyo.Objective(expr=sum(model.x.values()))

if rank == 1:
    model.x[1].fix(1)

results = opt.solve(model)
print('rank: ', rank, '    objective: ', pyo.value(model.obj.expr))
```

2.3.7 Changing the temporary directory

A "temporary" directory is used for many intermediate files. Normally, the name of the directory for temporary files is provided by the operating system, but the user can specify their own directory name. The pyomo command-line `--tempdir` option propagates through to the TempFileManager service. One can accomplish the same through the following few lines of code in a script:

```
from pyomo.common.tempfiles import TempfileManager
TempfileManager.tempdir = YourDirectoryNameGoesHere
```

2.4 Working with Abstract Models

2.4.1 Instantiating Models

If you start with a `ConcreteModel`, each component you add to the model will be fully constructed and initialized at the time it attached to the model. However, if you are starting with an `AbstractModel`, construction occurs in two

phases. When you first declare and attach components to the model, those components are empty containers and *not* fully constructed, even if you explicitly provide data.

```
>>> import pyomo.environ as pyo
>>> model = pyo.AbstractModel()
>>> model.is_constructed()
False

>>> model.p = pyo.Param(initialize=5)
>>> model.p.is_constructed()
False

>>> model.I = pyo.Set(initialize=[1,2,3])
>>> model.x = pyo.Var(model.I)
>>> model.x.is_constructed()
False
```

If you look at the model at this point, you will see that everything is "empty":

```
>>> model.pprint()
1 Set Declarations
    I : Size=0, Index=None, Ordered=Insertion
        Not constructed

1 Param Declarations
    p : Size=0, Index=None, Domain=Any, Default=None, Mutable=False
        Not constructed

1 Var Declarations
    x : Size=0, Index=I
        Not constructed

3 Declarations: p I x
```

Before you can manipulate modeling components or solve the model, you must first create a concrete *instance* by applying data to your abstract model. This can be done using the create_instance() method, which takes the abstract model and optional data and returns a new *concrete* instance by constructing each of the model components in the order in which they were declared (attached to the model). Note that the instance creation is performed "out of place"; that is, the original abstract model is left untouched.

```
>>> instance = model.create_instance()
>>> model.is_constructed()
False
>>> type(instance)
<class 'pyomo.core.base.PyomoModel.ConcreteModel'>
>>> instance.is_constructed()
True
>>> instance.pprint()
1 Set Declarations
    I : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {1, 2, 3}

1 Param Declarations
```

(continues on next page)

(continued from previous page)

```
    p : Size=1, Index=None, Domain=Any, Default=None, Mutable=False
        Key  : Value
        None :     5

1 Var Declarations
    x : Size=3, Index=I
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          1 :  None :  None :  None : False :  True :  Reals
          2 :  None :  None :  None : False :  True :  Reals
          3 :  None :  None :  None : False :  True :  Reals

3 Declarations: p I x
```

Note

AbstractModel users should note that in some examples, your concrete model instance is called "*instance*" and not "*model*". This is the case here, where we are explicitly calling `instance = model.create_instance()`.

The `create_instance()` method can also take a reference to external data, which overrides any data specified in the original component declarations. The data can be provided from several sources, including using a *dict*, *DataPortal*, or *DAT file*. For example:

```
>>> instance2 = model.create_instance({None: {'I': {None: [4,5]}}})
>>> instance2.pprint()
1 Set Declarations
    I : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    2 : {4, 5}

1 Param Declarations
    p : Size=1, Index=None, Domain=Any, Default=None, Mutable=False
        Key  : Value
        None :     5

1 Var Declarations
    x : Size=2, Index=I
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          4 :  None :  None :  None : False :  True :  Reals
          5 :  None :  None :  None : False :  True :  Reals

3 Declarations: p I x
```

2.4.2 Managing Data in AbstractModels

There are roughly three ways of using data to construct a Pyomo model:

1. use standard Python objects,
2. initialize a model with data loaded with a `DataPortal` object, and
3. load model data from a Pyomo data command file.

Standard Python data objects include native Python data types (e.g. lists, sets, and dictionaries) as well as standard

data formats like numpy arrays and Pandas data frames. Standard Python data objects can be used to define constant values in a Pyomo model, and they can be used to initialize `Set` and `Param` components. However, initializing `Set` and `Param` components in this manner provides few advantages over direct use of standard Python data objects. (An import exception is that components indexed by `Set` objects use less memory than components indexed by native Python data.)

The `DataPortal` class provides a generic facility for loading data from disparate sources. A `DataPortal` object can load data in a consistent manner, and this data can be used to simply initialize all `Set` and `Param` components in a model. `DataPortal` objects can be used to initialize both concrete and abstract models in a uniform manner, which is important in some scripting applications. But in practice, this capability is only necessary for abstract models, whose data components are initialized after being constructed. (In fact, all abstract data components in an abstract model are loaded from `DataPortal` objects.)

Finally, Pyomo data command files provide a convenient mechanism for initializing `Set` and `Param` components with a high-level data specification. Data command files can be used with both concrete and abstract models, though in a different manner. Data command files are parsed using a `DataPortal` object, which must be done explicitly for a concrete model. However, abstract models can load data from a data command file directly, after the model is constructed. Again, this capability is only necessary for abstract models, whose data components are initialized after being constructed.

The following sections provide more detail about how data can be used to initialize Pyomo models.

Using Standard Data Types

Defining Constant Values

In many cases, Pyomo models can be constructed without `Set` and `Param` data components. Native Python data types class can be simply used to define constant values in Pyomo expressions. Consequently, Python sets, lists and dictionaries can be used to construct Pyomo models, as well as a wide range of other Python classes.

TODO

More examples here: set, list, dict, numpy, pandas.

Initializing Set and Parameter Components

The `Set` and `Param` components used in a Pyomo model can also be initialized with standard Python data types. This enables some modeling efficiencies when manipulating sets (e.g. when re-using sets for indices), and it supports validation of set and parameter data values. The `Set` and `Param` components are initialized with Python data using the `initialize` option.

Set Components

In general, `Set` components can be initialized with iterable data. For example, simple sets can be initialized with:

- list, set and tuple data:

```
model.A = pyo.Set(initialize=[2, 3, 5])
model.B = pyo.Set(initialize=set([2, 3, 5]))
model.C = pyo.Set(initialize=(2, 3, 5))
```

- generators:

```
model.D = pyo.Set(initialize=range(9))
model.E = pyo.Set(initialize=(i for i in model.B if i % 2 == 0))
```

- numpy arrays:

```
f = numpy.array([2, 3, 5])
model.F = pyo.Set(initialize=f)
```

Sets can also be indirectly initialized with functions that return native Python data:

```
def g(model):
    return [2, 3, 5]


model.G = pyo.Set(initialize=g)
```

Indexed sets can be initialized with dictionary data where the dictionary values are iterable data:

```
H_init = {}
H_init[2] = [1, 3, 5]
H_init[3] = [2, 4, 6]
H_init[4] = [3, 5, 7]
model.H = pyo.Set([2, 3, 4], initialize=H_init)
```

Parameter Components

When a parameter is a single value, then a `Param` component can be simply initialized with a value:

```
model.a = pyo.Param(initialize=1.1)
```

More generally, `Param` components can be initialized with dictionary data where the dictionary values are single values:

```
model.b = pyo.Param([1, 2, 3], initialize={1: 1, 2: 2, 3: 3})
```

Parameters can also be indirectly initialized with functions that return native Python data:

```
def c(model):
    return {1: 1, 2: 2, 3: 3}


model.c = pyo.Param([1, 2, 3], initialize=c)
```

Using a Python Dictionary

Data can be passed to the model `create_instance()` method through a series of nested native Python dictionaries. The structure begins with a dictionary of *namespaces*, with the only required entry being the `None` namespace. Each namespace contains a dictionary that maps component names to dictionaries of component values. For scalar components, the required data dictionary maps the implicit index `None` to the desired value:

```
>>> import pyomo.environ as pyo
>>> m = pyo.AbstractModel()
>>> m.I = pyo.Set()
>>> m.p = pyo.Param()
>>> m.q = pyo.Param(m.I)
>>> m.r = pyo.Param(m.I, m.I, default=0)
>>> data = {None: {
...     'I': {None: [1,2,3]},
...     'p': {None: 100},
```

(continues on next page)

(continued from previous page)

```
...     'q': {1: 10, 2:20, 3:30},
...     'r': {(1,1): 110, (1,2): 120, (2,3): 230},
... }}
>>> i = m.create_instance(data)
>>> i.pprint()
1 Set Declarations
    I : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {1, 2, 3}

3 Param Declarations
    p : Size=1, Index=None, Domain=Any, Default=None, Mutable=False
        Key  : Value
        None :   100
    q : Size=3, Index=I, Domain=Any, Default=None, Mutable=False
        Key : Value
          1 :    10
          2 :    20
          3 :    30
    r : Size=9, Index=I*I, Domain=Any, Default=0, Mutable=False
        Key    : Value
        (1, 1) :   110
        (1, 2) :   120
        (2, 3) :   230

4 Declarations: I p q r
```

Data Command Files

Note

The discussion and presentation below are adapted from Chapter 6 of the second edition of the “Pyomo Book” [PyomoBookII]. The discussion of the `DataPortal` class uses these same examples to illustrate how data can be loaded into Pyomo models within Python scripts (see the *Data Portals* section).

Model Data

Pyomo’s *data command files* employ a domain-specific language whose syntax closely resembles the syntax of AMPL’s data commands [FGK02]. A data command file consists of a sequence of commands that either (a) specify set and parameter data for a model, or (b) specify where such data is to be obtained from external sources (e.g. table files, CSV files, spreadsheets and databases).

The following commands are used to declare data:

- The `set` command declares set data.
- The `param` command declares a table of parameter data, which can also include the declaration of the set data used to index the parameter data.
- The `table` command declares a two-dimensional table of parameter data.
- The `load` command defines how set and parameter data is loaded from external data sources, including ASCII table files, CSV files, XML files, YAML files, JSON files, ranges in spreadsheets, and database tables.

The following commands are also used in data command files:

- The `include` command specifies a data command file that is processed immediately.
- The `data` and `end` commands do not perform any actions, but they provide compatibility with AMPL scripts that define data commands.
- The `namespace` keyword allows data commands to be organized into named groups that can be enabled or disabled during model construction.

The following data types can be represented in a data command file:

- **Numeric value**: Any Python numeric value (e.g. integer, float, scientific notation, or boolean).
- **Simple string**: A sequence of alpha-numeric characters.
- **Quoted string**: A simple string that is included in a pair of single or double quotes. A quoted string can include quotes within the quoted string.

Numeric values are automatically converted to Python integer or floating point values when a data command file is parsed. Additionally, if a quoted string can be interpreted as a numeric value, then it will be converted to Python numeric types when the data is parsed. For example, the string "100" is converted to a numeric value automatically.

Warning

Pyomo data commands do *not* exactly correspond to AMPL data commands. The `set` and `param` commands are designed to closely match AMPL's syntax and semantics, though these commands only support a subset of the corresponding declarations in AMPL. However, other Pyomo data commands are not generally designed to match the semantics of AMPL.

Note

Pyomo data commands are terminated with a semicolon, and the syntax of data commands does not depend on whitespace. Thus, data commands can be broken across multiple lines – newlines and tab characters are ignored – and data commands can be formatted with whitespace with few restrictions.

The `set` Command

Simple Sets

The `set` data command explicitly specifies the members of either a single set or an array of sets, i.e., an indexed set. A single set is specified with a list of data values that are included in this set. The formal syntax for the set data command is:

```
set <setname> := [<value>] ... ;
```

A set may be empty, and it may contain any combination of numeric and non-numeric string values. For example, the following are valid `set` commands:

```
# An empty set
set A := ;

# A set of numbers
set A := 1 2 3;

```

(continues on next page)

(continued from previous page)

```
# A set of strings
set B := north south east west;

# A set of mixed types
set C :=
0
-1.0e+10
'foo bar'
infinity
"100"
;
```

Sets of Tuple Data

The `set` data command can also specify tuple data with the standard notation for tuples. For example, suppose that set `A` contains 3-tuples:

```
model.A = pyo.Set(dimen=3)
```

The following `set` data command then specifies that `A` is the set containing the tuples `(1,2,3)` and `(4,5,6)`:

```
set A := (1,2,3) (4,5,6) ;
```

Alternatively, set data can simply be listed in the order that the tuple is represented:

```
set A := 1 2 3 4 5 6 ;
```

Obviously, the number of data elements specified using this syntax should be a multiple of the set dimension.

Sets with 2-tuple data can also be specified in a matrix denoting set membership. For example, the following `set` data command declares 2-tuples in `A` using plus (+) to denote valid tuples and minus (-) to denote invalid tuples:

```
set A : A1 A2 A3 A4 :=
    1   +  -  -  +
    2   +  -  +  -
    3   -  +  -  - ;
```

This data command declares the following five 2-tuples: `('A1',1)`, `('A1',2)`, `('A2',3)`, `('A3',2)`, and `('A4',1)`.

Finally, a set of tuple data can be concisely represented with tuple *templates* that represent a *slice* of tuple data. For example, suppose that the set `A` contains 4-tuples:

```
model.A = pyo.Set(dimen=4)
```

The following `set` data command declares groups of tuples that are defined by a template and data to complete this template:

```
set A :=
    (1,2,*,4) A B
    (*,2,*,4) A B C D ;
```

A tuple template consists of a tuple that contains one or more asterisk (*) symbols instead of a value. These represent indices where the tuple value is replaced by the values from the list of values that follows the tuple template. In this example, the following tuples are in set `A`:

```
(1, 2, 'A', 4)
(1, 2, 'B', 4)
('A', 2, 'B', 4)
('C', 2, 'D', 4)
```

Set Arrays

The set data command can also be used to declare data for a set array. Each set in a set array must be declared with a separate set data command with the following syntax:

```
set <set-name>[<index>] := [<value>] ... ;
```

Because set arrays can be indexed by an arbitrary set, the index value may be a numeric value, a non-numeric string value, or a comma-separated list of string values.

Suppose that a set A is used to index a set B as follows:

```
model.A = pyo.Set()
model.B = pyo.Set(model.A)
```

Then set B is indexed using the values declared for set A:

```
set A := 1 aaa 'a b';

set B[1] := 0 1 2;
set B[aaa] := aa bb cc;
set B['a b'] := 'aa bb cc';
```

The param Command

Simple or non-indexed parameters are declared in an obvious way, as shown by these examples:

```
param A := 1.4;
param B := 1;
param C := abc;
param D := true;
param E := 1.0e+04;
```

Parameters can be defined with numeric data, simple strings and quoted strings. Note that parameters cannot be defined without data, so there is no analog to the specification of an empty set.

One-dimensional Parameter Data

Most parameter data is indexed over one or more sets, and there are a number of ways the param data command can be used to specify indexed parameter data. One-dimensional parameter data is indexed over a single set. Suppose that the parameter B is a parameter indexed by the set A:

```
model.A = pyo.Set()
model.B = pyo.Param(model.A)
```

A param data command can specify values for B with a list of index-value pairs:

```
set A := a c e;

param B := a 10 c 30 e 50;
```

Because whitespace is ignored, this example data command file can be reorganized to specify the same data in a tabular format:

```
set A := a c e;

param B :=
a 10
c 30
e 50
;
```

Multiple parameters can be defined using a single `param` data command. For example, suppose that parameters `B`, `C`, and `D` are one-dimensional parameters all indexed by the set `A`:

```
model.A = pyo.Set()
model.B = pyo.Param(model.A)
model.C = pyo.Param(model.A)
model.D = pyo.Param(model.A)
```

Values for these parameters can be specified using a single `param` data command that declares these parameter names followed by a list of index and parameter values:

```
set A := a c e;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The values in the `param` data command are interpreted as a list of sublists, where each sublist consists of an index followed by the corresponding numeric value.

Note that parameter values do not need to be defined for all indices. For example, the following data command file is valid:

```
set A := a c e g;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The index `g` is omitted from the `param` command, and consequently this index is not valid for the model instance that uses this data. More complex patterns of missing data can be specified using the period (`.`) symbol to indicate a missing value. This syntax is useful when specifying multiple parameters that do not necessarily have the same index values:

```
set A := a c e;

```

(continues on next page)

(continued from previous page)

```
param : B C D :=
a   . -1 1.1
c 30   . 3.3
e 50 -5    .
;
```

This example provides a concise representation of parameters that share a common index set while using different index values.

Note that this data file specifies the data for set `A` twice: (1) when `A` is defined and (2) implicitly when the parameters are defined. An alternate syntax for `param` allows the user to concisely specify the definition of an index set along with associated parameters:

```
param : A : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

Finally, we note that default values for missing data can also be specified using the `default` keyword:

```
set A := a c e;

param B default 0.0 :=
c 30
e 50
;
```

Note that default values can only be specified in `param` commands that define values for a single parameter.

Multi-Dimensional Parameter Data

Multi-dimensional parameter data is indexed over either multiple sets or a single multi-dimensional set. Suppose that parameter `B` is a parameter indexed by set `A` that has dimension 2:

```
model.A = pyo.Set(dimen=2)
model.B = pyo.Param(model.A)
```

The syntax of the `param` data command remains essentially the same when specifying values for `B` with a list of index and parameter values:

```
set A := a 1 c 2 e 3;

param B :=
a 1 10
c 2 30
e 3 50;
```

Missing and default values are also handled in the same way with multi-dimensional index sets:

```
set A := a 1 c 2 e 3;

param B default 0 :=
```

(continues on next page)

(continued from previous page)

```
a 1 10
c 2 .
e 3 50;
```

Similarly, multiple parameters can defined with a single `param` data command. Suppose that parameters `B`, `C`, and `D` are parameters indexed over set `A` that has dimension 2:

```
model.A = pyo.Set(dimen=2)
model.B = pyo.Param(model.A)
model.C = pyo.Param(model.A)
model.D = pyo.Param(model.A)
```

These parameters can be defined with a single `param` command that declares the parameter names followed by a list of index and parameter values:

```
set A := a 1 c 2 e 3;

param : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```

Similarly, the following `param` data command defines the index set along with the parameters:

```
param : A : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```

The `param` command also supports a matrix syntax for specifying the values in a parameter that has a 2-dimensional index. Suppose parameter `B` is indexed over set `A` that has dimension 2:

```
model.A = pyo.Set(dimen=2)
model.B = pyo.Param(model.A)
```

The following `param` command defines a matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B : a c e :=
1 1 2 3
2 4 5 6
3 7 8 9
;
```

Additionally, the following syntax can be used to specify a transposed matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B (tr) : 1 2 3 :=
a 1 4 7
```

(continues on next page)

(continued from previous page)

```
c 2 5 8
e 3 6 9
;
```

This functionality facilitates the presentation of parameter data in a natural format. In particular, the transpose syntax may allow the specification of tables for which the rows comfortably fit within a single line. However, a matrix may be divided column-wise into shorter rows since the line breaks are not significant in Pyomo data commands.

For parameters with three or more indices, the parameter data values may be specified as a series of slices. Each slice is defined by a template followed by a list of index and parameter values. Suppose that parameter `B` is indexed over set `A` that has dimension 4:

```
model.A = pyo.Set(dimen=4)
model.B = pyo.Param(model.A)
```

The following `param` command defines a matrix of parameter values with multiple templates:

```
set A := (a,1,a,1) (a,2,a,2) (b,1,b,1) (b,2,b,2);

param B :=

  [*,1,*,1] a a 10 b b 20
  [*,2,*,2] a a 30 b b 40
;
```

The `B` parameter consists of four values: `B[a,1,a,1]=10`, `B[b,1,b,1]=20`, `B[a,2,a,2]=30`, and `B[b,2,b,2]=40`.

The `table` Command

The `table` data command explicitly specifies a two-dimensional array of parameter data. This command provides a more flexible and complete data declaration than is possible with a `param` declaration. The following example illustrates a simple `table` command that declares data for a single parameter:

```
table M(A) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

The parameter `M` is indexed by column `A`, which must be pre-defined unless declared separately (see below). The column labels are provided after the colon and before the colon-equal (`:=`). Subsequently, the table data is provided. The syntax is not sensitive to whitespace, so the following is an equivalent `table` command:

```
table M(A) :
A  B  M   N :=
A1 B1 4.3 5.3 A2 B2 4.4 5.4 A3 B3 4.5 5.5 ;
```

Multiple parameters can be declared by simply including additional parameter names. For example:

```
table M(A) N(A,B) :
A  B  M   N :=
A1 B1 4.3 5.3
```

(continues on next page)

(continued from previous page)

```
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

This example declares data for the M and N parameters, which have different indexing columns. The indexing columns represent set data, which is specified separately. For example:

```
table A={A} Z={A,B} M(A) N(A,B) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

This example declares data for the M and N parameters, along with the A and Z indexing sets. The correspondence between the index set Z and the indices of parameter N can be made more explicit by indexing N by Z:

```
table A={A} Z={A,B} M(A) N(Z) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Set data can also be specified independent of parameter data:

```
table Z={A,B} Y={M,N} :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Warning

If a `table` command does not explicitly indicate the indexing sets, then these are assumed to be initialized separately. A `table` command can separately initialize sets and parameters in a Pyomo model, and there is no presumed association between the data that is initialized. For example, the `table` command initializes a set Z and a parameter M that are not related:

```
table Z={A,B} M(A):
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Finally, simple parameter values can also be specified with a `table` command:

```
table pi := 3.1416 ;
```

The previous examples considered examples of the `table` command where column labels are provided. The `table`

command can also be used without column labels. For example, the first example can be revised to omit column labels as follows:

```
table columns=4 M(1)={3} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

The `columns=4` is a keyword-value pair that defines the number of columns in this table; this must be explicitly specified in tables without column labels. The default column labels are integers starting from `1`; the labels are columns `1`, `2`, `3`, and `4` in this example. The `M` parameter is indexed by column `1`. The braces syntax declares the column where the `M` data is provided.

Similarly, set data can be declared referencing the integer column labels:

```
table columns=4 A={1} Z={1,2} M(1)={3} N(1,2)={4} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Declared set names can also be used to index parameters:

```
table columns=4 A={1} Z={1,2} M(A)={3} N(Z)={4} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Finally, we compare and contrast the `table` and `param` commands. Both commands can be used to declare parameter and set data, and both commands can be used to declare a simple parameter. However, there are some important differences between these data commands:

- The `param` command can declare a single set that is used to index one or more parameters. The `table` command can declare data for any number of sets, independent of whether they are used to index parameter data.
- The `param` command can declare data for multiple parameters only if they share the same index set. The `table` command can declare data for any number of parameters that are may be indexed separately.
- The `table` syntax unambiguously describes the dimensionality of indexing sets. The `param` command must be interpreted with a model that provides the dimension of the indexing set.

This last point provides a key motivation for the `table` command. Specifically, the `table` command can be used to reliably initialize concrete models using Pyomo's `DataPortal` class. By contrast, the `param` command can only be used to initialize concrete models with parameters that are indexed by a single column (i.e., a simple set).

The `load` Command

The `load` command provides a mechanism for loading data from a variety of external tabular data sources. This command loads a table of data that represents set and parameter data in a Pyomo model. The table consists of rows and columns for which all rows have the same length, all columns have the same length, and the first row represents labels for the column data.

The `load` command can load data from a variety of different external data sources:

- **TAB File**: A text file format that uses whitespace to separate columns of values in each row of a table.

- **CSV File**: A text file format that uses comma or other delimiters to separate columns of values in each row of a table.
- **XML File**: An extensible markup language for documents and data structures. XML files can represent tabular data.
- **Excel File**: A spreadsheet data format that is primarily used by the Microsoft Excel application.
- **Database**: A relational database.

This command uses a *data manager* that coordinates how data is extracted from a specified *data source*. In this way, the `load` command provides a generic mechanism that enables Pyomo models to interact with standard data repositories that are maintained in an application-specific manner.

Simple Load Examples

The simplest illustration of the `load` command is specifying data for an indexed parameter. Consider the file `Y.tab`:

```
A   Y
A1 3.3
A2 3.4
A3 3.5
```

This file specifies the values of parameter `Y` which is indexed by set `A`. The following `load` command loads the parameter data:

```
load Y.tab : [A] Y;
```

The first argument is the filename. The options after the colon indicate how the table data is mapped to model data. Option `[A]` indicates that set `A` is used as the index, and option `Y` indicates the parameter that is initialized.

Similarly, the following load command loads both the parameter data as well as the index set `A`:

```
load Y.tab : A=[A] Y;
```

The difference is the specification of the index set, `A=[A]`, which indicates that set `A` is initialized with the index loaded from the ASCII table file.

Set data can also be loaded from a ASCII table file that contains a single column of data:

```
A
A1
A2
A3
```

The `format` option must be specified to denote the fact that the relational data is being interpreted as a set:

```
load A.tab format=set : A;
```

Note that this allows for specifying set data that contains tuples. Consider file `C.tab`:

```
A   B
A1 1
A1 2
A1 3
A2 1
A2 2
```

(continues on next page)

(continued from previous page)

```
A2 3
A3 1
A3 2
A3 3
```

A similar load syntax will load this data into set C:

```
load C.tab format=set : C;
```

Note that this example requires that C be declared with dimension two.

Load Syntax Options

The syntax of the load command is broken into two parts. The first part ends with the colon, and it begins with a filename, database URL, or DSN (data source name). Additionally, this first part can contain option value pairs. The following options are recognized:

format	A string that denotes how the relational table is interpreted
password	The password that is used to access a database
query	The query that is used to request data from a database
range	The subset of a spreadsheet that is requestedindex{spreadsheet}
user	The user name that is used to access the data source
using	The data manager that is used to process the data source
table	The database table that is requested

The format option is the only option that is required for all data managers. This option specifies how a relational table is interpreted to represent set and parameter data. If the using option is omitted, then the filename suffix is used to select the data manager. The remaining options are specific to spreadsheets and relational databases (see below).

The second part of the load command consists of the specification of column names for indices and data. The remainder of this section describes different specifications and how they define how data is loaded into a model. Suppose file ABCD.tab defines the following relational table:

```
A  B  C D
A1 B1 1 10
A2 B2 2 20
A3 B3 3 30
```

There are many ways to interpret this relational table. It could specify a set of 4-tuples, a parameter indexed by 3-tuples, two parameters indexed by 2-tuples, and so on. Additionally, we may wish to select a subset of this table to initialize data in a model. Consequently, the load command provides a variety of syntax options for specifying how a table is interpreted.

A simple specification is to interpret the relational table as a set:

```
load ABCD.tab format=set : Z ;
```

Note that Z is a set in the model that the data is being loaded into. If this set does not exist, an error will occur while loading data from this table.

Another simple specification is to interpret the relational table as a parameter with indexed by 3-tuples:

```
load ABCD.tab : [A,B,C] D ;
```

Again, this requires that D be a parameter in the model that the data is being loaded into. Additionally, the index set for D must contain the indices that are specified in the table. The load command also allows for the specification of the index set:

```
load ABCD.tab : Z=[A,B,C] D ;
```

This specifies that the index set is loaded into the Z set in the model. Similarly, data can be loaded into another parameter than what is specified in the relational table:

```
load ABCD.tab : Z=[A,B,C] Y=D ;
```

This specifies that the index set is loaded into the Z set and that the data in the D column in the table is loaded into the Y parameter.

This syntax allows the load command to provide an arbitrary specification of data mappings from columns in a relational table into index sets and parameters. For example, suppose that a model is defined with set Z and parameters Y and W:

```
model.Z = pyo.Set()
model.Y = pyo.Param(model.Z)
model.W = pyo.Param(model.Z)
```

Then the following command defines how these data items are loaded using columns B, C and D:

```
load ABCD.tab : Z=[B] Y=D W=C;
```

When the using option is omitted the data manager is inferred from the filename suffix. However, the filename suffix does not always reflect the format of the data it contains. For example, consider the relational table in the file ABCD.txt:

```
A,B,C,D
A1,B1,1,10
A2,B2,2,20
A3,B3,3,30
```

We can specify the using option to load from this file into parameter D and set Z:

```
load ABCD.txt using=csv : Z=[A,B,C] D ;
```

Note

The data managers supported by Pyomo can be listed with the pyomo help subcommand

```
pyomo help --data-managers
```

The following data managers are supported in Pyomo 5.1:

```
Pyomo Data Managers
-------------------
  csv
      CSV file interface
  dat
      Pyomo data command file interface
  json
      JSON file interface
  pymysql
      pymysql database interface
```

```
pyodbc
    pyodbc database interface
pypyodbc
    pypyodbc database interface
sqlite3
    sqlite3 database interface
tab
    TAB file interface
xls
    Excel XLS file interface
xlsb
    Excel XLSB file interface
xlsm
    Excel XLSM file interface
xlsx
    Excel XLSX file interface
xml
    XML file interface
yaml
    YAML file interface
```

Interpreting Tabular Data

By default, a table is interpreted as columns of one or more parameters with associated index columns. The `format` option can be used to specify other interpretations of a table:

`array`	The table is a matrix representation of a two dimensional parameter.
`param`	The data is a simple parameter value.
`set`	Each row is a set element.
`set_array`	The table is a matrix representation of a set of 2-tuples.
`transposed_array`	The table is a transposed matrix representation of a two dimensional parameter.

We have previously illustrated the use of the `set` format value to interpret a relational table as a set of values or tuples. The following examples illustrate the other format values.

A table with a single value can be interpreted as a simple parameter using the `param` format value. Suppose that `Z.tab` contains the following table:

```
1.1
```

The following load command then loads this value into parameter `p`:

```
load Z.tab format=param: p;
```

Sets with 2-tuple data can be represented with a matrix format that denotes set membership. The `set_array` format value interprets a relational table as a matrix that defines a set of 2-tuples where `+` denotes a valid tuple and `-` denotes an invalid tuple. Suppose that `D.tab` contains the following relational table:

```
B   A1  A2  A3
1   +   -   -
2   -   +   -
3   -   -   +
```

Then the following load command loads data into set B:

```
load D.tab format=set_array: B;
```

This command declares the following 2-tuples: ('A1',1), ('A2',2), and ('A3',3).

Parameters with 2-tuple indices can be interpreted with a matrix format that where rows and columns are different indices. Suppose that U.tab contains the following table:

```
I  A1  A2  A3
I1 1.3 2.3 3.3
I2 1.4 2.4 3.4
I3 1.5 2.5 3.5
I4 1.6 2.6 3.6
```

Then the following load command loads this value into parameter U with a 2-dimensional index using the array format value.:

```
load U.tab format=array: A=[X] U;
```

The transpose_array format value also interprets the table as a matrix, but it loads the data in a transposed format:

```
load U.tab format=transposed_array: A=[X] U;
```

Note that these format values do not support the initialization of the index data.

Loading from Spreadsheets and Relational Databases

Many of the options for the load command are specific to spreadsheets and relational databases. The range option is used to specify the range of cells that are loaded from a spreadsheet. The range of cells represents a table in which the first row of cells defines the column names for the table.

Suppose that file ABCD.xls contains the range ABCD that is shown in the following figure:

ABCD | fx A

	A	B	C	D	E
1	A	B	C	D	
2	A1	B1	1	10	
3	A2	B2	2	20	
4	A3	B3	3	30	
5					

The following command loads this data to initialize parameter D and index Z:

```
load ABCD.xls range=ABCD : Z=[A,B,C] Y=D ;
```

Thus, the syntax for loading data from spreadsheets only differs from CSV and ASCII text files by the use of the range option.

When loading from a relational database, the data source specification is a filename or data connection string. Access to a database may be restricted, and thus the specification of username and password options may be required. Alternatively, these options can be specified within a data connection string.

A variety of database interface packages are available within Python. The `using` option is used to specify the database interface package that will be used to access a database. For example, the `pyodbc` interface can be used to connect to Excel spreadsheets. The following command loads data from the Excel spreadsheet `ABCD.xls` using the `pyodbc` interface. The command loads this data to initialize parameter `D` and index `Z`:

```
load ABCD.xls using=pyodbc table=ABCD : Z=[A,B,C] Y=D ;
```

The `using` option specifies that the `pyodbc` package will be used to connect with the Excel spreadsheet. The `table` option specifies that the table `ABCD` is loaded from this spreadsheet. Similarly, the following command specifies a data connection string to specify the ODBC driver explicitly:

```
load "Driver={Microsoft Excel Driver (*.xls)}; Dbq=ABCD.xls;"
    using=pyodbc
    table=ABCD : Z=[A,B,C] Y=D ;
```

ODBC drivers are generally tailored to the type of data source that they work with; this syntax illustrates how the `load` command can be tailored to the details of the database that a user is working with.

The previous examples specified the `table` option, which declares the name of a relational table in a database. Many databases support the Structured Query Language (SQL), which can be used to dynamically compose a relational table from other tables in a database. The classic diet problem will be used to illustrate the use of SQL queries to initialize a Pyomo model. In this problem, a customer is faced with the task of minimizing the cost for a meal at a fast food restaurant – they must purchase a sandwich, side, and a drink for the lowest cost. The following is a Pyomo model for this problem:

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# diet1.py
import pyomo.environ as pyo

infinity = float('inf')
MAX_FOOD_SUPPLY = 20.0  # There is a finite food supply

model = pyo.AbstractModel()

# --------------------------------------------------------

model.FOOD = pyo.Set()
model.cost = pyo.Param(model.FOOD, within=pyo.PositiveReals)
model.f_min = pyo.Param(model.FOOD, within=pyo.NonNegativeReals, default=0.0)


def f_max_validate(model, value, j):
    return model.f_max[j] > model.f_min[j]


model.f_max = pyo.Param(model.FOOD, validate=f_max_validate, default=MAX_FOOD_SUPPLY)
```

(continues on next page)

(continued from previous page)

```
model.NUTR = pyo.Set()
model.n_min = pyo.Param(model.NUTR, within=pyo.NonNegativeReals, default=0.0)
model.n_max = pyo.Param(model.NUTR, default=infinity)
model.amt = pyo.Param(model.NUTR, model.FOOD, within=pyo.NonNegativeReals)

# -------------------------------------------------------


def Buy_bounds(model, i):
    return (model.f_min[i], model.f_max[i])


model.Buy = pyo.Var(model.FOOD, bounds=Buy_bounds, within=pyo.NonNegativeIntegers)

# -------------------------------------------------------


def Total_Cost_rule(model):
    return sum(model.cost[j] * model.Buy[j] for j in model.FOOD)


model.Total_Cost = pyo.Objective(rule=Total_Cost_rule, sense=pyo.minimize)

# -------------------------------------------------------


def Entree_rule(model):
    entrees = [
        'Cheeseburger',
        'Ham Sandwich',
        'Hamburger',
        'Fish Sandwich',
        'Chicken Sandwich',
    ]
    return sum(model.Buy[e] for e in entrees) >= 1


model.Entree = pyo.Constraint(rule=Entree_rule)


def Side_rule(model):
    sides = ['Fries', 'Sausage Biscuit']
    return sum(model.Buy[s] for s in sides) >= 1


model.Side = pyo.Constraint(rule=Side_rule)


def Drink_rule(model):
    drinks = ['Lowfat Milk', 'Orange Juice']
    return sum(model.Buy[d] for d in drinks) >= 1
```

(continues on next page)

(continued from previous page)

```
model.Drink = pyo.Constraint(rule=Drink_rule)
```

Suppose that the file `diet1.sqlite` be a SQLite database file that contains the following data in the `Food` table:

FOOD	cost
Cheeseburger	1.84
Ham Sandwich	2.19
Hamburger	1.84
Fish Sandwich	1.44
Chicken Sandwich	2.29
Fries	0.77
Sausage Biscuit	1.29
Lowfat Milk	0.60
Orange Juice	0.72

In addition, the `Food` table has two additional columns, `f_min` and `f_max`, with no data for any row. These columns exist to match the structure for the parameters used in the model.

We can solve the `diet1` model using the Python definition in `diet1.py` and the data from this database. The file `diet.sqlite.dat` specifies a `load` command that uses that `sqlite3` data manager and embeds a SQL query to retrieve the data:

```
# File diet.sqlite.dat

load "diet.sqlite"
     using=sqlite3
     query="SELECT FOOD,cost,f_min,f_max FROM Food"
     : FOOD=[FOOD] cost f_min f_max ;
```

The PyODBC driver module will pass the SQL query through an Access ODBC connector, extract the data from the `diet1.mdb` file, and return it to Pyomo. The Pyomo ODBC handler can then convert the data received into the proper format for solving the model internally. More complex SQL queries are possible, depending on the underlying database and ODBC driver in use. However, the name and ordering of the columns queried are specified in the Pyomo data file; using SQL wildcards (e.g., `SELECT *`) or column aliasing (e.g., `SELECT f AS FOOD`) may cause errors in Pyomo's mapping of relational data to parameters.

The `include` Command

The `include` command allows a data command file to execute data commands from another file. For example, the following command file executes data commands from `ex1.dat` and then `ex2.dat`:

```
include ex1.dat;
include ex2.dat;
```

Pyomo is sensitive to the order of execution of data commands, since data commands can redefine set and parameter values. The `include` command respects this data ordering; all data commands in the included file are executed before the remaining data commands in the current file are executed.

The namespace Keyword

The namespace keyword is not a data command, but instead it is used to structure the specification of Pyomo's data commands. Specifically, a namespace declaration is used to group data commands and to provide a group label. Consider the following data command file:

```
set C := 1 2 3 ;

namespace ns1
{
    set C := 4 5 6 ;
}

namespace ns2
{
    set C := 7 8 9 ;
}
```

This data file defines two namespaces: ns1 and ns2 that initialize a set C. By default, data commands contained within a namespace are ignored during model construction; when no namespaces are specified, the set C has values 1,2,3. When namespace ns1 is specified, then the set C values are overridden with the set 4,5,6.

Data Portals

Pyomo's DataPortal class standardizes the process of constructing model instances by managing the process of loading data from different data sources in a uniform manner. A DataPortal object can load data from the following data sources:

- **TAB File**: A text file format that uses whitespace to separate columns of values in each row of a table.
- **CSV File**: A text file format that uses comma or other delimiters to separate columns of values in each row of a table.
- **JSON File**: A popular lightweight data-interchange format that is easily parsed.
- **YAML File**: A human friendly data serialization standard.
- **XML File**: An extensible markup language for documents and data structures. XML files can represent tabular data.
- **Excel File**: A spreadsheet data format that is primarily used by the Microsoft Excel application.
- **Database**: A relational database.
- **DAT File**: A Pyomo data command file.

Note that most of these data formats can express tabular data.

Warning

The DataPortal class requires the installation of Python packages to support some of these data formats:

- **YAML File**: pyyaml
- **Excel File**: win32com, openpyxl or xlrd

  These packages support different data Excel data formats: the win32com package supports .xls, .xlsm and .xlsx, the openpyxl package supports .xlsx and the xlrd package supports .xls.

- **Database**: pyodbc, pypyodbc, sqlite3 or pymysql

  These packages support different database interface APIs: the pyodbc and pypyodbc packages support the ODBC database API, the sqlite3 package uses the SQLite C library to directly interface with databases using the DB-API 2.0 specification, and pymysql is a pure-Python MySQL client.

DataPortal objects can be used to initialize both concrete and abstract Pyomo models. Consider the file A.tab, which defines a simple set with a tabular format:

```
A
A1
A2
A3
```

The load method is used to load data into a DataPortal object. Components in a concrete model can be explicitly initialized with data loaded by a DataPortal object:

```
data = pyo.DataPortal()
data.load(filename='A.tab', set="A", format="set")

model = pyo.ConcreteModel()
model.A = pyo.Set(initialize=data['A'])
```

All data needed to initialize an abstract model *must* be provided by a DataPortal object, and the use of the DataPortal object to initialize components is automated for the user:

```
model = pyo.AbstractModel()
model.A = pyo.Set()
data = pyo.DataPortal()
data.load(filename='A.tab', set=model.A)
instance = model.create_instance(data)
```

Note the difference in the execution of the load method in these two examples: for concrete models data is loaded by name and the format must be specified, and for abstract models the data is loaded by component, from which the data format can often be inferred.

The load method opens the data file, processes it, and loads the data in a format that can be used to construct a model instance. The load method can be called multiple times to load data for different sets or parameters, or to override data processed earlier. The load method takes a variety of arguments that define how data is loaded:

- filename: This option specifies the source data file.
- format: This option specifies the how to interpret data within a table. Valid formats are: set, set_array, param, table, array, and transposed_array.
- set: This option is either a string or model compent that defines a set that will be initialized with this data.
- param: This option is either a string or model compent that defines a parameter that will be initialized with this data. A list or tuple of strings or model components can be used to define multiple parameters that are initialized.
- index: This option is either a string or model compent that defines an index set that will be initialized with this data.
- using: This option specifies the Python package used to load this data source. This option is used when loading data from databases.

- `select`: This option defines the columns that are selected from the data source. The column order may be changed from the data source, which allows the `DataPortal` object to define
- `namespace`: This option defines the data namespace that will contain this data.

The use of these options is illustrated below.

The `DataPortal` class also provides a simple API for accessing set and parameter data that are loaded from different data sources. The `[]` operator is used to access set and parameter values. Consider the following example, which loads data and prints the value of the `[]` operator:

```
data = pyo.DataPortal()
data.load(filename='A.tab', set="A", format="set")
print(data['A'])  # ['A1', 'A2', 'A3']

data.load(filename='Z.tab', param="z", format="param")
print(data['z'])  # 1.1

data.load(filename='Y.tab', param="y", format="table")
for key in sorted(data['y']):
    print("%s %s" % (key, data['y'][key]))
```

The `DataPortal` class also has several methods for iterating over the data that has been loaded:

- `keys()`: Returns an iterator of the data keys.
- `values()`: Returns an iterator of the data values.
- `items()`: Returns an iterator of (name, value) tuples from the data.

Finally, the `data()` method provides a generic mechanism for accessing the underlying data representation used by `DataPortal` objects.

Loading Structured Data

JSON and YAML files are structured data formats that are well-suited for data serialization. These data formats do not represent data in tabular format, but instead they directly represent set and parameter values with lists and dictionaries:

- **Simple Set**: a list of string or numeric value
- **Indexed Set**: a dictionary that maps an index to a list of string or numeric value
- **Simple Parameter**: a string or numeric value
- **Indexed Parameter**: a dictionary that maps an index to a numeric value

For example, consider the following JSON file:

```
{   "A": ["A1", "A2", "A3"],
    "B": [[1, "B1"], [2, "B2"], [3, "B3"]],
    "C": {"A1": [1, 2, 3], "A3": [10, 20, 30]},
    "p": 0.1,
    "q": {"A1": 3.3, "A2": 3.4, "A3": 3.5},
    "r": [   {"index": [1, "B1"], "value": 3.3},
             {"index": [2, "B2"], "value": 3.4},
             {"index": [3, "B3"], "value": 3.5}]}
```

The data in this file can be used to load the following model:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.B = pyo.Set(dimen=2)
model.C = pyo.Set(model.A)
model.p = pyo.Param()
model.q = pyo.Param(model.A)
model.r = pyo.Param(model.B)
data.load(filename='T.json')
```

Note that no `set` or `param` option needs to be specified when loading a `JSON` or `YAML` file. All of the set and parameter data in the file are loaded by the `DataPortal>` object, and only the data needed for model construction is used.

The following YAML file has a similar structure:

```
A: [A1, A2, A3]
B:
- [1, B1]
- [2, B2]
- [3, B3]
C:
  'A1': [1, 2, 3]
  'A3': [10, 20, 30]
p: 0.1
q: {A1: 3.3, A2: 3.4, A3: 3.5}
r:
- index: [1, B1]
  value: 3.3
- index: [2, B2]
  value: 3.4
- index: [3, B3]
  value: 3.5
```

The data in this file can be used to load a Pyomo model with the same syntax as a JSON file:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.B = pyo.Set(dimen=2)
model.C = pyo.Set(model.A)
model.p = pyo.Param()
model.q = pyo.Param(model.A)
model.r = pyo.Param(model.B)
data.load(filename='T.yaml')
```

Loading Tabular Data

Many data sources supported by Pyomo are tabular data formats. Tabular data is numerical or textual data that is organized into one or more simple tables, where data is arranged in a matrix. Each table consists of a matrix of numeric string values, simple strings, and quoted strings. All rows have the same length, all columns have the same length, and the first row typically represents labels for the column data.

The following section describes the tabular data sources supported by Pyomo, and the subsequent sections illustrate ways that data can be loaded from tabular data using TAB files. Subsequent sections describe options for loading data

from Excel spreadsheets and relational databases.

Tabular Data

TAB files represent tabular data in an ascii file using whitespace as a delimiter. A TAB file consists of rows of values, where each row has the same length. For example, the file `PP.tab` has the format:

```
A   B   PP
A1  B1  4.3
A2  B2  4.4
A3  B3  4.5
```

CSV files represent tabular data in a format that is very similar to TAB files. Pyomo assumes that a CSV file consists of rows of values, where each row has the same length. For example, the file `PP.csv` has the format:

```
A,B,PP
A1,B1,4.3
A2,B2,4.4
A3,B3,4.5
```

Excel spreadsheets can express complex data relationships. A *range* is a contiguous, rectangular block of cells in an Excel spreadsheet. Thus, a range in a spreadsheet has the same tabular structure as is a TAB file or a CSV file. For example, consider the file `excel.xls` that has the range `PPtable`:

PPtable

A	B	PP
A1	B1	4.3
A2	B2	4.4
A3	B3	4.5

A relational database is an application that organizes data into one or more tables (or *relations*) with a unique key in each row. Tables both reflect the data in a database as well as the result of queries within a database.

XML files represent tabular using `table` and `row` elements. Each sub-element of a `row` element represents a different column, where each row has the same length. For example, the file `PP.xml` has the format:

```
<table>
  <row>
    <A value="A1"/><B value="B1"/><PP value="4.3"/>
  </row>
  <row>
    <A value="A2"/><B value="B2"/><PP value="4.4"/>
  </row>
  <row>
    <A value="A3"/><B value="B3"/><PP value="4.5"/>
  </row>
</table>
```

Loading Set Data

The `set` option is used specify a `Set` component that is loaded with data.

Loading a Simple Set

Consider the file `A.tab`, which defines a simple set:

```
A
A1
A2
A3
```

In the following example, a `DataPortal` object loads data for a simple set `A`:

```
model = pyo.AbstractModel()
model.A = pyo.Set()
data = pyo.DataPortal()
data.load(filename='A.tab', set=model.A)
instance = model.create_instance(data)
```

Loading a Set of Tuples

Consider the file `C.tab`:

```
A  B
A1 1
A1 2
A1 3
A2 1
A2 2
A2 3
A3 1
A3 2
A3 3
```

In the following example, a `DataPortal` object loads data for a two-dimensional set `C`:

```
model = pyo.AbstractModel()
model.C = pyo.Set(dimen=2)
data = pyo.DataPortal()
data.load(filename='C.tab', set=model.C)
instance = model.create_instance(data)
```

In this example, the column titles do not directly impact the process of loading data. Column titles can be used to select a subset of columns from a table that is loaded (see below).

Loading a Set Array

Consider the file `D.tab`, which defines an array representation of a two-dimensional set:

```
B  A1  A2  A3
1  +   -   -
2  -   +   -
3  -   -   +
```

In the following example, a `DataPortal` object loads data for a two-dimensional set `D`:

```
model = pyo.AbstractModel()
model.D = pyo.Set(dimen=2)
data = pyo.DataPortal()
data.load(filename='D.tab', set=model.D, format='set_array')
instance = model.create_instance(data)
```

The `format` option indicates that the set data is declared in a array format.

Loading Parameter Data

The `param` option is used specify a `Param` component that is loaded with data.

Loading a Simple Parameter

The simplest parameter is simply a singleton value. Consider the file `Z.tab`:

```
1.1
```

In the following example, a `DataPortal` object loads data for a simple parameter `z`:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.z = pyo.Param()
data.load(filename='Z.tab', param=model.z)
instance = model.create_instance(data)
```

Loading an Indexed Parameter

An indexed parameter can be defined by a single column in a table. For example, consider the file `Y.tab`:

```
A   Y
A1 3.3
A2 3.4
A3 3.5
```

In the following example, a `DataPortal` object loads data for an indexed parameter `y`:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(initialize=['A1', 'A2', 'A3'])
model.y = pyo.Param(model.A)
data.load(filename='Y.tab', param=model.y)
instance = model.create_instance(data)
```

When column names are not used to specify the index and parameter data, then the `DataPortal` object assumes that the rightmost column defines parameter values. In this file, the `A` column contains the index values, and the `Y` column contains the parameter values.

Loading Set and Parameter Values

Note that the data for set `A` is predefined in the previous example. The index set can be loaded with the parameter data using the `index` option. In the following example, a `DataPortal` object loads data for set `A` and the indexed parameter `y`

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.y = pyo.Param(model.A)
data.load(filename='Y.tab', param=model.y, index=model.A)
instance = model.create_instance(data)
```

An index set with multiple dimensions can also be loaded with an indexed parameter. Consider the file `PP.tab`:

```
A  B  PP
A1 B1 4.3
A2 B2 4.4
A3 B3 4.5
```

In the following example, a `DataPortal` object loads data for a tuple set and an indexed parameter:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(dimen=2)
model.p = pyo.Param(model.A)
data.load(filename='PP.tab', param=model.p, index=model.A)
instance = model.create_instance(data)
```

Loading a Parameter with Missing Values

Missing parameter data can be expressed in two ways. First, parameter data can be defined with indices that are a subset of valid indices in the model. The following example loads the indexed parameter `y`:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(initialize=['A1', 'A2', 'A3', 'A4'])
model.y = pyo.Param(model.A)
data.load(filename='Y.tab', param=model.y)
instance = model.create_instance(data)
```

The model defines an index set with four values, but only three parameter values are declared in the data file `Y.tab`.

Parameter data can also be declared with missing values using the period (`.`) symbol. For example, consider the file `S.tab`:

```
A  B  PP
A1 B1 4.3
A2 B2 4.4
A3 B3 4.5
```

In the following example, a `DataPortal` object loads data for the index set `A` and indexed parameter `y`:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.s = pyo.Param(model.A)
data.load(filename='S.tab', param=model.s, index=model.A)
instance = model.create_instance(data)
```

The period (.) symbol indicates a missing parameter value, but the index set A contains the index value for the missing parameter.

Loading Multiple Parameters

Multiple parameters can be initialized at once by specifying a list (or tuple) of component parameters. Consider the file XW.tab:

```
A   X    W
A1 3.3 4.3
A2 3.4 4.4
A3 3.5 4.5
```

In the following example, a DataPortal object loads data for parameters x and w:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(initialize=['A1', 'A2', 'A3'])
model.x = pyo.Param(model.A)
model.w = pyo.Param(model.A)
data.load(filename='XW.tab', param=(model.x, model.w))
instance = model.create_instance(data)
```

Selecting Parameter Columns

We have previously noted that the column names do not need to be specified to load set and parameter data. However, the select option can be to identify the columns in the table that are used to load parameter data. This option specifies a list (or tuple) of column names that are used, in that order, to form the table that defines the component data.

For example, consider the following load declaration:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.w = pyo.Param(model.A)
data.load(filename='XW.tab', select=('A', 'W'), param=model.w, index=model.A)
instance = model.create_instance(data)
```

The columns A and W are selected from the file XW.tab, and a single parameter is defined.

Loading a Parameter Array

Consider the file U.tab, which defines an array representation of a multiply-indexed parameter:

```
I   A1   A2   A3
I1 1.3 2.3 3.3
I2 1.4 2.4 3.4
I3 1.5 2.5 3.5
I4 1.6 2.6 3.6
```

In the following example, a DataPortal object loads data for a two-dimensional parameter u:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
```

(continues on next page)

(continued from previous page)

```
model.A = pyo.Set(initialize=['A1', 'A2', 'A3'])
model.I = pyo.Set(initialize=['I1', 'I2', 'I3', 'I4'])
model.u = pyo.Param(model.I, model.A)
data.load(filename='U.tab', param=model.u, format='array')
instance = model.create_instance(data)
```

The `format` option indicates that the parameter data is declared in a array format. The `format` option can also indicate that the parameter data should be transposed.

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(initialize=['A1', 'A2', 'A3'])
model.I = pyo.Set(initialize=['I1', 'I2', 'I3', 'I4'])
model.t = pyo.Param(model.A, model.I)
data.load(filename='U.tab', param=model.t, format='transposed_array')
instance = model.create_instance(data)
```

Note that the transposed parameter data changes the index set for the parameter.

Loading from Spreadsheets and Databases

Tabular data can be loaded from spreadsheets and databases using auxiliary Python packages that provide an interface to these data formats. Data can be loaded from Excel spreadsheets using the `win32com`, `xlrd` and `openpyxl` packages. For example, consider the following range of cells, which is named `PPtable`:

PPtable		
A	B	PP
A1	B1	4.3
A2	B2	4.4
A3	B3	4.5

In the following example, a `DataPortal` object loads the named range `PPtable` from the file `excel.xls`:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(dimen=2)
model.p = pyo.Param(model.A)
data.load(filename='excel.xls', range='PPtable', param=model.p, index=model.A)
instance = model.create_instance(data)
```

Note that the `range` option is required to specify the table of cell data that is loaded from the spreadsheet.

There are a variety of ways that data can be loaded from a relational database. In the simplest case, a table can be specified within a database:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set(dimen=2)
model.p = pyo.Param(model.A)
data.load(
    filename='PP.sqlite', using='sqlite3', table='PPtable', param=model.p, index=model.A
)
instance = model.create_instance(data)
```

In this example, the interface sqlite3 is used to load data from an SQLite database in the file PP.sqlite. More generally, an SQL query can be specified to dynamically generate a table. For example:

```
model = pyo.AbstractModel()
data = pyo.DataPortal()
model.A = pyo.Set()
model.p = pyo.Param(model.A)
data.load(
    filename='PP.sqlite',
    using='sqlite3',
    query="SELECT A,PP FROM PPtable",
    param=model.p,
    index=model.A,
)
instance = model.create_instance(data)
```

Data Namespaces

The DataPortal class supports the concept of a *namespace* to organize data into named groups that can be enabled or disabled during model construction. Various DataPortal methods have an optional namespace argument that defaults to None:

- data(name=None, namespace=None): Returns the data associated with data in the specified namespace
- []: For a DataPortal object data, the function data['A'] returns data corresponding to A in the default namespace, and data['ns1','A'] returns data corresponding to A in namespace ns1.
- namespaces(): Returns an iteratore for the data namespaces.
- keys(namespace=None): Returns an iterator of the data keys in the specified namespace.
- values(namespace=None): Returns and iterator of the data values in the specified namespace.
- items(namespace=None): Returns an iterator of (name, value) tuples in the specified namespace.

By default, data within a namespace are ignored during model construction. However, concrete models can be initialized with data from a specific namespace. Further, abstract models can be initialized with a list of namespaces that define the data used to initialized model components. For example, the following script generates two model instances from an abstract model using data loaded into different namespaces:

```
model = pyo.AbstractModel()
model.C = pyo.Set(dimen=2)
data = pyo.DataPortal()
data.load(filename='C.tab', set=model.C, namespace='ns1')
data.load(filename='D.tab', set=model.C, namespace='ns2', format='set_array')
instance1 = model.create_instance(data, namespaces=['ns1'])
instance2 = model.create_instance(data, namespaces=['ns2'])
```

Storing Data from Pyomo Models

Currently, Pyomo has rather limited capabilities for storing model data into standard Python data types and serialized data formats. However, this capability is under active development.

Storing Model Data in Excel

TODO

More here.

2.4.3 The pyomo Command

The `pyomo` command is issued to the DOS prompt or a Unix shell. To see a list of Pyomo command line options, use:

```
pyomo solve --help
```

Note

There are two dashes before `help`.

In this section we will detail some of the options.

Passing Options to a Solver

To pass arguments to a solver when using the `pyomo solve` command, append the Pyomo command line with the argument `--solver-options=` followed by an argument that is a string to be sent to the solver (perhaps with dashes added by Pyomo). So for most MIP solvers, the mip gap can be set using

```
--solver-options="mipgap=0.01"
```

Multiple options are separated by a space. Options that do not take an argument should be specified with the equals sign followed by either a space or the end of the string.

For example, to specify that the solver is GLPK, then to specify a mipgap of two percent and the GLPK cuts option, use

```
--solver=glpk --solver-options="mipgap=0.02 cuts="
```

If there are multiple "levels" to the keyword, as is the case for some Gurobi and CPLEX options, the tokens are separated by underscore. For example, `mip cuts all` would be specified as `mip_cuts_all`. For another example, to set the solver to be CPLEX, then to set a mip gap of one percent and to specify 'y' for the sub-option `numerical` to the option `emphasis` use

```
--solver=cplex --solver-options="mipgap=0.001 emphasis_numerical=y"
```

See *Sending Options to the Solver* for a discussion of passing options in a script.

Troubleshooting

Many of things that can go wrong are covered by error messages, but sometimes they can be confusing or do not provide enough information. Depending on what the troubles are, there might be ways to get a little additional information.

If there are syntax errors in the model file, for example, it can occasionally be helpful to get error messages directly from the Python interpreter rather than through Pyomo. Suppose the name of the model file is scuc.py, then

```
python scuc.py
```

can sometimes give useful information for fixing syntax errors.

When there are no syntax errors, but there troubles reading the data or generating the information to pass to a solver, then the `--verbose` option provides a trace of the execution of Pyomo. The user should be aware that for some models this option can generate a lot of output.

If there are troubles with solver (i.e., after Pyomo has output "Applying Solver"), it is often helpful to use the option `--stream-solver` that causes the solver output to be displayed rather than trapped. (See <<TeeTrue>> for information about getting this output in a script). Advanced users may wish to examine the files that are generated to be passed to a solver. The type of file generated is controlled by the `--solver-io` option and the `--keepfiles` option instructs pyomo to keep the files and output their names. However, the `--symbolic-solver-labels` option should usually also be specified so that meaningful names are used in these files.

When there seem to be troubles expressing the model, it is often useful to embed print commands in the model in places that will yield helpful information. Consider the following snippet:

```
def ax_constraint_rule(model, i):
    # return the expression for the constraint for i
    print("ax_constraint_rule was called for i=", str(i))
    return sum(model.a[i, j] * model.x[j] for j in model.J) >= model.b[i]


# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

The effect will be to output every member of the set `model.I` at the time the constraint named `model.AxbConstraint` is constructed.

Direct Interfaces to Solvers

In many applications, the default solver interface works well. However, in some cases it is useful to specify the interface using the `solver-io` option. For example, if the solver supports a direct Python interface, then the option would be specified on the command line as

```
--solver-io=python
```

Here are some of the choices:

- lp: generate a standard linear programming format file with filename extension `lp`
- nlp: generate a file with a standard format that supports linear and nonlinear optimization with filename extension `n1lp`
- os: generate an OSiL format XML file.
- python: use the direct Python interface.

Note

Not all solvers support all interfaces.

2.4.4 `BuildAction` and `BuildCheck`

This is a somewhat advanced topic. In some cases, it is desirable to trigger actions to be done as part of the model building process. The `BuildAction` function provides this capability in a Pyomo model. It takes as arguments optional index sets and a function to perform the action. For example,

```
model.BuildBpts = pyo.BuildAction(model.J, rule=bpts_build)
```

calls the function `bpts_build` for each member of `model.J`. The function `bpts_build` should have the model and a variable for the members of `model.J` as formal arguments. In this example, the following would be a valid declaration for the function:

```
def bpts_build(model, j):
```

A full example, which extends the *Symbolic Index Sets* and *Piecewise Linear Expressions* examples, is

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# abstract2piecebuild.py
# Similar to abstract2piece.py, but the breakpoints are created using a build action

import pyomo.environ as pyo

model = pyo.AbstractModel()

model.I = pyo.Set()
model.J = pyo.Set()

model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)

model.Topx = pyo.Param(default=6.1)  # range of x variables
model.PieceCnt = pyo.Param(default=100)

# the next line declares a variable indexed by the set J
model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals, bounds=(0, model.Topx))
model.y = pyo.Var(model.J, domain=pyo.NonNegativeReals)

# to avoid warnings, we set breakpoints beyond the bounds
# we are using a dictionary so that we can have different
# breakpoints for each index. But we won't.
model.bpts = {}


def bpts_build(model, j):
    model.bpts[j] = []
    for i in range(model.PieceCnt + 2):
        model.bpts[j].append(float((i * model.Topx) / model.PieceCnt))


# The object model.BuildBpts is not referred to again;
```

(continues on next page)

(continued from previous page)

```
# the only goal is to trigger the action at build time
model.BuildBpts = pyo.BuildAction(model.J, rule=bpts_build)


def f4(model, j, xp):
    # we not need j in this example, but it is passed as the index for the constraint
    return xp**4


model.ComputePieces = pyo.Piecewise(
    model.J, model.y, model.x, pw_pts=model.bpts, pw_constr_type='EQ', f_rule=f4
)


def obj_expression(model):
    return pyo.summation(model.c, model.y)


model.OBJ = pyo.Objective(rule=obj_expression)


def ax_constraint_rule(model, i):
    # return the expression for the constraint for i
    return sum(model.a[i, j] * model.x[j] for j in model.J) >= model.b[i]


# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

This example uses the build action to create a model component with breakpoints for a *Piecewise Linear Expressions* function. The `BuildAction` is triggered by the assignment to `model.BuildBpts`. This object is not referenced again, the only goal is to cause the execution of `bpts_build,` which places data in the `model.bpts` dictionary. Note that if `model.bpts` had been a `Set`, then it could have been created with an `initialize` argument to the `Set` declaration. Since it is a special-purpose dictionary to support the *Piecewise Linear Expressions* functionality in Pyomo, we use a `BuildAction`.

Another application of `BuildAction` can be initialization of Pyomo model data from Python data structures, or efficient initialization of Pyomo model data from other Pyomo model data. Consider the *Sparse Index Sets* example. Rather than using an initialization for each list of sets `NodesIn` and `NodesOut` separately using `initialize`, it is a little more efficient and probably a little clearer, to use a build action.

The full model is:

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# Isinglebuild.py
```

(continues on next page)

(continued from previous page)

```
# NodesIn and NodesOut are created by a build action using the Arcs
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.Nodes = pyo.Set()
model.Arcs = pyo.Set(dimen=2)

model.NodesOut = pyo.Set(model.Nodes, within=model.Nodes, initialize=[])
model.NodesIn = pyo.Set(model.Nodes, within=model.Nodes, initialize=[])


def Populate_In_and_Out(model):
    # loop over the arcs and put the end points in the appropriate places
    for i, j in model.Arcs:
        model.NodesIn[j].add(i)
        model.NodesOut[i].add(j)


model.In_n_Out = pyo.BuildAction(rule=Populate_In_and_Out)

model.Flow = pyo.Var(model.Arcs, domain=pyo.NonNegativeReals)
model.FlowCost = pyo.Param(model.Arcs)

model.Demand = pyo.Param(model.Nodes)
model.Supply = pyo.Param(model.Nodes)


def Obj_rule(model):
    return pyo.summation(model.FlowCost, model.Flow)


model.Obj = pyo.Objective(rule=Obj_rule, sense=pyo.minimize)


def FlowBalance_rule(model, node):
    return (
        model.Supply[node]
        + sum(model.Flow[i, node] for i in model.NodesIn[node])
        - model.Demand[node]
        - sum(model.Flow[node, j] for j in model.NodesOut[node])
        == 0
    )


model.FlowBalance = pyo.Constraint(model.Nodes, rule=FlowBalance_rule)
```

For this model, the same data file can be used as for Isinglecomm.py in *Sparse Index Sets* such as the toy data file:

```
set Nodes := CityA CityB CityC ;

set Arcs :=
```

(continues on next page)

(continued from previous page)

```
CityA CityB
CityA CityC
CityC CityB
;

param : FlowCost :=
CityA CityB 1.4
CityA CityC 2.7
CityC CityB 1.6
 ;

param Demand :=
CityA 0
CityB 1
CityC 1
;

param Supply :=
CityA 2
CityB 0
CityC 0
;
```

Build actions can also be a way to implement data validation, particularly when multiple Sets or Parameters must be analyzed. However, the the `BuildCheck` component is preferred for this purpose. It executes its rule just like a `BuildAction` but will terminate the construction of the model instance if the rule returns `False`.

2.5 Debugging Models

TODO

2.6 Contributing to Pyomo

We welcome all contributions including bug fixes, feature enhancements, and documentation improvements. Pyomo manages source code contributions via GitHub pull requests (PRs).

- *Contribution Requirements*
  - *Coding Standards*
  - *Testing*
  - *Python Version Support*
  - *Use of Generative AI*
- *Review Process*
- *Where to put contributed code*
  - *Namespaces for Contributed and Experimental Code*
- *Submitting a new Contributed Package*

  - *Example: Structure of a Contributed Package*
    - *Minimal Directory Layout*
    - *Package Initialization*
    - *Core Functionality*
    - *Tests*
- *Working on Forks and Branches*
  - *Working with my fork and the GitHub Online UI*
    - *Using GitHub UI to merge Pyomo main into a branch on your fork*
  - *Working with remotes and the git command-line*
  - *Setting up your development environment*

2.6.1 Contribution Requirements

A PR should be 1 set of related changes. PRs for large-scale non-functional changes (i.e. PEP8, comments) should be separated from functional changes. This simplifies the review process and ensures that functional changes aren't obscured by large amounts of non-functional changes.

We do not squash and merge PRs so all commits in your branch will appear in the main history. In addition to well-documented PR descriptions, we encourage modular/targeted commits with descriptive commit messages.

Coding Standards

- Formatted using black
- Passes spell checks using typos
- Adhere to Pyomo's *Development Principles*
- No use of `__author__`
- Inside `pyomo.contrib`: Contact information for the contribution maintainer (such as a Github ID) should be included in the Sphinx documentation

The first step of Pyomo's GitHub Actions workflow is to run black and a spell-checker to ensure style guide compliance and minimize typos. Before opening a pull request, please run:

```
# Auto-apply correct formatting
pip install black
black <path>
# Find typos in files
conda install typos
typos --config .github/workflows/typos.toml <path>
```

If the spell-checker returns a failure for a word that is spelled correctly, please add the word to the `.github/workflows/typos.toml` file. Note also that `black` reads from `pyproject.toml` to determine correct configuration, so if you are running `black` indirectly (for example, using an IDE integration), please ensure you are not overriding the project-level configuration set in that file.

Online Pyomo documentation is generated using Sphinx with the `napoleon` extension enabled. For API documentation we use one of these supported styles for docstrings, but we prefer the NumPy standard. Whichever you choose, we require compliant docstrings for:

- Modules
- Public and Private Classes
- Public and Private Functions

We also encourage you to include examples, especially for new features and contributions to `pyomo.contrib`.

Testing

Pyomo uses unittest, pytest, GitHub Actions, and Jenkins for testing and continuous integration. Submitted code should include tests to establish the validity of its results and/or effects. Unit tests are preferred but we also accept integration tests. We require at least 70% coverage of the lines modified in the PR and prefer coverage closer to 90%. We also require that all tests pass before a PR will be merged.

Tests must import the Pyomo test harness from `pyomo.common.unittest` instead of using Python's built-in `unittest` module directly. This wrapper extends the standard testing framework with Pyomo-specific capabilities, including support for test timeouts and Pyomo-specific assertions for comparing expressions and nested containers with numerical tolerance. Using the provided interface ensures that all tests run consistently across Pyomo's multiple CI environments. A small example is shown below:

```
import pyomo.common.unittest as unittest

class TestSomething(unittest.TestCase):
    def test_basic(self):
        self.assertEqual(1 + 1, 2)
```

Developers can also use any of the predefined `pytest` markers to categorize their tests appropriately. Markers are declared in `pyproject.toml`. Some commonly used markers are:

- `expensive`: tests that take a long time to run
- `mpi`: tests that require MPI
- `solver(id='name')`: tests for a specific solver, e.g., `@pytest.mark.solver("name")`
- `solver(vendor='name')`: tests for a set of solvers (matching up to the first underscore), e.g., `solver(vendor="gurobi")` will run tests marked with `solver("gurobi")`, `solver("gurobi_direct")`, and `solver("gurobi_persistent")`

More details about Pyomo-defined default test behavior can be found in the conftest.py file.

> **Note**
>
> If you are having issues getting tests to pass on your Pull Request, please tag any of the core developers to ask for help.

The Pyomo main branch provides a Github Actions workflow (configured in the `.github/` directory) that will test any changes pushed to a branch with a subset of the complete test harness that includes multiple virtual machines (`ubuntu`, `mac-os`, `windows`) and multiple Python versions. For existing forks, fetch and merge your fork (and branches) with Pyomo's main. For new forks, you will need to enable GitHub Actions in the 'Actions' tab on your fork. This will enable the tests to run automatically with each push to your fork.

At any point in the development cycle, a "work in progress" pull request may be opened by including '[WIP]' at the beginning of the PR title. Any pull requests marked '[WIP]' or draft will not be reviewed or merged by the core development team. However, any '[WIP]' pull request left open for an extended period of time without active development may be marked 'stale' and closed.

> **Note**
>
> Draft and WIP Pull Requests will **NOT** trigger tests. This is an effort to reduce our CI backlog. Please make use of the provided branch test suite for evaluating / testing draft functionality.

Python Version Support

By policy, Pyomo supports and tests the currently supported Python versions, as can be seen on Status of Python Versions. It is expected that tests will pass for all of the supported and tested versions of Python, unless otherwise stated.

At the time of the first Pyomo release after the end-of-life of a minor Python version, we will remove testing and support for that Python version.

This will also result in a bump in the minor Pyomo version.

For example, assume Python 3.A is declared end-of-life while Pyomo is on version 6.3.Y. After the release of Pyomo 6.3.(Y+1), Python 3.A will be removed, and the next Pyomo release will be 6.4.0.

Use of Generative AI

Pyomo contributors are welcome to use AI-assisted coding tools (e.g., Copilot/Codex) to draft code, tests, documentation, and commit messages. However, the author remains fully responsible for the correctness, security, licensing compliance, and maintainability of every changed line and must be prepared to explain design choices and review changed code personally before submitting. The use of PR and issue templates is mandatory and contributions that do not use the templates will be closed.

PRs with substantial AI-generated portions often require significantly more reviewer time (including extra scrutiny of tests) and may therefore take longer to review or be deprioritized, especially when the author is not actively engaging with maintainers (e.g., responding promptly on the PR, coordinating by email, or participating in developer calls). To help us review efficiently, contributors must disclose whether and how AI tools were used and highlight any areas of uncertainty or where they want focused reviewer feedback.

The use of an AI agent to autonomously open, or comment on, PRs or issues is not permitted. When interacting with maintainers please do not use AI to speak for you. The Pyomo Developers want to interact with humans, not chatbots. We reserve the right to close without review PRs and issues we deem to be "AI slop".

We recognize that generative AI tools are rapidly evolving and this AI contribution policy will be updated regularly based on our observations. The following list includes specific items we are seeing with AI-generated code contributions and our expectations:

- **Use of ``mock`` in unit tests is prohibited**: We do not allow the use of `mock` in our unit tests unless the developer can provide solid justification as to its necessity. We have observed that AI-generated unit tests excessively use `mock` to write weak tests that ignore the broader context of the code being tested.
- **Keep comments concise and curated**: Humans are reading every comment on every PR and issue. Help us out by keeping comments short and direct.

2.6.2 Review Process

After a PR is opened it will be reviewed by at least two members of the core development team. The core development team consists of anyone with write-access to the Pyomo repository. PRs opened by a core developer only require one review. The reviewers will decide if they think a PR should be merged or if more changes are necessary.

Reviewers look for:

Core and Addons:
Code rigor, standards compliance, test coverage above a threshold, and avoidance of unintended side effects (e.g., regressions or backwards incompatibilities)

Devel:
Basic code correctness and clarity, with an understanding that these areas are experimental and evolving

All areas:
Code formatting (using `black`), documentation, and tests

Note

For more information about Pyomo's development principles and the stability expectations for `addons` and `devel`, see *Development Principles*.

The core development team tries to review PRs in a timely manner, but we make no guarantees on review timeframes. Smaller, focused PRs are preferred and are generally reviewed more quickly. Larger PRs require more review effort and may take significantly longer. In addition, PRs might not be reviewed in the order in which they are opened.

The development team makes use of the "Draft" PR status for communication between the PR author and reviewers. Reviewers will convert PRs that are waiting for input from the PR author (e.g., due to test failures, insufficient coverage, or requested changes) back to "Draft" state. Once the author has addressed the issues, they should mark the PR as "ready to review" to signal the reviewers and request updated reviews.

Note

Reviewers will *not* monitor or review PRs still marked "Draft".

Note

PRs left in "Draft" state for an extended period of time may be proposed for closure to reduce impact on the testing infrastructure.

2.6.3 Where to put contributed code

In order to contribute to Pyomo, you must first make a fork of the Pyomo git repository. Next, you should create a branch on your fork dedicated to the development of the new feature or bug fix you're interested in. Once you have this branch checked out, you can start coding. Bug fixes and minor enhancements to existing Pyomo functionality should be made in the appropriate files in the Pyomo code base.

We refer to the modules that form the foundation of the Pyomo environment as `pyomo` core. This includes the base expression systems, modeling components, model compilers, and solver interfaces. The core development team has committed to maintaining these capabilities, adhering to the strictest policies for testing and backwards compatibility.

Larger features, new modeling components, or experimental functionality should be placed in one of Pyomo's extension namespaces, described below.

Namespaces for Contributed and Experimental Code

Pyomo organizes non-core functionality into a small number of clearly defined namespaces. Contributors should place new functionality according to its intended stability and maintenance expectations:

- `pyomo.addons` – For mostly stable, supported extensions that build on the Pyomo core. These packages are maintained by dedicated contributors, follow Pyomo's coding and testing standards, and adhere to the same backwards compatibility and deprecation policies as the rest of the codebase.
- `pyomo.devel` – For experimental or rapidly evolving contributions. These modules serve as early experimentation for research ideas, prototypes, or specialized modeling components. Functionality under this namespace

may change or be removed between releases without deprecation warnings.

- `pyomo.unsupported` - For contributions that no longer have an active maintainer nor any future development plans. Functionality under this namespace may not work and is **NOT** tested through the standard test harness.

This tiered namespace structure provides contributors a clear pathway from **experimentation to supported integration**, while protecting users from unexpected changes in stable areas of the codebase.

Namespace	Intended Use	Stability
`pyomo.devel`	Active research and experimental code	Unstable; APIs may change without warning
`pyomo.addons`	Mostly stable, supported extensions maintained by contributors	Mostly stable APIs; follow Pyomo's standards
`pyomo.unsupported`	Unsupported, unmaintained code	No guarantee of functionality and no regular testing
`pyomo`	Core Pyomo modeling framework	Fully supported and versioned

2.6.4 Submitting a new Contributed Package

Including contributed packages in the Pyomo source tree provides a convenient mechanism for defining new functionality that can be optionally deployed by users. We expect this mechanism to include Pyomo extensions and experimental modeling capabilities. However, contributed packages are treated as optional packages, which are not necessarily maintained by the Pyomo developer team. Thus, it is the responsibility of the code contributor to keep these packages up-to-date.

Contributed packages will be considered as pull requests, which will be reviewed by the Pyomo developer team. Specifically, this review will consider the suitability of the proposed capability, whether tests are available to check the execution of the code, and whether documentation is available to describe the capability. Contributed packages will be tested along with Pyomo. If test failures arise, then these packages will be disabled and an issue will be created to resolve these test failures. The Pyomo team reserves the right to remove contributed packages that are not maintained.

When submitting a new package (under either `addons` or `devel`), please ensure that:

- The package has at least one maintainer responsible for its upkeep.
- The code includes tests that can be run through Pyomo's continuous integration framework.
- The package includes documentation that clearly describes its purpose and usage, preferably as online documentation in `doc/OnlineDocs`.
- Optional dependencies are properly declared in `setup.py` under the appropriate `[optional]` section.
- The contribution passes all standard style and formatting checks.

Example: Structure of a Contributed Package

This section illustrates a minimal example of how a contributed package may be structured within the `pyomo.devel` or `pyomo.addons` namespaces. This example is provided for documentation purposes only and is not included as source code in the Pyomo repository.

Minimal Directory Layout

At a minimum, a contributed package should follow a structure similar to the following:

```
pyomo/devel/example_package/
 __init__.py
```

(continues on next page)

(continued from previous page)

```
core.py
tests/
    __init__.py
    test_example_package.py
```

Package Initialization

The package `__init__.py` file should expose the primary public interfaces of the package and avoid unnecessary imports. Contributed packages must be safe to import as optional components and should not introduce side effects at import time.

For example:

```
# pyomo/devel/example_package/__init__.py
from pyomo.devel.example_package.core import example_function
```

Core Functionality

The main functionality of the contributed package should be implemented in one or more modules within the package directory (for example, `core.py`). These modules should follow Pyomo's coding standards, documentation requirements, and dependency management policies.

Tests

All contributed packages must include tests. Tests should be placed in a `tests` subpackage and use the Pyomo test harness provided by `pyomo.common.unittest`.

At a minimum, tests should verify that the package can be imported and that its primary functionality executes as expected. For example:

```
import pyomo.common.unittest as unittest

class TestExamplePackage(unittest.TestCase):
    def test_import(self):
        import pyomo.devel.example_package
```

Tests for contributed packages are run as part of the Pyomo test suite and must not have an unconditional import of optional dependencies. Tests that exercise functionality requiring optional dependencies must be properly guarded (e.g., with `@unittest.skipIf()` / `@unittest.skipUnless()`). Pyomo provides a standard tool for supporting the delayed import of optional dependencies (see `attempt_import()`) as well as a central location for importing many common optional dependencies (see *`pyomo.common.dependencies`*). For example, tests that require `numpy` may be marked using the Pyomo test harness as follows:

```
import pyomo.common.unittest as unittest
from pyomo.common.dependencies import numpy as np, numpy_available

@unittest.skipIf(not numpy_available, "NumPy is not available")
class TestExampleWithNumpy(unittest.TestCase):
    def test_numpy_functionality(self):
        a = np.array([1, 2, 3])
        self.assertEqual(a.sum(), 6)
```

2.6.5 Working on Forks and Branches

All Pyomo development should be done on forks of the Pyomo repository. In order to fork the Pyomo repository, visit https://github.com/Pyomo/pyomo, click the "Fork" button in the upper right corner, and follow the instructions.

This section discusses two recommended workflows for contributing pull-requests to Pyomo. The first workflow, labeled *Working with my fork and the GitHub Online UI*, does not require the use of 'remotes', and suggests updating your fork using the GitHub online UI. The second workflow, labeled *Working with remotes and the git command-line*, outlines a process that defines separate remotes for your fork and the main Pyomo repository.

More information on git can be found at https://git-scm.com/book/en/v2. Section 2.5 has information on working with remotes.

Working with my fork and the GitHub Online UI

After creating your fork (per the instructions above), you can then clone your fork of the repository with

```
git clone https://github.com/<username>/pyomo.git
```

For new development, we strongly recommend working on feature branches. When you have a new feature to implement, create the branch with the following.

```
cd pyomo/     # to make sure you are in the folder managed by git
git branch <branch_name>
git checkout <branch_name>
```

Development can now be performed. When you are ready, commit any changes you make to your local repository. This can be done multiple times with informative commit messages for different tasks in the feature development.

```
git add <filename>
git status  # to check that you have added the correct files
git commit -m 'informative commit message to describe changes'
```

In order to push the changes in your local branch to a branch on your fork, use

```
git push origin <branch_name>
```

When you have completed all the changes and are ready for a pull request, make sure all the changes have been pushed to the branch <branch_name> on your fork.

- visit https://github.com/<username>/pyomo.
- Just above the list of files and directories in the repository, you should see a button that says "Branch: main". Click on this button, and choose the correct branch.
- Click the "New pull request" button just to the right of the "Branch: <branch_name>" button.
- Fill out the pull request template and click the green "Create pull request" button.

At times during your development, you may want to merge changes from the Pyomo main development branch into the feature branch on your fork and in your local clone of the repository.

Using GitHub UI to merge Pyomo main into a branch on your fork

To update your fork, you will actually be merging a pull-request from the head Pyomo repository into your fork.

- Visit https://github.com/Pyomo/pyomo.
- Click on the "New pull request" button just above the list of files and directories.

- You will see the title “Compare changes” with some small text below it which says “Compare changes across branches, commits, tags, and more below. If you need to, you can also compare across forks.” Click the last part of this: “compare across forks”.
- You should now see four buttons just below this: “base repository: Pyomo/pyomo”, “base: main”, “head repository: Pyomo/pyomo”, and “compare: main”. Click the leftmost button and choose “<username>/Pyomo”.
- Then click the button which is second to the left, and choose the branch which you want to merge Pyomo main into. The four buttons should now read: “base repository: <username>/pyomo”, “base: <branch_name>”, “head repository: Pyomo/pyomo”, and “compare: main”. This is setting you up to merge a pull-request from Pyomo’s main branch into your fork’s <branch_name> branch.
- You should also now see a pull request template. If you fill out the pull request template and click “Create pull request”, this will create a pull request which will update your fork and branch with any changes that have been made to the main branch of Pyomo.
- You can then merge the pull request by clicking the green “Merge pull request” button from your fork on GitHub.

Working with remotes and the git command-line

After you have created your fork, you can clone the fork and setup git ‘remotes’ that allow you to merge changes from (and to) different remote repositories. Below, we have included a set of recommendations, but, of course, there are other valid GitHub workflows that you can adopt.

The following commands show how to clone your fork and setup two remotes, one for your fork, and one for the head Pyomo repository.

```
git clone https://github.com/<username>/pyomo.git
git remote rename origin my-fork
git remote add head-pyomo https://github.com/pyomo/pyomo.git
```

Note, you can see a list of your remotes with

```
git remote -v
```

The commands for creating a local branch and performing local commits are the same as those listed in the previous section above. Below are some common tasks based on this multi-remote setup.

If you have changes that have been committed to a local feature branch (<branch_name>), you can push these changes to the branch on your fork with,

```
git push my-fork <branch_name>
```

In order to update a local branch with changes from a branch of the Pyomo repository,

```
git checkout <branch_to_update>
git fetch head-pyomo
git merge head-pyomo/<branch_to_update_from> --ff-only
```

The “–ff-only” only allows a merge if the merge can be done by a fast-forward. If you do not require a fast-forward, you can drop this option. The most common concrete example of this would be

```
git checkout main
git fetch head-pyomo
git merge head-pyomo/main --ff-only
```

The above commands pull changes from the main branch of the head Pyomo repository into the main branch of your local clone. To push these changes to the main branch on your fork,

```
git push my-fork main
```

Setting up your development environment

After cloning your fork, you will want to install Pyomo from source.

Step 1 (recommended): Create a new `conda` environment.

```
conda create --name pyomodev
```

You may change the environment name from `pyomodev` as you see fit. Then activate the environment:

```
conda activate pyomodev
```

Step 2 (optional): Install PyUtilib

The hard dependency on PyUtilib was removed in Pyomo 6.0.0. There is still a soft dependency for any code related to `pyomo.dataportal.plugins.sheet`.

If your contribution requires PyUtilib, you will likely need the main branch of PyUtilib to contribute. Clone a copy of the repository in a new directory:

```
git clone https://github.com/PyUtilib/pyutilib
```

Then in the directory containing the clone of PyUtilib run:

```
python -m pip install -e .
```

Step 3: Install Pyomo

Finally, move to the directory containing the clone of your Pyomo fork and run:

```
python -m pip install -e .[tests,docs,optional]
```

This command registers the cloned code with the active Python environment (`pyomodev`) and installs all possible optional dependencies. Using `-e` ensures that your changes to the source code for `pyomo` are automatically used by the active environment. You can create another conda environment to switch to alternate versions of pyomo (e.g., stable).

> **Note**
>
> The `optional` and `docs` dependencies are not strictly required; however, we recommend installing them to ensure that a large number of tests can be run locally. Optional packages that are not available will cause tests to skip.

CHAPTER

THREE

EXPLANATIONS

3.1 Pyomo Philosophy

3.1.1 Abstract Models

Note: TODO: this is a copy of "Abstract vs Concrete" from Getting Started. This should be expanded here.

A mathematical model can be defined using symbols that represent data values. For example, the following equations represent a linear program (LP) to find optimal values for the vector x with parameters n and b, and parameter vectors a and c:

$$\begin{array}{lll} \min & \sum_{j=1}^n c_j x_j & \\ \text{s.t.} & \sum_{j=1}^n a_{ij} x_j \geq b_i & \forall i = 1 \ldots m \\ & x_j \geq 0 & \forall j = 1 \ldots n \end{array}$$

Note: As a convenience, we use the symbol $\forall$ to mean "for all" or "for each."

We call this an *abstract* or *symbolic* mathematical model since it relies on unspecified parameter values. Data values can be used to specify a *model instance*. The `AbstractModel` class provides a context for defining and initializing abstract optimization models in Pyomo when the data values will be supplied at the time a solution is to be obtained.

In many contexts, a mathematical model can and should be directly defined with the data values supplied at the time of the model definition. We call these *concrete* mathematical models. For example, the following LP model is a concrete instance of the previous abstract model:

$$\begin{array}{ll} \min & 2x_1 + 3x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{array}$$

The `ConcreteModel` class is used to define concrete optimization models in Pyomo.

Note: Python programmers will probably prefer to write concrete models, while users of some other algebraic modeling languages may tend to prefer to write abstract models. The choice is largely a matter of taste; some applications may be a little more straightforward using one or the other.

3.1.2 Pyomo Component Design

TODO

3.1.3 Pyomo Expressions

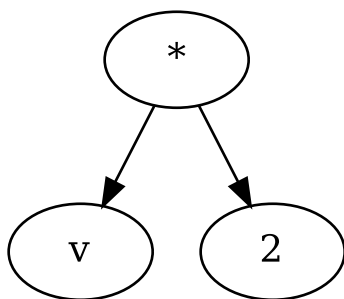
Warning

This documentation does not explicitly reference objects in pyomo.core.kernel. While the Pyomo5 expression system works with pyomo.core.kernel objects, the documentation of these documents was not sufficient to appropriately describe the use of kernel objects in expressions.

Pyomo supports the declaration of symbolic expressions that represent objectives, constraints and other optimization modeling components. Pyomo expressions are represented in an expression tree, where the leaves are operands, such as constants or variables, and the internal nodes contain operators. Pyomo relies on so-called magic methods to automate the construction of symbolic expressions. For example, consider an expression `e` declared as follows:

```
M = pyo.ConcreteModel()
M.v = pyo.Var()

e = M.v * 2
```

Python determines that the magic method `__mul__` is called on the `M.v` object, with the argument `2`. This method returns a Pyomo expression object `ProductExpression` that has arguments `M.v` and `2`. This represents the following symbolic expression tree:



Note

End-users will not likely need to know details related to how symbolic expressions are generated and managed in Pyomo. Thus, most of the following documentation of expressions in Pyomo is most useful for Pyomo developers. However, the discussion of runtime performance in the first section will help end-users write large-scale models.

Building Expressions Faster

Expression Generation

Pyomo expressions can be constructed using native binary operators in Python. For example, a sum can be created in a simple loop:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))

s = 0
for i in range(5):
    s = s + M.x[i]
```

Additionally, Pyomo expressions can be constructed using functions that iteratively apply Python binary operators. For example, the Python `sum()` function can be used to replace the previous loop:

```
s = sum(M.x[i] for i in range(5))
```

The `sum()` function is both more compact and more efficient. Using `sum()` avoids the creation of temporary variables, and the summation logic is executed in the Python interpreter while the loop is interpreted.

Linear, Quadratic and General Nonlinear Expressions

Pyomo can express a very wide range of algebraic expressions, and there are three general classes of expressions that are recognized by Pyomo:

- **linear polynomials**
- **quadratic polynomials**
- **nonlinear expressions**, including higher-order polynomials and expressions with intrinsic functions

These classes of expressions are leveraged to efficiently generate compact representations of expressions, and to transform expression trees into standard forms used to interface with solvers. Note that There not all quadratic polynomials are recognized by Pyomo; in other words, some quadratic expressions are treated as nonlinear expressions.

For example, consider the following quadratic polynomial:

```
s = sum(M.x[i] for i in range(5)) ** 2
```

This quadratic polynomial is treated as a nonlinear expression unless the expression is explicitly processed to identify quadratic terms. This *lazy* identification of of quadratic terms allows Pyomo to tailor the search for quadratic terms only when they are explicitly needed.

Pyomo Utility Functions

Pyomo includes several similar functions that can be used to create expressions:

prod
: A function to compute a product of Pyomo expressions.

quicksum
: A function to efficiently compute a sum of Pyomo expressions.

sum_product
: A function that computes a generalized dot product.

prod

The *prod* function is analogous to the builtin `sum()` function. Its main argument is a variable length argument list, `args`, which represents expressions that are multiplied together. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))
M.z = pyo.Var()

# The product M.x[0] * M.x[1] * ... * M.x[4]
e1 = pyo.prod(M.x[i] for i in M.x)

# The product M.x[0]*M.z
e2 = pyo.prod([M.x[0], M.z])

# The product M.z*(M.x[0] + ... + M.x[4])
e3 = pyo.prod([sum(M.x[i] for i in M.x), M.z])
```

quicksum

The behavior of the *quicksum* function is similar to the builtin `sum()` function, but this function often generates a more compact Pyomo expression. Its main argument is a variable length argument list, `args`, which represents expressions that are summed together. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))

# Summation using the Python sum() function
e1 = sum(M.x[i] ** 2 for i in M.x)

# Summation using the Pyomo quicksum function
e2 = pyo.quicksum(M.x[i] ** 2 for i in M.x)
```

The summation is customized based on the `start` and `linear` arguments. The `start` defines the initial value for summation, which defaults to zero. If `start` is a numeric value, then the `linear` argument determines how the sum is processed:

- If `linear` is `False`, then the terms in `args` are assumed to be nonlinear.
- If `linear` is `True`, then the terms in `args` are assumed to be linear.
- If `linear` is `None`, the first term in `args` is analyze to determine whether the terms are linear or nonlinear.

This argument allows the *quicksum* function to customize the expression representation used, and specifically a more compact representation is used for linear polynomials. The *quicksum* function can be slower than the builtin `sum()` function, but this compact representation can generate problem representations more quickly.

Consider the following example:

```
M = pyo.ConcreteModel()
M.A = pyo.RangeSet(100000)
M.p = pyo.Param(M.A, mutable=True, initialize=1)
M.x = pyo.Var(M.A)

start = time.time()
e = sum((M.x[i] - 1) ** M.p[i] for i in M.A)
```

(continues on next page)

(continued from previous page)

```
print("sum:      %f" % (time.time() - start))

start = time.time()
generate_standard_repn(e)
print("repn:     %f" % (time.time() - start))

start = time.time()
e = pyo.quicksum((M.x[i] - 1) ** M.p[i] for i in M.A)
print("quicksum: %f" % (time.time() - start))

start = time.time()
generate_standard_repn(e)
print("repn:     %f" % (time.time() - start))
```

The sum consists of linear terms because the exponents are one. The following output illustrates that quicksum can identify this linear structure to generate expressions more quickly:

```
sum:      1.447861
repn:     0.870225
quicksum: 1.388344
repn:     0.864316
```

If `start` is not a numeric value, then the *quicksum* sets the initial value to `start` and executes a simple loop to sum the terms. This allows the sum to be stored in an object that is passed into the function (e.g. the linear context manager `linear_expression`).

> **Warning**
>
> By default, `linear` is `None`. While this allows for efficient expression generation in normal cases, there are circumstances where the inspection of the first term in `args` is misleading. Consider the following example:
>
> ```
> M = pyo.ConcreteModel()
> M.x = pyo.Var(range(5))
>
> e = pyo.quicksum(M.x[i] ** 2 if i > 0 else M.x[i] for i in range(5))
> ```
>
> The first term created by the generator is linear, but the subsequent terms are nonlinear. Pyomo gracefully transitions to a nonlinear sum, but in this case *quicksum* is doing additional work that is not useful.

sum_product

The *sum_product* function supports a generalized dot product. The `args` argument contains one or more components that are used to create terms in the summation. If the `args` argument contains a single components, then its sequence of terms are summed together; the sum is equivalent to calling *quicksum*. If two or more components are provided, then the result is the summation of their terms multiplied together. For example:

```
M = pyo.ConcreteModel()
M.z = pyo.RangeSet(5)
M.x = pyo.Var(range(10))
M.y = pyo.Var(range(10))
```

(continues on next page)

(continued from previous page)

```
# Sum the elements of x
e1 = pyo.sum_product(M.x)

# Sum the product of elements in x and y
e2 = pyo.sum_product(M.x, M.y)

# Sum the product of elements in x and y, over the index set z
e3 = pyo.sum_product(M.x, M.y, index=M.z)
```

The `denom` argument specifies components whose terms are in the denominator. For example:

```
# Sum the product of x_i/y_i
e1 = pyo.sum_product(M.x, denom=M.y)

# Sum the product of 1/(x_i*y_i)
e2 = pyo.sum_product(denom=(M.x, M.y))
```

The terms summed by this function are explicitly specified, so *sum_product* can identify whether the resulting expression is linear, quadratic or nonlinear. Consequently, this function is typically faster than simple loops, and it generates compact representations of expressions..

Finally, note that the *dot_product* function is an alias for *sum_product*.

Design Overview

Historical Comparison

This document describes the “Pyomo5” expressions, which were introduced in Pyomo 5.6. The main differences between “Pyomo5” expressions and the previous expression system, called “Coopr3”, are:

- Pyomo5 supports both CPython and PyPy implementations of Python, while Coopr3 only supports CPython.

  The key difference in these implementations is that Coopr3 relies on CPython reference counting, which is not part of the Python language standard. Hence, this implementation is not guaranteed to run on other implementations of Python.

  Pyomo5 does not rely on reference counting, and it has been tested with PyPy. In the future, this should allow Pyomo to support other Python implementations (e.g. Jython).

- Pyomo5 expression objects are immutable, while Coopr3 expression objects are mutable.

  This difference relates to how expression objects are managed in Pyomo. Once created, Pyomo5 expression objects cannot be changed. Further, the user is guaranteed that no “side effects” occur when expressions change at a later point in time. By contrast, Coopr3 allows expressions to change in-place, and thus “side effects” make occur when expressions are changed at a later point in time. (See discussion of entanglement below.)

- Pyomo5 provides more consistent runtime performance than Coopr3.

  While this documentation does not provide a detailed comparison of runtime performance between Coopr3 and Pyomo5, the following performance considerations also motivated the creation of Pyomo5:

  - There were surprising performance inconsistencies in Coopr3. For example, the following two loops had dramatically different runtime:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(100))

# This loop is fast.
e = 0
for i in range(100):
    e = e + M.x[i]

# This loop is slow.
e = 0
for i in range(100):
    e = M.x[i] + e
```

- Coopr3 eliminates side effects by automatically cloning sub-expressions. Unfortunately, this can easily lead to unexpected cloning in models, which can dramatically slow down Pyomo model generation. For example:

```
M = pyo.ConcreteModel()
M.p = pyo.Param(initialize=3)
M.q = 1 / M.p
M.x = pyo.Var(range(100))

# The value M.q is cloned every time it is used.
e = 0
for i in range(100):
    e = e + M.x[i] * M.q
```

- Coopr3 leverages recursion in many operations, including expression cloning. Even simple non-linear expressions can result in deep expression trees where these recursive operations fail because Python runs out of stack space.

- The immutable representation used in Pyomo5 requires more memory allocations than Coopr3 in simple loops. Hence, a pure-Python execution of Pyomo5 can be 10% slower than Coopr3 for model construction. But when Cython is used to optimize the execution of Pyomo5 expression generation, the runtimes for Pyomo5 and Coopr3 are about the same. (In principle, Cython would improve the runtime of Coopr3 as well, but the limitations noted above motivated a new expression system in any case.)

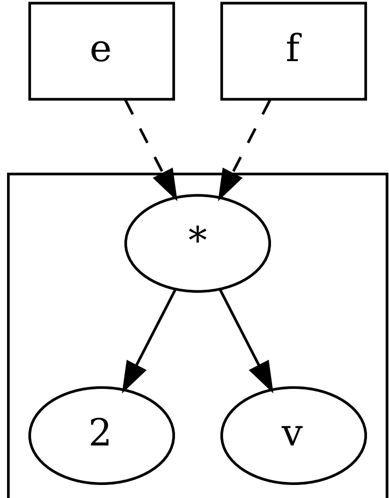
Expression Entanglement and Mutability

Pyomo fundamentally relies on the use of magic methods in Python to generate expression trees, which means that Pyomo has very limited control for how expressions are managed in Python. For example:

- Python variables can point to the same expression tree

```
M = pyo.ConcreteModel()
M.v = pyo.Var()

e = f = 2 * M.v
```

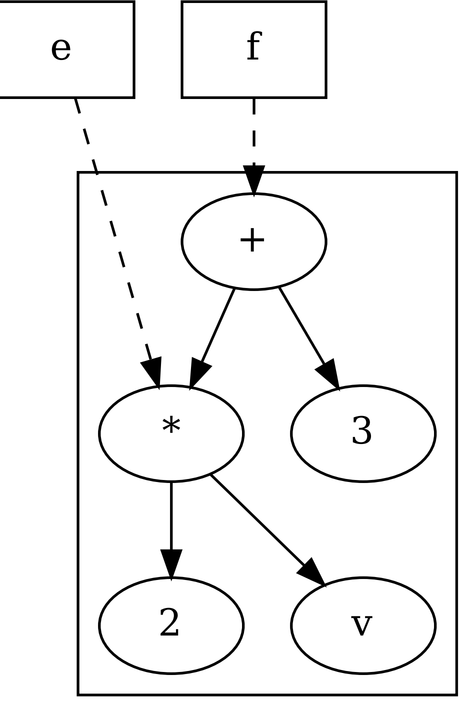
This is illustrated as follows:



- A variable can point to a sub-tree that another variable points to

```
M = pyo.ConcreteModel()
M.v = pyo.Var()

e = 2 * M.v
f = e + 3
```

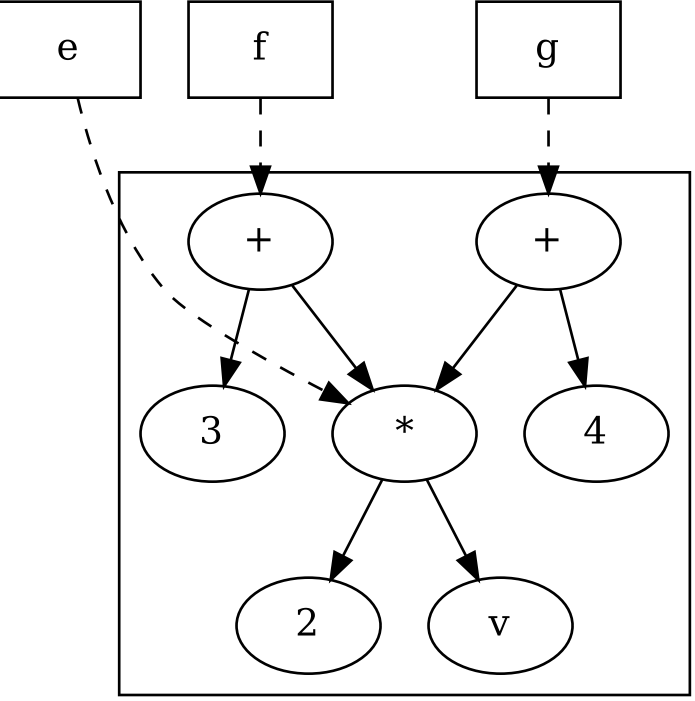
This is illustrated as follows:



- Two expression trees can point to the same sub-tree

```
M = pyo.ConcreteModel()
M.v = pyo.Var()

e = 2 * M.v
f = e + 3
g = e + 4
```

This is illustrated as follows:



In each of these examples, it is almost impossible for a Pyomo user or developer to detect whether expressions are being shared. In CPython, the reference counting logic can support this to a limited degree. But no equivalent mechanisms are available in PyPy and other Python implementations.

Entangled Sub-Expressions

We say that expressions are *entangled* if they share one or more sub-expressions. The first example above does not represent entanglement, but rather the fact that multiple Python variables can point to the same expression tree. In the second and third examples, the expressions are entangled because the subtree represented by `e` is shared. However, if a leave node like `M.v` is shared between expressions, we do not consider those expressions entangled.

Expression entanglement is problematic because shared expressions complicate the expected behavior when sub-expressions are changed. Consider the following example:

```
M = pyo.ConcreteModel()
M.v = pyo.Var()
M.w = pyo.Var()

e = 2 * M.v
f = e + 3

e += M.w
```

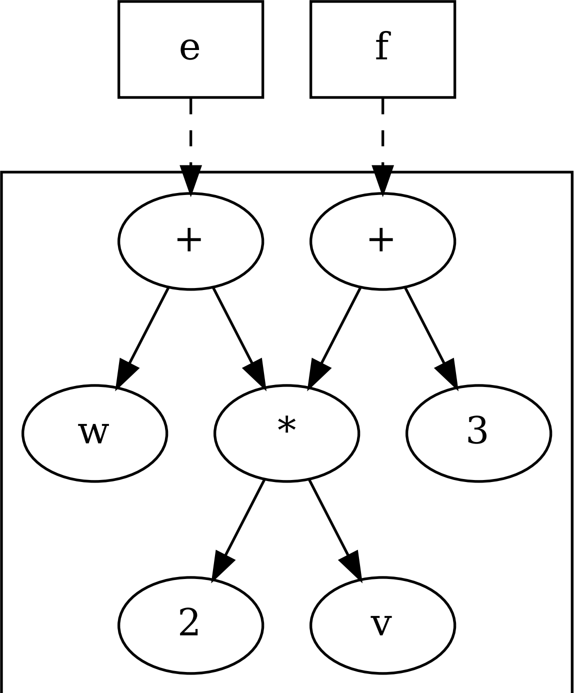
What is the value of `e` after `M.w` is added to it? What is the value of `f`? The answers to these questions are not immediately obvious, and the fact that Coopr3 uses mutable expression objects makes them even less clear. However,

Pyomo5 and Coopr3 enforce the following semantics:

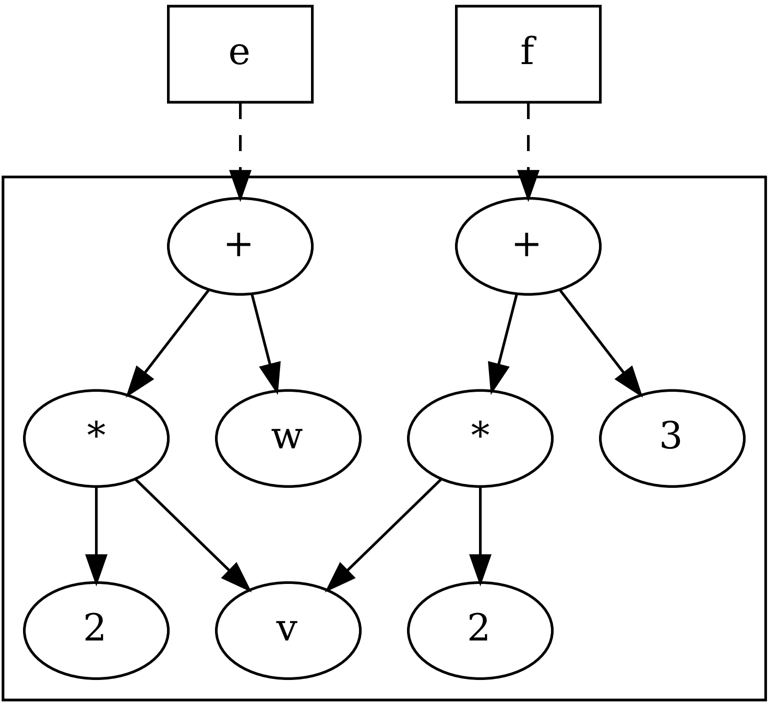
> A change to an expression e that is a sub-expression of f does not change the expression tree for f.

This property ensures a change to an expression does not create side effects that change the values of other, previously defined expressions.

For instance, the previous example results in the following (in Pyomo5):



With Pyomo5 expressions, each sub-expression is immutable. Thus, the summation operation generates a new expression `e` without changing existing expression objects referenced in the expression tree for `f`. By contrast, Coopr3 imposes the same property by cloning the expression `e` before added `M.w`, resulting in the following:



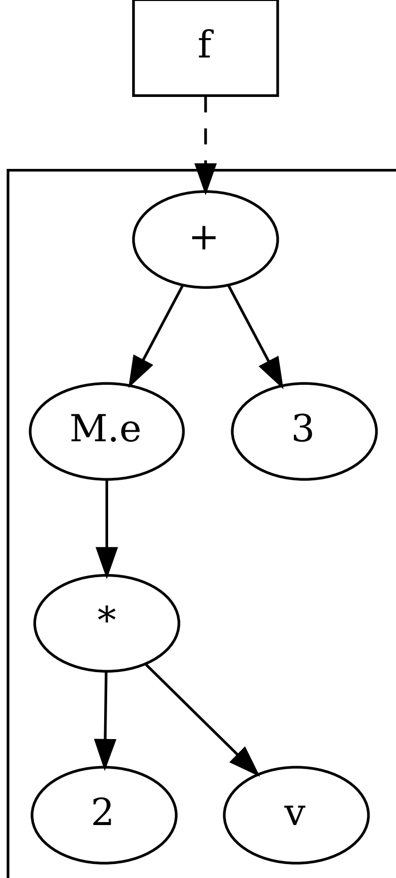
This example also illustrates that leaves may be shared between expressions.

Mutable Expression Components

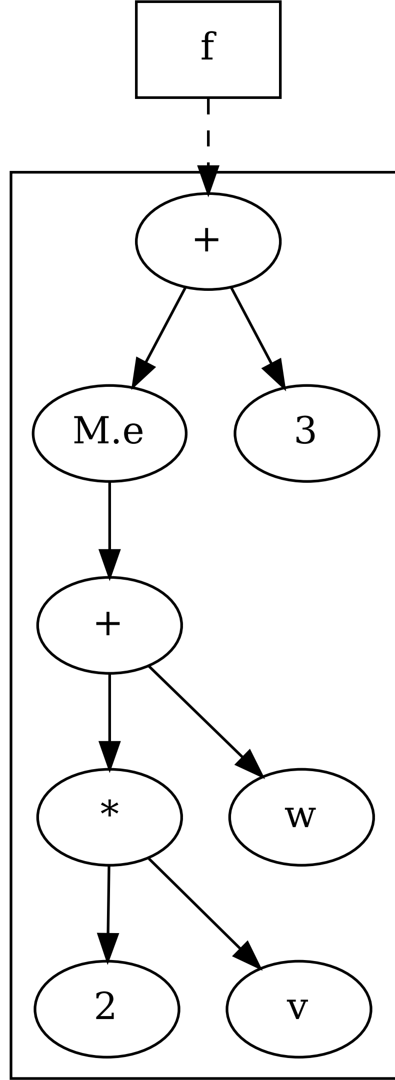
There is one important exception to the entanglement property described above. The `Expression` component is treated as a mutable expression when shared between expressions. For example:

```
M = pyo.ConcreteModel()
M.v = pyo.Var()
M.w = pyo.Var()

M.e = pyo.Expression(expr=2 * M.v)
f = M.e + 3

M.e += M.w
```

Here, the expression `M.e` is a so-called *named expression* that the user has declared. Named expressions are explicitly intended for re-use within models, and they provide a convenient mechanism for changing sub-expressions in complex applications. In this example, the expression tree is as follows before `M.w` is added:



And the expression tree is as follows after `M.w` is added.



When considering named expressions, Pyomo5 and Coopr3 enforce the following semantics:

> A change to a named expression *e* that is a sub-expression of *f* changes the expression tree for *f*, because *f* continues to point to *e* after it is changed.

Design Details

Warning

Pyomo expression trees are not composed of Python objects from a single class hierarchy. Consequently, Pyomo relies on duck typing to ensure that valid expression trees are created.

Most Pyomo expression trees have the following form

1. Interior nodes are objects that inherit from the `ExpressionBase` class. These objects typically have one or more child nodes. Linear expression nodes do not have child nodes, but they are treated as interior nodes in the expression tree because they references other leaf nodes.
2. Leaf nodes are numeric values, parameter components and variable components, which represent the *inputs* to the expression.

Expression Classes

Expression classes typically represent unary and binary operations. The following table describes the standard operators in Python and their associated Pyomo expression class:

Operation	Python Syntax	Pyomo Class
sum	`x + y`	`SumExpression`
product	`x * y`	`ProductExpression`
negation	`- x`	`NegationExpression`
division	`x / y`	`DivisionExpression`
power	`x ** y`	`PowExpression`
inequality	`x <= y`	`InequalityExpression`
equality	`x == y`	`EqualityExpression`

Additionally, there are a variety of other Pyomo expression classes that capture more general logical relationships, which are summarized in the following table:

Operation	Example	Pyomo Class
external function	`myfunc(x,y,z)`	`ExternalFunctionExpression`
logical if-then-else	`Expr_if(IF=x, THEN=y, ELSE=z)`	`Expr_ifExpression`
intrinsic function	`sin(x)`	`UnaryFunctionExpression`
absolute function	`abs(x)`	`AbsExpression`

Expression objects are immutable. Specifically, the list of arguments to an expression object (a.k.a. the list of child nodes in the tree) cannot be changed after an expression class is constructed. To enforce this property, expression objects have a standard API for accessing expression arguments:

- `args` - a class property that returns a generator that yields the expression arguments
- `arg(i)` - a function that returns the `i`-th argument
- `nargs()` - a function that returns the number of expression arguments

Warning

Developers should never use the `_args_` property directly! The semantics for the use of this data has changed since earlier versions of Pyomo. For example, in some expression classes the the value `nargs()` may not equal `len(_args_)`!

Expression trees can be categorized in four different ways:

- constant expressions - expressions that do not contain numeric constants and immutable parameters.
- mutable expressions - expressions that contain mutable parameters but no variables.

- potentially variable expressions - expressions that contain variables, which may be fixed.
- fixed expressions - expressions that contain variables, all of which are fixed.

These three categories are illustrated with the following example:

```
m = pyo.ConcreteModel()
m.p = pyo.Param(default=10, mutable=False)
m.q = pyo.Param(default=10, mutable=True)
m.x = pyo.Var()
m.y = pyo.Var(initialize=1)
m.y.fixed = True
```

The following table describes four different simple expressions that consist of a single model component, and it shows how they are categorized:

Category	m.p	m.q	m.x	m.y
constant	True	False	False	False
not potentially variable	True	True	False	False
potentially_variable	False	False	True	True
fixed	True	True	False	True

Expressions classes contain methods to test whether an expression tree is in each of these categories. Additionally, Pyomo includes custom expression classes for expression trees that are *not potentially variable*. These custom classes will not normally be used by developers, but they provide an optimization of the checks for potentially variability.

Special Expression Classes

The following classes are *exceptions* to the design principles describe above.

Named Expressions

Named expressions allow for changes to an expression after it has been constructed. For example, consider the expression `f` defined with the `Expression` component:

```
M = pyo.ConcreteModel()
M.v = pyo.Var()
M.w = pyo.Var()

M.e = pyo.Expression(expr=2 * M.v)
f = M.e + 3  # f == 2*v + 3
M.e += M.w  # f == 2*v + 3 + w
```

Although `f` is an immutable expression, whose definition is fixed, a sub-expressions is the named expression `M.e`. Named expressions have a mutable value. In other words, the expression that they point to can change. Thus, a change to the value of `M.e` changes the expression tree for any expression that includes the named expression.

Note

The named expression classes are not implemented as sub-classes of `NumericExpression`. This reflects design constraints related to the fact that these are modeling components that belong to class hierarchies other than the expression class hierarchy, and Pyomo's design prohibits the use of multiple inheritance for these classes.

Linear Expressions

Pyomo includes a special expression class for linear expressions. The class LinearExpression provides a compact description of linear polynomials. Specifically, it includes a constant value constant and two lists for coefficients and variables: linear_coefs and linear_vars.

This expression object does not have arguments, and thus it is treated as a leaf node by Pyomo visitor classes. Further, the expression API functions described above do not work with this class. Thus, developers need to treat this class differently when walking an expression tree (e.g. when developing a problem transformation).

Sum Expressions

Pyomo does not have a binary sum expression class. Instead, it has an n-ary summation class, SumExpression. This expression class treats sums as n-ary sums for efficiency reasons; many large optimization models contain large sums. But note that this class maintains the immutability property described above. This class shares an underlying list of arguments with other SumExpression objects. A particular object owns the first n arguments in the shared list, but different objects may have different values of n.

This class acts like a normal immutable expression class, and the API described above works normally. But direct access to the shared list could have unexpected results.

Mutable Expressions

Finally, Pyomo includes several **mutable** expression classes that are private. These are not intended to be used by users, but they might be useful for developers in contexts where the developer can appropriately control how the classes are used. Specifically, immutability eliminates side-effects where changes to a sub-expression unexpectedly create changes to the expression tree. But within the context of model transformations, developers may be able to limit the use of expressions to avoid these side-effects. The following mutable private classes are available in Pyomo:

_MutableSumExpression
: This class is used in the nonlinear_expression context manager to efficiently combine sums of nonlinear terms.

_MutableLinearExpression
: This class is used in the linear_expression context manager to efficiently combine sums of linear terms.

Expression Semantics

Pyomo clear semantics regarding what is considered a valid leaf and interior node.

The following classes are valid interior nodes:

- Subclasses of ExpressionBase
- Classes that that are *duck typed* to match the API of the ExpressionBase class. For example, the named expression class Expression.

The following classes are valid leaf nodes:

- Members of nonpyomo_leaf_types, which includes standard numeric data types like int, float and long, as well as numeric data types defined by *numpy* and other commonly used packages. This set also includes *NonNumericValue*, which is used to wrap non-numeric arguments to the ExternalFunctionExpression class.
- Parameter component classes like *ScalarParam* and _ParamData, which arise in expression trees when the parameters are declared as mutable. (Immutable parameters are identified when generating expressions, and they are replaced with their associated numeric value.)
- Variable component classes like *ScalarVar* and _GeneralVarData, which often arise in expression trees. <pyomo.core.expr.pyomo5_variable_types>`.

Note

In some contexts the `LinearExpression` class can be treated as an interior node, and sometimes it can be treated as a leaf. This expression object does not have any child arguments, so `nargs()` is zero. But this expression references variables and parameters in a linear expression, so in that sense it does not represent a leaf node in the tree.

Context Managers

Pyomo defines several context managers that can be used to declare the form of expressions, and to define a mutable expression object that efficiently manages sums.

The `linear_expression` object is a context manager that can be used to declare a linear sum. For example, consider the following two loops:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))

s = 0
for i in range(5):
    s += M.x[i]

with pyo.linear_expression() as e:
    for i in range(5):
        e += M.x[i]
```

The first apparent difference in these loops is that the value of `s` is explicitly initialized while `e` is initialized when the context manager is entered. However, a more fundamental difference is that the expression representation for `s` differs from `e`. Each term added to `s` results in a new, immutable expression. By contrast, the context manager creates a mutable expression representation for `e`. This difference allows for both (a) a more efficient processing of each sum, and (b) a more compact representation for the expression.

The difference between `linear_expression` and `nonlinear_expression` is the underlying representation that each supports. Note that both of these are instances of context manager classes. In singled-threaded applications, these objects can be safely used to construct different expressions with different context declarations.

Finally, note that these context managers can be passed into the `start` method for the *`quicksum`* function. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))
M.y = pyo.Var(range(5))

with pyo.linear_expression() as e:
    pyo.quicksum((M.x[i] for i in M.x), start=e)
    pyo.quicksum((M.y[i] for i in M.y), start=e)
```

This sum contains terms for `M.x[i]` and `M.y[i]`. The syntax in this example is not intuitive because the sum is being stored in `e`.

Note

We do not generally expect users or developers to use these context managers. They are used by the *`quicksum`* and *`sum_product`* functions to accelerate expression generation, and there are few cases where the direct use of these context managers would provide additional utility to users and developers.

Managing Expressions

Creating a String Representation of an Expression

There are several ways that string representations can be created from an expression, but the `expression_to_string` function provides the most flexible mechanism for generating a string representation. The options to this function control distinct aspects of the string representation.

Algebraic vs. Nested Functional Form

The default string representation is an algebraic form, which closely mimics the Python operations used to construct an expression. The `verbose` flag can be set to `True` to generate a string representation that is a nested functional form. For example:

```
import pyomo.core.expr as EXPR

M = pyo.ConcreteModel()
M.x = pyo.Var()

e = pyo.sin(M.x) + 2 * M.x

# sin(x) + 2*x
print(EXPR.expression_to_string(e))

# sum(sin(x), prod(2, x))
print(EXPR.expression_to_string(e, verbose=True))
```

Labeler and Symbol Map

The string representation used for variables in expression can be customized to define different label formats. If the `labeler` option is specified, then this function (or class functor) is used to generate a string label used to represent the variable. Pyomo defines a variety of labelers in the *pyomo.core.base.label* module. For example, the `NumericLabeler` defines a functor that can be used to sequentially generate simple labels with a prefix followed by the variable count:

```
import pyomo.core.expr as EXPR

M = pyo.ConcreteModel()
M.x = pyo.Var()
M.y = pyo.Var()

e = pyo.sin(M.x) + 2 * M.y

# sin(x1) + 2*x2
print(EXPR.expression_to_string(e, labeler=pyo.NumericLabeler('x')))
```

The `smap` option is used to specify a symbol map object (*`SymbolMap`*), which caches the variable label data. This option is normally specified in contexts where the string representations for many expressions are being generated. In that context, a symbol map ensures that variables in different expressions have a consistent label in their associated string representations.

Other Ways to Generate String Representations

There are two other standard ways to generate string representations:

- Call the `__str__()` magic method (e.g. using the Python `str()` function. This calls `expression_to_string`, using the default values for all arguments.
- Call the `to_string()` method on the `ExpressionBase` class. This calls `expression_to_string` and accepts the same arguments.

Evaluating Expressions

Expressions can be evaluated when all variables and parameters in the expression have a value. The `value` function can be used to walk the expression tree and compute the value of an expression. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var()
M.x.value = math.pi / 2.0
val = pyo.value(M.x)
assert isclose(val, math.pi / 2.0)
```

Additionally, expressions define the `__call__()` method, so the following is another way to compute the value of an expression:

```
val = M.x()
assert isclose(val, math.pi / 2.0)
```

If a parameter or variable is undefined, then the `value` function and `__call__()` method will raise an exception. This exception can be suppressed using the `exception` option. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var()
val = pyo.value(M.x, exception=False)
assert val is None
```

This option is useful in contexts where adding a try block is inconvenient in your modeling script.

> **Note**
>
> Both the `value` function and `__call__()` method call the `evaluate_expression` function. In practice, this function will be slightly faster, but the difference is only meaningful when expressions are evaluated many times.

Identifying Components and Variables

Expression transformations sometimes need to find all nodes in an expression tree that are of a given type. Pyomo contains two utility functions that support this functionality. First, the `identify_components` function is a generator function that walks the expression tree and yields all nodes whose type is in a specified set of node types. For example:

```
import pyomo.core.expr as EXPR

M = pyo.ConcreteModel()
M.x = pyo.Var()
M.p = pyo.Param(mutable=True)

```

(continues on next page)

(continued from previous page)

```
e = M.p + M.x
s = set([type(M.p)])
assert list(EXPR.identify_components(e, s)) == [M.p]
```

The `identify_variables` function is a generator function that yields all nodes that are variables. Pyomo uses several different classes to represent variables, but this set of variable types does not need to be specified by the user. However, the `include_fixed` flag can be specified to omit fixed variables. For example:

```
import pyomo.core.expr as EXPR

M = pyo.ConcreteModel()
M.x = pyo.Var()
M.y = pyo.Var()

e = M.x + M.y
M.y.value = 1
M.y.fixed = True

assert set(id(v) for v in EXPR.identify_variables(e)) == set([id(M.x), id(M.y)])
assert set(id(v) for v in EXPR.identify_variables(e, include_fixed=False)) == set(
    [id(M.x)]
)
```

Walking an Expression Tree with a Visitor Class

Many of the utility functions defined above are implemented by walking an expression tree and performing an operation at nodes in the tree. For example, evaluating an expression is performed using a post-order depth-first search process where the value of a node is computed using the values of its children.

Walking an expression tree can be tricky, and the code requires intimate knowledge of the design of the expression system. Pyomo includes several classes that define visitor patterns for walking expression tree:

`StreamBasedExpressionVisitor`
: The most general and extensible visitor class. This visitor implements an event-based approach for walking the tree inspired by the `expat` library for processing XML files. The visitor has seven event callbacks that users can hook into, providing very fine-grained control over the expression walker.

`ExpressionValueVisitor`
: When the `visitor()` method is called on each node in the tree, the *values* of its children have been computed. The *value* of the node is returned from `visitor()`.

`ExpressionReplacementVisitor`
: When the `visitor()` method is called on each node in the tree, it may clone or otherwise replace the node using objects for its children (which themselves may be clones or replacements from the original child objects). The new node object is returned from `visitor()`.

These classes define a variety of suitable tree search methods:

- `StreamBasedExpressionVisitor`
  - `walk_expression`: depth-first traversal of the expression tree.
- `ExpressionReplacementVisitor`
  - `walk_expression`: depth-first traversal of the expression tree.
- `ExpressionValueVisitor`

- dfs_postorder_stack: postorder depth-first search using a nonrecursive stack

To implement a visitor object, a user needs to provide specializations for specific events. For legacy visitors based on the PyUtilib visitor pattern (e.g., ExpressionValueVisitor), one must create a subclass and override at least one of the following:

visit()
: Defines the operation that is performed when a node is visited. In the ExpressionValueVisitor and ExpressionReplacementVisitor visitor classes, this method returns a value that is used by its parent node.

visiting_potential_leaf()
: Checks if the search should terminate with this node. If no, then this method returns the tuple (False, None). If yes, then this method returns (False, value), where *value* is computed by this method.

finalize()
: This method defines the final value that is returned from the visitor. This is not normally redefined.

For modern visitors based on the StreamBasedExpressionVisitor, one can either define a subclass, pass the callbacks to an instance of the base class, or assign the callbacks as attributes on an instance of the base class. The StreamBasedExpressionVisitor provides seven callbacks, which are documented in the class documentation.

Detailed documentation of the APIs for these methods is provided with the class documentation for these visitors.

StreamBasedExpressionVisitor Example

In this example, we describe an visitor class that counts the number of nodes in an expression (including leaf nodes). Consider the following class:

```
import pyomo.core.expr as EXPR


class SizeofVisitor(EXPR.StreamBasedExpressionVisitor):
    def initializeWalker(self, expr):
        self.counter = 0
        return True, expr

    def exitNode(self, node, data):
        self.counter += 1

    def finalizeResult(self, result):
        return self.counter
```

The initializeWalker() method creates a counter, and the exitNode() method increments this counter for every node that is visited. The finalizeResult() method returns the value of this counter after the tree has been walked. The following function illustrates this use of this visitor class:

```
def sizeof_expression(expr):
    #
    # Create the visitor object
    #
    visitor = SizeofVisitor()
    #
    # Compute the value using the :func:`walk_expression` search method.
    #
    return visitor.walk_expression(expr)
```

ExpressionValueVisitor Example

In this example, we describe an visitor class that clones the expression tree (including leaf nodes). Consider the following class:

```
import pyomo.core.expr as EXPR


class CloneVisitor(EXPR.ExpressionValueVisitor):
    def __init__(self):
        self.memo = {'__block_scope__': {id(None): False}}

    def visit(self, node, values):
        #
        # Clone the interior node
        #
        return node.create_node_with_local_data(values)

    def visiting_potential_leaf(self, node):
        #
        # Clone leaf nodes in the expression tree
        #
        if node.__class__ in pyo.native_numeric_types or not node.is_expression_type():
            return True, copy.deepcopy(node, self.memo)

        return False, None
```

The `visit()` method creates a new expression node with children specified by `values`. The `visiting_potential_leaf()` method performs a `deepcopy()` on leaf nodes, which are native Python types or non-expression objects.

```
def clone_expression(expr):
    #
    # Create the visitor object
    #
    visitor = CloneVisitor()
    #
    # Clone the expression using the :func:`dfs_postorder_stack`
    # search method.
    #
    return visitor.dfs_postorder_stack(expr)
```

ExpressionReplacementVisitor Example

In this example, we describe an visitor class that replaces variables with scaled variables, using a mutable parameter that can be modified later. the following class:

```
import pyomo.core.expr as EXPR


class ScalingVisitor(EXPR.ExpressionReplacementVisitor):
    def __init__(self, scale):
        super(ScalingVisitor, self).__init__()
        self.scale = scale
```

(continues on next page)

(continued from previous page)

```
    def beforeChild(self, node, child, child_idx):
        #
        # Native numeric types are terminal nodes; this also catches all
        # nodes that do not conform to the ExpressionBase API (i.e.,
        # define is_variable_type)
        #
        if child.__class__ in pyo.native_numeric_types:
            return False, child
        #
        # Replace leaf variables with scaled variables
        #
        if child.is_variable_type():
            return False, self.scale[id(child)] * child
        #
        # Everything else can be processed normally
        #
        return True, None
```

No other method need to be defined. The `beforeChild()` method identifies variable nodes and returns a product expression that contains a mutable parameter.

```
def scale_expression(expr, scale):
    #
    # Create the visitor object
    #
    visitor = ScalingVisitor(scale)
    #
    # Scale the expression using the :func:`dfs_postorder_stack`
    # search method.
    #
    return visitor.walk_expression(expr)
```

The `scale_expression()` function is called with an expression and a dictionary, `scale`, that maps variable ID to model parameter. For example:

```
M = pyo.ConcreteModel()
M.x = pyo.Var(range(5))
M.p = pyo.Param(range(5), mutable=True)

scale = {}
for i in M.x:
    scale[id(M.x[i])] = M.p[i]

e = pyo.quicksum(M.x[i] for i in M.x)
f = scale_expression(e, scale)

# p[0]*x[0] + p[1]*x[1] + p[2]*x[2] + p[3]*x[3] + p[4]*x[4]
print(f)
```

3.1.4 Model Transformations

TODO

3.2 Modeling in Pyomo

3.2.1 Math Programming

Sets

Declaration

Sets can be declared using instances of the `Set` and `RangeSet` classes or by assigning set expressions. The simplest set declaration creates a set and postpones creation of its members:

```
model.A = pyo.Set()
```

The `Set` class takes optional arguments such as:

dimen
: Dimension of the members of the set; `None` for "jagged" sets (where members do not have a uniform length).

doc
: String describing the set

filter
: A Boolean function used during construction to indicate if a potential new member should be assigned to the set

initialize
: An iterable containing the initial members of the Set, or function that returns an iterable of the initial members the set.

ordered
: A Boolean indicator that the set is ordered; the default is `True` (Set is ordered by insertion order)

validate
: A Boolean function that validates new member data

within
: Set used for validation; it is a super-set of the set being declared.

In general, Pyomo attempts to infer the "dimensionality" of Set components (that is, the number of apparent indices) when they are constructed. However, there are situations where Pyomo either cannot detect a dimensionality (e.g., a `Set` that was not initialized with any members), or you the user may want to assert the dimensionality of the set. This can be accomplished through the `dimen` keyword. For example, to create a set whose members will be tuples with two items, one could write:

```
model.B = pyo.Set(dimen=2)
```

To create a set of all the numbers in set `model.A` doubled, one could use

```
def DoubleA_init(model):
    return (i*2 for i in model.A)
model.C = pyo.Set(initialize=DoubleA_init)
```

As an aside we note that as always in Python, there are lot of ways to accomplish the same thing. Also, note that this will generate an error if `model.A` contains elements for which multiplication times two is not defined.

The `initialize` option can accept any Python iterable, including a `set`, `list`, or `tuple`. This data may be returned from a function or specified directly as in

```
model.D = pyo.Set(initialize=['red', 'green', 'blue'])
```

The `initialize` option can also specify either a generator or a function to specify the Set members. In the case of a generator, all data yielded by the generator will become the initial set members:

```
def X_init(m):
    for i in range(10):
        yield 2*i+1
model.X = pyo.Set(initialize=X_init)
```

For initialization functions, Pyomo supports two signatures. In the first, the function returns an iterable (`set`, `list`, or `tuple`) containing the data with which to initialize the Set:

```
def Y_init(m):
    return [2*i+1 for i in range(10)]
model.Y = pyo.Set(initialize=Y_init)
```

In the second signature, the function is called for each element, passing the element number in as an extra argument. This is repeated until the function returns the special value `Set.End`:

```
def Z_init(model, i):
    if i > 10:
        return pyo.Set.End
    return 2*i+1
model.Z = pyo.Set(initialize=Z_init)
```

Note that the element number starts with 1 and not 0:

```
>>> model.X.pprint()
X : Size=1, Index=None, Ordered=Insertion
    Key  : Dimen : Domain : Size : Members
    None :     1 :    Any :   10 : {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
>>> model.Y.pprint()
Y : Size=1, Index=None, Ordered=Insertion
    Key  : Dimen : Domain : Size : Members
    None :     1 :    Any :   10 : {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
>>> model.Z.pprint()
Z : Size=1, Index=None, Ordered=Insertion
    Key  : Dimen : Domain : Size : Members
    None :     1 :    Any :   10 : {3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
```

Additional information about iterators for set initialization is in the [PyomoBookIII] book.

Note

For Abstract models, data specified in an input file or through the `data` argument to `AbstractModel.create_instance()` will override the data specified by the initialize options.

If sets are given as arguments to `Set` without keywords, they are interpreted as indexes for an array of sets. For example, to create an array of sets that is indexed by the members of the set `model.A`, use:

```
model.E = pyo.Set(model.A)
```

Arguments can be combined. For example, to create an array of sets, indexed by set `model.A` where each set contains three dimensional members, use:

```
model.F = pyo.Set(model.A, dimen=3)
```

The `initialize` option can be used to create a set that contains a sequence of numbers, but the `RangeSet` class provides a concise mechanism for simple sequences. This class takes as its arguments a start value, a final value, and a step size. If the `RangeSet` has only a single argument, then that value defines the final value in the sequence; the first value and step size default to one. If two values given, they are the first and last value in the sequence and the step size defaults to one. For example, the following declaration creates a set with the numbers 1.5, 5 and 8.5:

```
model.G = pyo.RangeSet(1.5, 10, 3.5)
```

Operations

Sets may also be created by storing the result of *set operations* using other Pyomo sets. Pyomo supports set operations including union, intersection, difference, and symmetric difference:

```
model.I = model.A | model.D # union
model.J = model.A & model.D # intersection
model.K = model.A - model.D # difference
model.L = model.A ^ model.D # exclusive-or
```

For example, the cross-product operator is the asterisk (*). To define a new set `M` that is the cross product of sets `B` and `C`, one could use

```
model.M = model.B * model.C
```

This creates a *virtual* set that holds references to the original sets, so any updates to the original sets (`B` and `C`) will be reflected in the new set (`M`). In contrast, you can also create a *concrete* set, which directly stores the values of the cross product at the time of creation and will *not* reflect subsequent changes in the original sets with:

```
model.M_concrete = pyo.Set(initialize=model.B * model.C)
```

Finally, you can indicate that the members of a set are restricted to be in the cross product of two other sets, one can use the `within` keyword:

```
model.N = pyo.Set(within=model.B * model.C)
```

Predefined Virtual Sets

For use in specifying domains for sets, parameters and variables, Pyomo provides the following pre-defined virtual sets:

- `Any` = all possible values
- `Reals` = floating point values
- `PositiveReals` = strictly positive floating point values
- `NonPositiveReals` = non-positive floating point values
- `NegativeReals` = strictly negative floating point values
- `NonNegativeReals` = non-negative floating point values
- `PercentFraction` = floating point values in the interval [0,1]

- `UnitInterval` = alias for PercentFraction
- `Integers` = integer values
- `PositiveIntegers` = positive integer values
- `NonPositiveIntegers` = non-positive integer values
- `NegativeIntegers` = negative integer values
- `NonNegativeIntegers` = non-negative integer values
- `Boolean` = Boolean values, which can be represented as False/True, 0/1, 'False'/'True' and 'F'/'T'
- `Binary` = the integers {0, 1}

For example, if the set `model.O` is declared to be within the virtual set `NegativeIntegers` then an attempt to add anything other than a negative integer will result in an error. Here is the declaration:

```
model.O = pyo.Set(within=pyo.NegativeIntegers)
```

Sparse Index Sets

Sets provide indexes for parameters, variables and other sets. Index set issues are important for modelers in part because of efficiency considerations, but primarily because the right choice of index sets can result in very natural formulations that are conducive to understanding and maintenance. Pyomo leverages Python to provide a rich collection of options for index set creation and use.

The choice of how to represent indexes often depends on the application and the nature of the instance data that are expected. To illustrate some of the options and issues, we will consider problems involving networks. In many network applications, it is useful to declare a set of nodes, such as

```
model.Nodes = pyo.Set()
```

and then a set of arcs can be created with reference to the nodes.

Consider the following simple version of minimum cost flow problem:

$$\begin{array}{lll} \text{minimize} & \sum_{a \in \mathcal{A}} c_a x_a & \\ \text{subject to:} & S_n + \sum_{(i,n) \in \mathcal{A}} x_{(i,n)} & \\ & -D_n - \sum_{(n,j) \in \mathcal{A}} x_{(n,j)} & n \in \mathcal{N} \\ & x_a \geq 0, & a \in \mathcal{A} \end{array}$$

where

- Set: Nodes $\equiv \mathcal{N}$
- Set: Arcs $\equiv \mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$
- Var: Flow on arc $(i, j) \equiv x_{i,j},\ (i, j) \in \mathcal{A}$
- Param: Flow Cost on arc $(i, j) \equiv c_{i,j},\ (i, j) \in \mathcal{A}$
- Param: Demand at node latexmath:$i \equiv D_i,\ i \in \mathcal{N}$
- Param: Supply at node latexmath:$i \equiv S_i,\ i \in \mathcal{N}$

In the simplest case, the arcs can just be the cross product of the nodes, which is accomplished by the definition

```
model.Arcs = model.Nodes*model.Nodes
```

that creates a set with two dimensional members. For applications where all nodes are always connected to all other nodes this may suffice. However, issues can arise when the network is not fully dense. For example, the burden of avoiding flow on arcs that do not exist falls on the data file where high-enough costs must be provided for those arcs. Such a scheme is not very elegant or robust.

For many network flow applications, it might be better to declare the arcs using

```
model.Arcs = pyo.Set(dimen=2)
```

or

```
model.Arcs = pyo.Set(within=model.Nodes*model.Nodes)
```

where the difference is that the first version will provide error checking as data is assigned to the set elements. This would enable specification of a sparse network in a natural way. But this results in a need to change the `FlowBalance` constraint because as it was written in the simple example, it sums over the entire set of nodes for each node. One way to remedy this is to sum only over the members of the set `model.arcs` as in

```
def FlowBalance_rule(m, node):
    return m.Supply[node] \
        + sum(m.Flow[i, node] for i in m.Nodes if (i,node) in m.Arcs) \
        - m.Demand[node] \
        - sum(m.Flow[node, j] for j in m.Nodes if (j,node) in m.Arcs) \
        == 0
```

This will be OK unless the number of nodes becomes very large for a sparse network, then the time to generate this constraint might become an issue (admittely, only for very large networks, but such networks do exist).

Another method, which comes in handy in many network applications, is to have a set for each node that contain the nodes at the other end of arcs going to the node at hand and another set giving the nodes on out-going arcs. If these sets are called `model.NodesIn` and `model.NodesOut` respectively, then the flow balance rule can be re-written as

```
def FlowBalance_rule(m, node):
    return m.Supply[node] \
        + sum(m.Flow[i, node] for i in m.NodesIn[node]) \
        - m.Demand[node] \
        - sum(m.Flow[node, j] for j in m.NodesOut[node]) \
        == 0
```

The data for `NodesIn` and `NodesOut` could be added to the input file, and this may be the most efficient option.

For all but the largest networks, rather than reading `Arcs`, `NodesIn` and `NodesOut` from a data file, it might be more elegant to read only `Arcs` from a data file and declare `model.NodesIn` with an `initialize` option specifying the creation as follows:

```
def NodesIn_init(m, node):
    for i, j in m.Arcs:
        if j == node:
            yield i
model.NodesIn = pyo.Set(model.Nodes, initialize=NodesIn_init)
```

with a similar definition for `model.NodesOut`. This code creates a list of sets for `NodesIn`, one set of nodes for each node. The full model is:

```
import pyomo.environ as pyo

```

(continues on next page)

(continued from previous page)

```
model = pyo.AbstractModel()

model.Nodes = pyo.Set()
model.Arcs = pyo.Set(dimen=2)

def NodesOut_init(m, node):
    for i, j in m.Arcs:
        if i == node:
            yield j
model.NodesOut = pyo.Set(model.Nodes, initialize=NodesOut_init)

def NodesIn_init(m, node):
    for i, j in m.Arcs:
        if j == node:
            yield i
model.NodesIn = pyo.Set(model.Nodes, initialize=NodesIn_init)

model.Flow = pyo.Var(model.Arcs, domain=pyo.NonNegativeReals)
model.FlowCost = pyo.Param(model.Arcs)

model.Demand = pyo.Param(model.Nodes)
model.Supply = pyo.Param(model.Nodes)

def Obj_rule(m):
    return pyo.summation(m.FlowCost, m.Flow)
model.Obj = pyo.Objective(rule=Obj_rule, sense=pyo.minimize)

def FlowBalance_rule(m, node):
    return m.Supply[node] \
        + sum(m.Flow[i, node] for i in m.NodesIn[node]) \
        - m.Demand[node] \
        - sum(m.Flow[node, j] for j in m.NodesOut[node]) \
        == 0
model.FlowBalance = pyo.Constraint(model.Nodes, rule=FlowBalance_rule)
```

for this model, a toy data file (in AMPL “`.dat`” format) would be:

```
set Nodes := CityA CityB CityC ;

set Arcs :=
CityA CityB
CityA CityC
CityC CityB
;

param : FlowCost :=
CityA CityB 1.4
CityA CityC 2.7
CityC CityB 1.6
 ;

param Demand :=
```

(continues on next page)

(continued from previous page)

```
CityA 0
CityB 1
CityC 1
;

param Supply :=
CityA 2
CityB 0
CityC 0
;
```

A similar result can be accomplished more efficiently (because we only iterate over the Arcs twice) using initialization functions that accept only a model block and return a `dict` with all the information needed for the indexed set:

```
def NodesIn_init(m):
    # Create a dict to show NodesIn list for every node
    d = {i: [] for i in m.Nodes}
    # loop over the arcs and record the end points
    for i, j in model.Arcs:
        d[j].append(i)
    return d
model.NodesIn = pyo.Set(model.Nodes, initialize=NodesIn_init)

def NodesOut_init(m):
    d = {i: [] for i in m.Nodes}
    for i, j in model.Arcs:
        d[i].append(j)
    return d
model.NodesOut = pyo.Set(model.Nodes, initialize=NodesOut_init)
```

Alternatively, this can also be done even more efficiently, and perhaps more clearly, outside the context of Set initialization. For concrete models, scripts can explicitly add elements to the Sets after declaration:

```
model.NodesIn = pyo.Set(model.Nodes, within=model.Nodes)
model.NodesOut = pyo.Set(model.Nodes, within=model.Nodes)

# loop over the arcs and record the end points
for i, j in model.Arcs:
    model.NodesIn[j].add(i)
    model.NodesOut[i].add(j)
```

For abstract models, that action must be deferred to instance construction time using a `BuildAction` (for more information, see *BuildAction and BuildCheck*):

```
model.NodesIn = pyo.Set(model.Nodes, within=model.Nodes)
model.NodesOut = pyo.Set(model.Nodes, within=model.Nodes)

def Populate_In_and_Out(model):
    # loop over the arcs and record the end points
    for i, j in model.Arcs:
        model.NodesIn[j].add(i)
        model.NodesOut[i].add(j)
```

(continues on next page)

(continued from previous page)

```
model.In_n_Out = pyo.BuildAction(rule=Populate_In_and_Out)
```

Sparse Index Sets Example

One may want to have a constraint that holds

$$\forall\, i \in I, k \in K, v \in V_k$$

There are many ways to accomplish this, but one good way is to create a set of tuples composed of all `model.k, model.V[k]` pairs. This can be done as follows:

```
def kv_init(m):
    return ((k,v) for k in m.K for v in m.V[k])
model.KV = pyo.Set(dimen=2, initialize=kv_init)
```

We can now create the constraint $x_{i,k,v} \leq a_{i,k} y_i \;\forall\, i \in I, k \in K, v \in V_k$ with:

```
model.a = pyo.Param(model.I, model.K, default=1)

model.y = pyo.Var(model.I)
model.x = pyo.Var(model.I, model.KV)

def c1_rule(m, i, k, v):
    return m.x[i,k,v] <= m.a[i,k]*m.y[i]
model.c1 = pyo.Constraint(model.I, model.KV, rule=c1_rule)
```

Parameters

The word "parameters" is used in many settings. In Pyomo, a `Param` represents the fixed data of an optimization model. Unlike variables (`Var`), which the solver determines, parameters are inputs that define

the specific instance of the problem you are solving.

Common examples of parameters include costs, demands, capacities, or physical constants. While you could use standard Python variables to store these values, using Pyomo `Param` components offers several advantages:

- **Index Management:** Params can be indexed by Pyomo `Set` objects, ensuring consistency between your data and the model structure. In a `ConcreteModel`, they can also be indexed by standard Python iterables like lists, tuples, or ranges.
- **Validation:** You can define rules to ensure that the data provided (e.g., from an external file) is valid before solving.
- **Symbolic Representation:** In large or complex models, using Params allows Pyomo to maintain the structure of the model separately from the specific values.

Note

When working with a `ConcreteModel`, many modelers choose to use standard Python variables, lists, or dictionaries to store their data instead of Pyomo `Param` objects. This is a common and valid practice.

However, you must use a Pyomo `Param` if:

- You are using an `AbstractModel` (which requires components to be declared before data is loaded).

- You need a **mutable** parameter to change values and re-solve the model without the overhead of rebuilding it from scratch.
- You want to leverage Pyomo's built-in data validation and index-checking capabilities.

Declaration and Options

Parameters are declared as instances of the `Param` class. They can be scalar (single value) or indexed by one or more sets (Pyomo `Set` or other iterables). For example:

```
model.A = pyo.RangeSet(1,3)
model.B = pyo.Set(initialize=['dog', 'cat'])
# Scalar parameter
model.rho = pyo.Param(initialize=0.5)
# Indexed parameter (by Set)
model.P = pyo.Param(model.A, model.B)
# Indexed parameter (by standard list)
model.Q = pyo.Param(['a', 'b', 'c'], initialize={'a': 1, 'b': 2, 'c': 3})
```

If there are indexes for a `Param`, they are provided as the first positional arguments and do not have a keyword label. In addition to these optional indexes, `Param` takes the following keyword arguments:

- `default` = The parameter value used if no other value is specified for an index.
- `doc` = A string describing the parameter.
- `initialize` = A function, dictionary, or other Python object used to provide initial data.
- `mutable` = Boolean indicating if values can be changed after construction (see below).
- `validate` = A callback function to verify data integrity.
- `within` = A set (e.g., `NonNegativeReals`) used for domain validation.

Basic Initialization

There are many ways to provide data to a `Param`. For example, given `model.A` with values `{1, 2, 3}`, here are two ways to create a diagonal matrix:

```
v={}
v[1,1] = 9
v[2,2] = 16
v[3,3] = 25
model.S1 = pyo.Param(model.A, model.A, initialize=v, default=0)
```

You can also use an initialization function that Pyomo calls for each index:

```
def s_init(model, i, j):
    if i == j:
        return i*i
    else:
        return 0.0
model.S2 = pyo.Param(model.A, model.A, initialize=s_init)
```

Note

In an `AbstractModel`, data specified in an external input file (e.g., a `.dat` file) will override the data specified by the `initialize` option.

Validation

Parameter values can be checked by a validation function. In the following example, we ensure every value of `model.T` is greater than 3.14159:

```
t_data = {1: 10, 2: 3, 3: 20}

def t_validate(model, v, i):
    return v > 3.14159

model.T = pyo.Param(model.A, validate=t_validate, initialize=t_data)
```

This example will produce the following error:

```
Traceback (most recent call last):
  ...
ValueError: Invalid parameter value: T[2] = '3', value type=<class 'int'>.
    Value failed parameter validation rule
```

Performance vs. Flexibility: Mutable Parameters

By default, Pyomo parameters are **immutable** (`mutable=False`). This choice is driven by performance:

- **Immutable (Default):** Pyomo "pre-computes" these values into the algebraic expressions during model construction. This results in faster model generation and significantly lower memory usage, especially for large models. Key advantages include:
  - **Memory Efficiency:** For indexed parameters, Pyomo avoids creating individual component data objects, significantly reducing memory overhead.
  - **Expression Speed:** Values are injected as constants directly into the expression tree. This allows Pyomo to optimize expression tree walking.
  - **Simplification:** Pyomo can simplify constant sub-expressions during model construction (e.g., `5 * model.p * model.q[i]` is simplified to a single float if `p` and `q` are immutable), further accelerating subsequent processing.
- **Mutable:** Pyomo maintains the parameter as a symbolic object within expressions. This allows you to change the value and re-solve without rebuilding the entire model, but it adds computational overhead.

It is important to note that even immutable `Param` objects carry some overhead. For the fastest possible model instantiation in a `ConcreteModel`, using native Python data structures (like dictionaries or lists) to provide values directly into expressions is usually faster than using `Param` components. However, as noted earlier, `Param` provides benefits like validation and the ability to update values if declared as mutable.

When to use Mutable

Use Immutable if:

- The data is static and never changes during the lifetime of the model.
- You want to maximize performance and minimize memory usage for large models.

Use Mutable if:

- You are running a loop (e.g., sensitivity analysis) where you change parameter values and re-solve.
- You want to update values frequently without the "re-construction" bottleneck.
- The parameter is part of a nonlinear expression that you need to update.
- You want named constants to be preserved in the Pyomo expressions (e.g., for documentation of debugging purposes)

Comparison: Param vs. Var

It is common to confuse mutable parameters with variables. The following table summarizes the key differences:

Feature	Param (Immutable)	Param (Mutable)	Var (fixed)	Var (free)
Can change after model construction?	No	Yes	Yes	Yes
Rebuilds model on change?	Yes (requires new Param)	No	No	No
Solver sees it as:	A constant number	A constant number	A constant number	An optimization variable

Note

Should I use a mutable Param or a fixed Var? While functionally similar, you should use a `Param` for data that defines the problem instance (like costs or demands) and a `Var` for the decisions the solver needs to make. Use *fix()* on a `Var` when you want to temporarily hold a decision constant, and use a mutable `Param` when you need to update input data for sensitivity analysis

or iterative algorithms.

Variables

Variables are intended to ultimately be given values by an optimization package. They are declared and optionally bounded, given initial values, and documented using the Pyomo `Var` function. If index sets are given as arguments to this function they are used to index the variable. Other optional directives include:

- bounds = A function (or Python object) that gives a (lower,upper) bound pair for the variable
- domain = A set that is a super-set of the values the variable can take on.
- initialize = A function (or Python object) that gives a starting value for the variable; this is particularly important for non-linear models
- within = (synonym for `domain`)

The following code snippet illustrates some aspects of these options by declaring a *singleton* (i.e. unindexed) variable named `model.LumberJack` that will take on real values between zero and 6 and it initialized to be 1.5:

```
model.LumberJack = pyo.Var(within=pyo.NonNegativeReals, bounds=(0, 6), initialize=1.5)
```

Instead of the `initialize` option, initialization is sometimes done with a Python assignment statement as in

```
model.LumberJack = 1.5
```

For indexed variables, bounds and initial values are often specified by a rule (a Python function) that itself may make reference to parameters or other data. The formal arguments to these rules begins with the model followed by the indexes. This is illustrated in the following code snippet that makes use of Python dictionaries declared as lb and ub that are used by a function to provide bounds:

```
model.A = pyo.Set(initialize=['Scones', 'Tea'])
lb = {'Scones': 2, 'Tea': 4}
ub = {'Scones': 5, 'Tea': 7}


def fb(model, i):
    return (lb[i], ub[i])


model.PriceToCharge = pyo.Var(model.A, domain=pyo.PositiveIntegers, bounds=fb)
```

> **Note**
>
> Many of the pre-defined virtual sets that are used as domains imply bounds. A strong example is the set `Boolean` that implies bounds of zero and one.

Objectives

An objective is a function of variables that returns a value that an optimization package attempts to maximize or minimize. The `Objective` function in Pyomo declares an objective. Although other mechanisms are possible, this function is typically passed the name of another function that gives the expression. Here is a very simple version of such a function that assumes `model.x` has previously been declared as a `Var`:

```
>>> def ObjRule(model):
...     return 2*model.x[1] + 3*model.x[2]
>>> model.obj1 = pyo.Objective(rule=ObjRule)
```

It is more common for an objective function to refer to parameters as in this example that assumes that `model.p` has been declared as a `Param` and that `model.x` has been declared with the same index set, while `model.y` has been declared as a singleton:

```
>>> def ObjRule(model):
...     return pyo.summation(model.p, model.x) + model.y
>>> model.obj2 = pyo.Objective(rule=ObjRule, sense=pyo.maximize)
```

This example uses the `sense` option to specify maximization. The default sense is `minimize`.

Constraints

Most constraints are specified using equality or inequality expressions that are created using a rule, which is a Python function. For example, if the variable `model.x` has the indexes 'butter' and 'scones', then this constraint limits the sum over these indexes to be exactly three:

```
def teaOKrule(model):
    return model.x['butter'] + model.x['scones'] == 3


model.TeaConst = pyo.Constraint(rule=teaOKrule)
```

Instead of expressions involving equality (==) or inequalities (<= or >=), constraints can also be expressed using a 3-tuple if the form (lb, expr, ub) where lb and ub can be `None`, which is interpreted as lb <= expr <= ub. Variables can appear only in the middle expr. For example, the following two constraint declarations have the same meaning:

```
model.x = pyo.Var()


def aRule(model):
    return model.x >= 2


model.Boundx = pyo.Constraint(rule=aRule)


def bRule(model):
    return (2, model.x, None)


model.boundx = pyo.Constraint(rule=bRule)
```

For this simple example, it would also be possible to declare `model.x` with a `bounds` option to accomplish the same thing.

Constraints (and objectives) can be indexed by lists or sets. When the declaration contains lists or sets as arguments, the elements are iteratively passed to the rule function. If there is more than one, then the cross product is sent. For example the following constraint could be interpreted as placing a budget of i on the ith item to buy where the cost per item is given by the parameter `model.a`:

```
model.A = pyo.RangeSet(1, 10)
model.a = pyo.Param(model.A, within=pyo.PositiveReals)
model.ToBuy = pyo.Var(model.A)


def bud_rule(model, i):
    return model.a[i] * model.ToBuy[i] <= i


aBudget = pyo.Constraint(model.A, rule=bud_rule)
```

Note

Python and Pyomo are case sensitive so `model.a` is not the same as `model.A`.

Expressions

In this section, we use the word “expression” in two ways: first in the general sense of the word and second to describe a class of Pyomo objects that have the name `Expression` as described in the subsection on expression objects.

Rules to Generate Expressions

Both objectives and constraints make use of rules to generate expressions. These are Python functions that return the appropriate expression. These are first-class functions that can access global data as well as data passed in, including the model object.

Operations on model elements results in expressions, which seems natural in expressions like the constraints we have seen so far. It is also possible to build up expressions. The following example illustrates this, along with a reference to global Python data in the form of a Python variable called `switch`:

```
switch = 3

model.A = pyo.RangeSet(1, 10)
model.c = pyo.Param(model.A)
model.d = pyo.Param()
model.x = pyo.Var(model.A, domain=pyo.Boolean)


def pi_rule(model):
    accexpr = pyo.summation(model.c, model.x)
    if switch >= 2:
        accexpr = accexpr - model.d
    return accexpr >= 0.5


PieSlice = pyo.Constraint(rule=pi_rule)
```

In this example, the constraint that is generated depends on the value of the Python variable called `switch`. If the value is 2 or greater, then the constraint is `summation(model.c, model.x) - model.d >= 0.5`; otherwise, the `model.d` term is not present.

Warning

Because model elements result in expressions, not values, the following does not work as expected in an abstract model!

```
model.A = pyo.RangeSet(1, 10)
model.c = pyo.Param(model.A)
model.d = pyo.Param()
model.x = pyo.Var(model.A, domain=pyo.Boolean)


def pi_rule(model):
    accexpr = pyo.summation(model.c, model.x)
    if model.d >= 2:  # NOT in an abstract model!!
        accexpr = accexpr - model.d
    return accexpr >= 0.5


PieSlice = pyo.Constraint(rule=pi_rule)
```

The trouble is that `model.d >= 2` results in an expression, not its evaluated value. Instead use `if value(model.d) >= 2`

Note

Pyomo supports non-linear expressions and can call non-linear solvers such as Ipopt.

Piecewise Linear Expressions

Pyomo has facilities to add piecewise constraints of the form y=f(x) for a variety of forms of the function f.

The piecewise types other than SOS2, BIGM_SOS1, BIGM_BIN are implement as described in the paper [VAN10].

There are two basic forms for the declaration of the constraint:

```
# model.pwconst = Piecewise(indexes, yvar, xvar, **Keywords)
# model.pwconst = Piecewise(yvar,xvar,**Keywords)
```

where `pwconst` can be replaced by a name appropriate for the application. The choice depends on whether the x and y variables are indexed. If so, they must have the same index sets and these sets are give as the first arguments.

Keywords:

- **pw_pts={ },[ ],( )**

  A dictionary of lists (where keys are the index set) or a single list (for the non-indexed case or when an identical set of breakpoints is used across all indices) defining the set of domain breakpoints for the piecewise linear function.

  > **Note**
  >
  > pw_pts is always required. These give the breakpoints for the piecewise function and are expected to fully span the bounds for the independent variable(s).

- **pw_repn=<Option>**

  Indicates the type of piecewise representation to use. This can have a major impact on solver performance. Options: (Default "SOS2")

  – "SOS2" - Standard representation using sos2 constraints.

  – "BIGM_BIN" - BigM constraints with binary variables. The theoretically tightest M values are automatically determined.

  – "BIGM_SOS1" - BigM constraints with sos1 variables. The theoretically tightest M values are automatically determined.

  – "DCC" - Disaggregated convex combination model.

  – "DLOG" - Logarithmic disaggregated convex combination model.

  – "CC" - Convex combination model.

  – "LOG" - Logarithmic branching convex combination.

  – "MC" - Multiple choice model.

  – "INC" - Incremental (delta) method.

  > **Note**
  >
  > Step functions are supported for all but the two BIGM options. Refer to the 'force_pw' option.

- **pw_constr_type= <Option>**

  Indicates the bound type of the piecewise function. Options:

  - “UB” - y variable is bounded above by piecewise function.
  - “LB” - y variable is bounded below by piecewise function.
  - “EQ” - y variable is equal to the piecewise function.

- **f_rule=f(model,i,j,. . . ,x), { }, [ ], ( )**

  An object that returns a numeric value that is the range value corresponding to each piecewise domain point. For functions, the first argument must be a Pyomo model. The last argument is the domain value at which the function evaluates (Not a Pyomo `Var`). Intermediate arguments are the corresponding indices of the Piecewise component (if any). Otherwise, the object can be a dictionary of lists/tuples (with keys the same as the indexing set) or a singe list/tuple (when no indexing set is used or when all indices use an identical piecewise function). Examples:

```
# A function that changes with index
def f(model, j, x):
    if j == 2:
        return x**2 + 1.0
    else:
        return x**2 + 5.0


# A nonlinear function
f = lambda model, x: pyo.exp(x) + pyo.value(model.p)

# A step function
f = [0, 0, 1, 1, 2, 2]
```

- **force_pw=True/False**

  Using the given function rule and pw_pts, a check for convexity/concavity is implemented. If (1) the function is convex and the piecewise constraints are lower bounds or if (2) the function is concave and the piecewise constraints are upper bounds then the piecewise constraints will be substituted for linear constraints. Setting ‘force_pw=True’ will force the use of the original piecewise constraints even when one of these two cases applies.

- **warning_tol=<float>**

  To aid in debugging, a warning is printed when consecutive slopes of piecewise segments are within <warning_tol> of each other. Default=1e-8

- **warn_domain_coverage=True/False**

  Print a warning when the feasible region of the domain variable is not completely covered by the piecewise breakpoints. Default=True

- **unbounded_domain_var=True/False**

  Allow an unbounded or partially bounded Pyomo Var to be used as the domain variable. Default=False

Note

This does not imply unbounded piecewise segments will be constructed. The outermost piecewise breakpoints will bound the domain variable at each index. However, the Var attributes .lb and .ub will not be modified.

Here is an example of an assignment to a Python dictionary variable that has keywords for a picewise constraint:

```
kwds = {
    'pw_constr_type': 'EQ',
    'pw_repn': 'SOS2',
    'sense': pyo.maximize,
    'force_pw': True,
}
```

Here is a simple example based on the example given earlier in *Symbolic Index Sets*. In this new example, the objective function is the sum of c times x to the fourth. In this example, the keywords are passed directly to the `Piecewise` function without being assigned to a dictionary variable. The upper bound on the x variables was chosen whimsically just to make the example. The important thing to note is that variables that are going to appear as the independent variable in a piecewise constraint must have bounds.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

# abstract2piece.py
# Similar to abstract2.py, but the objective is now c times x to the fourth power

import pyomo.environ as pyo

model = pyo.AbstractModel()

model.I = pyo.Set()
model.J = pyo.Set()

Topx = 6.1  # range of x variables

model.a = pyo.Param(model.I, model.J)
model.b = pyo.Param(model.I)
model.c = pyo.Param(model.J)

# the next line declares a variable indexed by the set J
model.x = pyo.Var(model.J, domain=pyo.NonNegativeReals, bounds=(0, Topx))
model.y = pyo.Var(model.J, domain=pyo.NonNegativeReals)

# to avoid warnings, we set breakpoints at or beyond the bounds
PieceCnt = 100
bpts = []
for i in range(PieceCnt + 2):
    bpts.append(float((i * Topx) / PieceCnt))


def f4(model, j, xp):
    # we not need j, but it is passed as the index for the constraint
    return xp**4
```

(continues on next page)

(continued from previous page)

```
model.ComputeObj = pyo.Piecewise(
    model.J, model.y, model.x, pw_pts=bpts, pw_constr_type='EQ', f_rule=f4
)


def obj_expression(model):
    return pyo.summation(model.c, model.y)


model.OBJ = pyo.Objective(rule=obj_expression)


def ax_constraint_rule(model, i):
    # return the expression for the constraint for i
    return sum(model.a[i, j] * model.x[j] for j in model.J) >= model.b[i]


# the next line creates one constraint for each member of the set model.I
model.AxbConstraint = pyo.Constraint(model.I, rule=ax_constraint_rule)
```

A more advanced example is provided in `abstract2piecebuild.py` in *BuildAction and BuildCheck*.

`Expression` Objects

Pyomo `Expression` objects are very similar to the `Param` component (with `mutable=True`) except that the underlying values can be numeric constants or Pyomo expressions. Here's an illustration of expression objects in an AbstractModel. An expression object with an index set that is the numbers 1, 2, 3 is created and initialized to be the model variable x times the index. Later in the model file, just to illustrate how to do it, the expression is changed but just for the first index to be x squared.

```
model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.0)


def _e(m, i):
    return m.x * i


model.e = pyo.Expression([1, 2, 3], rule=_e)

instance = model.create_instance()

print(pyo.value(instance.e[1]))  # -> 1.0
print(instance.e[1]())  # -> 1.0
print(instance.e[1].value)  # -> a pyomo expression object

# Change the underlying expression
instance.e[1].value = instance.x**2

# ... solve
# ... load results
```

(continues on next page)

(continued from previous page)

```
# print the value of the expression given the loaded optimal solution
print(pyo.value(instance.e[1]))
```

An alternative is to create Python functions that, potentially, manipulate model objects. E.g., if you define a function

```
def f(x, p):
    return x + p
```

You can call this function with or without Pyomo modeling components as the arguments. E.g., f(2,3) will return a number, whereas f(model.x, 3) will return a Pyomo expression due to operator overloading.

If you take this approach you should note that anywhere a Pyomo expression is used to generate another expression (e.g., f(model.x, 3) + 5), the initial expression is always cloned so that the new generated expression is independent of the old. For example:

```
model = pyo.ConcreteModel()
model.x = pyo.Var()

# create a Pyomo expression
e1 = model.x + 5

# create another Pyomo expression
# e1 is copied when generating e2
e2 = e1 + model.x
```

If you want to create an expression that is shared between other expressions, you can use the `Expression` component.

Special Ordered Sets (SOS)

Pyomo allows users to declare special ordered sets (SOS) within their problems. These are sets of variables among which only a certain number of variables can be non-zero, and those that are must be adjacent according to a given order.

Special ordered sets of types 1 (SOS1) and 2 (SOS2) are the classic ones, but the concept can be generalised: a SOS of type N cannot have more than N of its members taking non-zero values, and those that do must be adjacent in the set. These can be useful for modelling and computational performance purposes.

By explicitly declaring these, users can keep their formulations and respective solving times shorter than they would otherwise, since the logical constraints that enforce the SOS do not need to be implemented within the model and are instead (ideally) handled algorithmically by the solver.

Special ordered sets can be declared one by one or indexed via other sets.

Non-indexed Special Ordered Sets

A single SOS of type **N** involving all members of a pyomo Var component can be declared in one line:

```
# import pyomo
import pyomo.environ as pyo
# declare the model
model = pyo.AbstractModel()
# the type of SOS
```

(continues on next page)

(continued from previous page)

```
N = 1 # or 2, 3, ...
# the set that indexes the variables
model.A = pyo.Set()
# the variables under consideration
model.x = pyo.Var(model.A)
# the sos constraint
model.mysos = pyo.SOSConstraint(var=model.x, sos=N)
```

In the example above, the weight of each variable is determined automatically based on their position/order in the pyomo Var component (`model.x`).

Alternatively, the weights can be specified through a pyomo Param component (`model.mysosweights`) indexed by the set also indexing the variables (`model.A`):

```
# the set that indexes the variables
model.A = pyo.Set()
# the variables under consideration
model.x = pyo.Var(model.A)
# the weights for each variable used in the sos constraints
model.mysosweights = pyo.Param(model.A)
# the sos constraint
model.mysos = pyo.SOSConstraint(
    var=model.x,
    sos=N,
    weights=model.mysosweights
    )
```

Indexed Special Ordered Sets

Multiple SOS of type **N** involving members of a pyomo Var component (`model.x`) can be created using two additional sets (`model.A` and `model.mysosvarindexset`):

```
# the set that indexes the variables
model.A = pyo.Set()
# the variables under consideration
model.x = pyo.Var(model.A)
# the set indexing the sos constraints
model.B = pyo.Set()
# the sets containing the variable indexes for each constraint
model.mysosvarindexset = pyo.Set(model.B)
# the sos constraints
model.mysos = pyo.SOSConstraint(
    model.B,
    var=model.x,
    sos=N,
    index=model.mysosvarindexset
    )
```

In the example above, the weights are determined automatically from the position of the variables. Alternatively, they can be specified through a pyomo Param component (`model.mysosweights`) and an additional set (`model.C`):

```
# the set that indexes the variables
model.A = pyo.Set()
```

(continues on next page)

(continued from previous page)

```
# the variables under consideration
model.x = pyo.Var(model.A)
# the set indexing the sos constraints
model.B = pyo.Set()
# the sets containing the variable indexes for each constraint
model.mysosvarindexset = pyo.Set(model.B)
# the set that indexes the variables used in the sos constraints
model.C = pyo.Set(within=model.A)
# the weights for each variable used in the sos constraints
model.mysosweights = pyo.Param(model.C)
# the sos constraints
model.mysos = pyo.SOSConstraint(
    model.B,
    var=model.x,
    sos=N,
    index=model.mysosvarindexset,
    weights=model.mysosweights,
    )
```

Declaring Special Ordered Sets using rules

Arguably the best way to declare an SOS is through rules. This option allows users to specify the variables and weights through a method provided via the `rule` parameter. If this parameter is used, users must specify a method that returns one of the following options:

- a list of the variables in the SOS, whose respective weights are then determined based on their position;
- a tuple of two lists, the first for the variables in the SOS and the second for the respective weights;
- or, pyomo.environ.SOSConstraint.Skip, if the SOS is not to be declared.

If one is content on having the weights determined based on the position of the variables, then the following example using the `rule` parameter is sufficient:

```
# the set that indexes the variables
model.A = pyo.Set()
# the variables under consideration
model.x = pyo.Var(model.A, domain=pyo.NonNegativeReals)
# the rule method creating the constraint
def rule_mysos(m):
    return [m.x[a] for a in m.x]
# the sos constraint(s)
model.mysos = pyo.SOSConstraint(rule=rule_mysos, sos=N)
```

If the weights must be determined in some other way, then the following example illustrates how they can be specified for each member of the SOS using the `rule` parameter:

```
# the set that indexes the variables
model.A = pyo.Set()
# the variables under consideration
model.x = pyo.Var(model.A, domain=pyo.NonNegativeReals)
# the rule method creating the constraint
def rule_mysos(m):
    var_list = [m.x[a] for a in m.x]
```

(continues on next page)

(continued from previous page)

```
    weight_list = [i+1 for i in range(len(var_list))]
    return (var_list, weight_list)
# the sos constraint(s)
model.mysos = pyo.SOSConstraint(rule=rule_mysos, sos=N)
```

The `rule` parameter also allows users to create SOS comprising variables from different pyomo Var components, as shown below:

```
# the set that indexes the x variables
model.A = pyo.Set()
# the set that indexes the y variables
model.B = pyo.Set()
# the set that indexes the SOS constraints
model.C = pyo.Set()
# the x variables, which will be used in the constraints
model.x = pyo.Var(model.A, domain=pyo.NonNegativeReals)
# the y variables, which will be used in the constraints
model.y = pyo.Var(model.B, domain=pyo.NonNegativeReals)
# the x variable indices for each constraint
model.mysosindex_x = pyo.Set(model.C)
# the y variable indices for each constraint
model.mysosindex_y = pyo.Set(model.C)
# the weights for the x variable indices
model.mysosweights_x = pyo.Param(model.A)
# the weights for the y variable indices
model.mysosweights_y = pyo.Param(model.B)
# the rule method with which each constraint c is built
def rule_mysos(m, c):
    var_list = [m.x[a] for a in m.mysosindex_x[c]]
    var_list.extend([m.y[b] for b in m.mysosindex_y[c]])
    weight_list = [m.mysosweights_x[a] for a in m.mysosindex_x[c]]
    weight_list.extend([m.mysosweights_y[b] for b in m.mysosindex_y[c]])
    return (var_list, weight_list)
# the sos constraint(s)
model.mysos = pyo.SOSConstraint(
    model.C,
    rule=rule_mysos,
    sos=N
    )
```

Compatible solvers

Not all LP/MILP solvers are compatible with SOS declarations and Pyomo might not be ready to interact with all those that are. The following is a list of solvers known to be compatible with special ordered sets through Pyomo:

- CBC
- SCIP
- Gurobi
- CPLEX

Please note that declaring an SOS is no guarantee that a solver will use it as such in the end. Some solvers, namely Gurobi and CPLEX, might reformulate problems with explicit SOS declarations, if they perceive that to be useful.

Full example with non-indexed SOS constraint

```
import pyomo.environ as pyo
from pyomo.opt import check_available_solvers
from math import isclose
N = 1
model = pyo.ConcreteModel()
model.x = pyo.Var([1], domain=pyo.NonNegativeReals, bounds=(0,40))
model.A = pyo.Set(initialize=[1,2,4,6])
model.y = pyo.Var(model.A, domain=pyo.NonNegativeReals, bounds=(0,2))
model.OBJ = pyo.Objective(
    expr=(1*model.x[1]+
          2*model.y[1]+
          3*model.y[2]+
          -0.1*model.y[4]+
          0.5*model.y[6])
    )
model.ConstraintYmin = pyo.Constraint(
    expr = (model.x[1]+
            model.y[1]+
            model.y[2]+
            model.y[6] >= 0.25
            )
    )
model.mysos = pyo.SOSConstraint(
    var=model.y,
    sos=N
    )
solver_name = 'scip'
solver_available = bool(check_available_solvers(solver_name))
if solver_available:
     opt = pyo.SolverFactory(solver_name)
     opt.solve(model, tee=False)
     assert isclose(pyo.value(model.OBJ), 0.05, abs_tol=1e-3)
```

Suffixes

Suffixes provide a mechanism for declaring extraneous model data, which can be used in a number of contexts. Most commonly, suffixes are used by solver plugins to store extra information about the solution of a model. This and other suffix functionality is made available to the modeler through the use of the Suffix component class. Uses of Suffix include:

- Importing extra information from a solver about the solution of a mathematical program (e.g., constraint duals, variable reduced costs, basis information).
- Exporting information to a solver or algorithm to aid in solving a mathematical program (e.g., warm-starting information, variable branching priorities).
- Tagging modeling components with local data for later use in advanced scripting algorithms.

Suffix Notation and the Pyomo NL File Interface

The Suffix component used in Pyomo has been adapted from the suffix notation used in the modeling language AMPL [FGK02]. Therefore, it follows naturally that AMPL style suffix functionality is fully available using Pyomo's NL file interface. For information on AMPL style suffixes the reader is referred to the AMPL website:

> http://www.ampl.com

A number of scripting examples that highlight the use AMPL style suffix functionality are available in the `examples/pyomo/suffixes` directory distributed with Pyomo.

Declaration

The effects of declaring a Suffix component on a Pyomo model are determined by the following traits:

- direction: This trait defines the direction of information flow for the suffix. A suffix direction can be assigned one of four possible values:
  - `LOCAL` - suffix data stays local to the modeling framework and will not be imported or exported by a solver plugin (default)
  - `IMPORT` - suffix data will be imported from the solver by its respective solver plugin
  - `EXPORT` - suffix data will be exported to a solver by its respective solver plugin
  - `IMPORT_EXPORT` - suffix data flows in both directions between the model and the solver or algorithm
- datatype: This trait advertises the type of data held on the suffix for those interfaces where it matters (e.g., the NL file interface). A suffix datatype can be assigned one of three possible values:
  - `FLOAT` - the suffix stores floating point data (default)
  - `INT` - the suffix stores integer data
  - `None` - the suffix stores any type of data

Note

Exporting suffix data through Pyomo's NL file interface requires all active export suffixes have a strict datatype (i.e., `datatype=None` is not allowed).

The following code snippet shows examples of declaring a Suffix component on a Pyomo model:

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()

# Export integer data
model.priority = pyo.Suffix(
    direction=pyo.Suffix.EXPORT, datatype=pyo.Suffix.INT)

# Export and import floating point data
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT_EXPORT)

# Store floating point data
model.junk = pyo.Suffix()
```

Declaring a Suffix with a non-local direction on a model is not guaranteed to be compatible with all solver plugins in Pyomo. Whether a given Suffix is acceptable or not depends on both the solver and solver interface being used. In

some cases, a solver plugin will raise an exception if it encounters a Suffix type that it does not handle, but this is not true in every situation. For instance, the NL file interface is generic to all AMPL-compatible solvers, so there is no way to validate that a Suffix of a given name, direction, and datatype is appropriate for a solver. One should be careful in verifying that Suffix declarations are being handled as expected when switching to a different solver or solver interface.

Operations

The Suffix component class provides a dictionary interface for mapping Pyomo modeling components to arbitrary data. This mapping functionality is captured within the ComponentMap base class, which is also available within Pyomo's modeling environment. The ComponentMap can be used as a more lightweight replacement for Suffix in cases where a simple mapping from Pyomo modeling components to arbitrary data values is required.

Note

ComponentMap and Suffix use the built-in `id()` function for hashing entry keys. This design decision arises from the fact that most of the modeling components found in Pyomo are either not hashable or use a hash based on a mutable numeric value, making them unacceptable for use as keys with the built-in `dict` class.

Warning

The use of the built-in `id()` function for hashing entry keys in ComponentMap and Suffix makes them inappropriate for use in situations where built-in object types must be used as keys. It is strongly recommended that only Pyomo modeling components be used as keys in these mapping containers (`Var`, `Constraint`, etc.).

Warning

Do not attempt to pickle or deepcopy instances of ComponentMap or Suffix unless doing so along with the components for which they hold mapping entries. As an example, placing one of these objects on a model and then cloning or pickling that model is an acceptable scenario.

In addition to the dictionary interface provided through the ComponentMap base class, the Suffix component class also provides a number of methods whose default semantics are more convenient for working with indexed modeling components. The easiest way to highlight this functionality is through the use of an example.

```
model = pyo.ConcreteModel()
model.x = pyo.Var()
model.y = pyo.Var([1,2,3])
model.foo = pyo.Suffix()
```

In this example we have a concrete Pyomo model with two different types of variable components (indexed and non-indexed) as well as a Suffix declaration (foo). The next code snippet shows examples of adding entries to the suffix foo.

```
# Assign a suffix value of 1.0 to model.x
model.foo.set_value(model.x, 1.0)

# Same as above with dict interface
model.foo[model.x] = 1.0

```

(continues on next page)

(continued from previous page)

```
# Assign a suffix value of 0.0 to all indices of model.y
# By default this expands so that entries are created for
# every index (y[1], y[2], y[3]) and not model.y itself
model.foo.set_value(model.y, 0.0)

# The same operation using the dict interface results in an entry only
# for the parent component model.y
model.foo[model.y] = 50.0

# Assign a suffix value of -1.0 to model.y[1]
model.foo.set_value(model.y[1], -1.0)

# Same as above with the dict interface
model.foo[model.y[1]] = -1.0
```

In this example we highlight the fact that the `__setitem__` and `setValue` entry methods can be used interchangeably except in the case where indexed components are used (model.y). In the indexed case, the `__setitem__` approach creates a single entry for the parent indexed component itself, whereas the `setValue` approach by default creates an entry for each index of the component. This behavior can be controlled using the optional keyword 'expand', where assigning it a value of `False` results in the same behavior as `__setitem__`.

Other operations like accessing or removing entries in our mapping can performed as if the built-in `dict` class is in use.

```
>>> print(model.foo.get(model.x))
1.0
>>> print(model.foo[model.x])
1.0

>>> print(model.foo.get(model.y[1]))
-1.0
>>> print(model.foo[model.y[1]])
-1.0

>>> print(model.foo.get(model.y[2]))
0.0
>>> print(model.foo[model.y[2]])
0.0

>>> print(model.foo.get(model.y))
50.0
>>> print(model.foo[model.y])
50.0

>>> del model.foo[model.y]
>>> print(model.foo.get(model.y))
None

>>> print(model.foo[model.y])
Traceback (most recent call last):
  ...
KeyError: "Component with id '...': y"
```

The non-dict method `clear_value` can be used in place of `__delitem__` to remove entries, where it inherits the same default behavior as `setValue` for indexed components and does not raise a KeyError when the argument does not exist as a key in the mapping.

```
>>> model.foo.clear_value(model.y)

>>> print(model.foo[model.y[1]])
Traceback (most recent call last):
  ...
KeyError: "Component with id '...': y[1]"

>>> del model.foo[model.y[1]]
Traceback (most recent call last):
  ...
KeyError: "Component with id '...': y[1]"

>>> model.foo.clear_value(model.y[1])
```

A summary non-dict Suffix methods is provided here:

clearAllValues()
: Clears all suffix data.

clear_value(component, expand=True)
: Clears suffix information for a component.

setAllValues(value)
: Sets the value of this suffix on all components.

setValue(component, value, expand=True)
: Sets the value of this suffix on the specified component.

updateValues(data_buffer, expand=True)
: Updates the suffix data given a list of component,value tuples. Provides an improvement in efficiency over calling setValue on every component.

getDatatype()
: Return the suffix datatype.

setDatatype(datatype)
: Set the suffix datatype.

getDirection()
: Return the suffix direction.

setDirection(direction)
: Set the suffix direction.

importEnabled()
: Returns True when this suffix is enabled for import from solutions.

exportEnabled()

Returns True when this suffix is enabled for export to solvers.

Importing Suffix Data

Importing suffix information from a solver solution is achieved by declaring a Suffix component with the appropriate name and direction. Suffix names available for import may be specific to third-party solvers as well as individual solver interfaces within Pyomo. The most common of these, available with most solvers and solver interfaces, is constraint dual multipliers. Requesting that duals be imported into suffix data can be accomplished by declaring a Suffix component on the model.

```
model = pyo.ConcreteModel()
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)
model.x = pyo.Var()
model.obj = pyo.Objective(expr=model.x)
model.con = pyo.Constraint(expr=model.x >= 1.0)
```

The existence of an active suffix with the name dual that has an import style suffix direction will cause constraint dual information to be collected into the solver results (assuming the solver supplies dual information). In addition to this, after loading solver results into a problem instance (using a python script or Pyomo callback functions in conjunction with the `pyomo` command), one can access the dual values associated with constraints using the dual Suffix component.

```
>>> results = pyo.SolverFactory('glpk').solve(model)
>>> pyo.assert_optimal_termination(results)
>>> print(model.dual[model.con])
1.0
```

Alternatively, the `pyomo` option `--solver-suffixes` can be used to request suffix information from a solver. In the event that suffix names are provided via this command-line option, the `pyomo` script will automatically declare these Suffix components on the constructed instance making these suffixes available for import.

Exporting Suffix Data

Exporting suffix data is accomplished in a similar manner as to that of importing suffix data. One simply needs to declare a Suffix component on the model with an export style suffix direction and associate modeling component values with it. The following example shows how one can declare a special ordered set of type 1 using AMPL-style suffix notation in conjunction with Pyomo's NL file interface.

```
model = pyo.ConcreteModel()
model.y = pyo.Var([1,2,3], within=pyo.NonNegativeReals)

model.sosno = pyo.Suffix(direction=pyo.Suffix.EXPORT)
model.ref = pyo.Suffix(direction=pyo.Suffix.EXPORT)

# Add entry for each index of model.y
model.sosno.set_value(model.y, 1)
model.ref[model.y[1]] = 0
model.ref[model.y[2]] = 1
model.ref[model.y[3]] = 2
```

Most AMPL-compatible solvers will recognize the suffix names `sosno` and `ref` as declaring a special ordered set, where a positive value for `sosno` indicates a special ordered set of type 1 and a negative value indicates a special ordered set of type 2.

> **Note**
>
> Pyomo provides the `SOSConstraint` component for declaring special ordered sets, which is recognized by all solver interfaces, including the NL file interface.

Pyomo's NL file interface will recognize an EXPORT style Suffix component with the name 'dual' as supplying initializations for constraint multipliers. As such it will be treated separately than all other EXPORT style suffixes encountered in the NL writer, which are treated as AMPL-style suffixes. The following example script shows how one can warmstart the interior-point solver Ipopt by supplying both primal (variable values) and dual (suffixes) solution information. This dual suffix information can be both imported and exported using a single Suffix component with an IMPORT_EXPORT direction.

```
model = pyo.ConcreteModel()
model.x1 = pyo.Var(bounds=(1,5),initialize=1.0)
model.x2 = pyo.Var(bounds=(1,5),initialize=5.0)
model.x3 = pyo.Var(bounds=(1,5),initialize=5.0)
model.x4 = pyo.Var(bounds=(1,5),initialize=1.0)
model.obj = pyo.Objective(
    expr=model.x1*model.x4*(model.x1 + model.x2 + model.x3) + model.x3)
model.inequality = pyo.Constraint(
    expr=model.x1*model.x2*model.x3*model.x4 >= 25.0)
model.equality = pyo.Constraint(
    expr=model.x1**2 + model.x2**2 + model.x3**2 + model.x4**2 == 40.0)

### Declare all suffixes
# Ipopt bound multipliers (obtained from solution)
model.ipopt_zL_out = pyo.Suffix(direction=pyo.Suffix.IMPORT)
model.ipopt_zU_out = pyo.Suffix(direction=pyo.Suffix.IMPORT)
# Ipopt bound multipliers (sent to solver)
model.ipopt_zL_in = pyo.Suffix(direction=pyo.Suffix.EXPORT)
model.ipopt_zU_in = pyo.Suffix(direction=pyo.Suffix.EXPORT)
# Obtain dual solutions from first solve and send to warm start
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT_EXPORT)

ipopt = pyo.SolverFactory('ipopt')
```

The difference in performance can be seen by examining Ipopt's iteration log with and without warm starting:

- Without Warmstart:

```
ipopt.solve(model, tee=True)
```

```
...
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.6109693e+01 1.12e+01 5.28e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.6982239e+01 7.30e-01 1.02e+01  -1.0 6.11e-01    -  7.19e-02 1.00e+00f  1
   2  1.7318411e+01 ...
   ...
   8  1.7014017e+01 ...

Number of Iterations....: 8
...
```

- With Warmstart:

```
### Set Ipopt options for warm-start
# The current values on the ipopt_zU_out and ipopt_zL_out suffixes will
# be used as initial conditions for the bound multipliers to solve the
# new problem
model.ipopt_zL_in.update(model.ipopt_zL_out)
model.ipopt_zU_in.update(model.ipopt_zU_out)
ipopt.options['warm_start_init_point'] = 'yes'
ipopt.options['warm_start_bound_push'] = 1e-6
ipopt.options['warm_start_mult_bound_push'] = 1e-6
ipopt.options['mu_init'] = 1e-6

ipopt.solve(model, tee=True)
```

```
...
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.7014032e+01 2.00e-06 4.07e-06  -6.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.7014019e+01 3.65e-12 1.00e-11  -6.0 2.50e-01    -  1.00e+00 1.00e+00h  1
   2  1.7014017e+01 ...

Number of Iterations....: 2
...
```

Using Suffixes With an AbstractModel

In order to allow the declaration of suffix data within the framework of an AbstractModel, the Suffix component can be initialized with an optional construction rule. As with constraint rules, this function will be executed at the time of model construction. The following simple example highlights the use of the `rule` keyword in suffix initialization. Suffix rules are expected to return an iterable of (component, value) tuples, where the `expand=True` semantics are applied for indexed components.

```
model = pyo.AbstractModel()
model.x = pyo.Var()
model.c = pyo.Constraint(expr=model.x >= 1)

def foo_rule(m):
   return ((m.x, 2.0), (m.c, 3.0))
model.foo = pyo.Suffix(rule=foo_rule)
```

```
>>> # Instantiate the model
>>> inst = model.create_instance()

>>> print(inst.foo[inst.x])
2.0
>>> print(inst.foo[inst.c])
3.0

>>> # Note that model.x and inst.x are not the same object
>>> print(inst.foo[model.x])
Traceback (most recent call last):
  ...
KeyError: "Component with id '...': x"
```

The next example shows an abstract model where suffixes are attached only to the variables:

```
model = pyo.AbstractModel()
model.I = pyo.RangeSet(1,4)
model.x = pyo.Var(model.I)
def c_rule(m, i):
    return m.x[i] >= i
model.c = pyo.Constraint(model.I, rule=c_rule)

def foo_rule(m):
    return ((m.x[i], 3.0*i) for i in m.I)
model.foo = pyo.Suffix(rule=foo_rule)
```

```
>>> # instantiate the model
>>> inst = model.create_instance()
>>> for i in inst.I:
...     print((i, inst.foo[inst.x[i]]))
(1, 3.0)
(2, 6.0)
(3, 9.0)
(4, 12.0)
```

3.2.2 Dynamic Optimization with pyomo.DAE

The pyomo.DAE modeling extension [PyomoDAE-paper] allows users to incorporate systems of differential algebraic equations (DAE)s in a Pyomo model. The modeling components in this extension are able to represent ordinary or partial differential equations. The differential equations do not have to be written in a particular format and the components are flexible enough to represent higher-order derivatives or mixed partial derivatives. Pyomo.DAE also includes model transformations which use simultaneous discretization approaches to transform a DAE model into an algebraic model. Finally, pyomo.DAE includes utilities for simulating DAE models and initializing dynamic optimization problems.

Modeling Components

Pyomo.DAE introduces three new modeling components to Pyomo:

`pyomo.dae.ContinuousSet`	Represents a bounded continuous domain
`pyomo.dae.DerivativeVar`	Represents derivatives in a model and defines how a `Var` is differentiated
`pyomo.dae.Integral`	Represents an integral over a continuous domain

As will be shown later, differential equations can be declared using using these new modeling components along with the standard Pyomo *Var* and *Constraint* components.

ContinuousSet

This component is used to define continuous bounded domains (for example 'spatial' or 'time' domains). It is similar to a Pyomo *Set* component and can be used to index things like variables and constraints. Any number of `ContinuousSets` can be used to index a component and components can be indexed by both *Sets* and `ContinuousSets` in arbitrary order.

In the current implementation, models with ContinuousSet components may not be solved until every ContinuousSet has been discretized. Minimally, a ContinuousSet must be initialized with two numeric values representing the upper and lower bounds of the continuous domain. A user may also specify additional points in the domain to be used as finite element points in the discretization.

class pyomo.dae.**ContinuousSet**(**args*, ***kwds*)

Represents a bounded continuous domain

Minimally, this set must contain two numeric values defining the bounds of a continuous range. Discrete points of interest may be added to the continuous set. A continuous set is one dimensional and may only contain numerical values.

Parameters

- **initialize** (*list*) – Default discretization points to be included
- **bounds** (*tuple*) – The bounding points for the continuous domain. The bounds will be included as discrete points in the ContinuousSet and will be used to bound the points added to the ContinuousSet through the 'initialize' argument, a data file, or the add() method

_changed

This keeps track of whether or not the ContinuousSet was changed during discretization. If the user specifies all of the needed discretization points before the discretization then there is no need to go back through the model and reconstruct things indexed by the ContinuousSet

Type

boolean

_fe

This is a sorted list of the finite element points in the ContinuousSet. i.e. this list contains all the discrete points in the ContinuousSet that are not collocation points. Points that are both finite element points and collocation points will be included in this list.

Type

list

_discretization_info

This is a dictionary which contains information on the discretization transformation which has been applied to the ContinuousSet.

Type

dict

construct(*values=None*)

Constructs a ContinuousSet component

find_nearest_index(*target*, *tolerance=None*)

Returns the index of the nearest point in the ContinuousSet.

If a tolerance is specified, the index will only be returned if the distance between the target and the closest point is less than or equal to that tolerance. If there is a tie for closest point, the index on the left is returned.

Parameters

- **target** (*float*)
- **tolerance** (*float* or *None*)

Return type

float or *None*

get_changed()

Returns flag indicating if the ContinuousSet was changed during discretization

Returns "True" if additional points were added to the ContinuousSet while applying a discretization scheme

Return type
boolean

get_discretization_info()

Returns a *dict* with information on the discretization scheme that has been applied to the ContinuousSet.

Return type
dict

get_finite_elements()

Returns the finite element points

If the ContinuousSet has been discretizaed using a collocation scheme, this method will return a list of the finite element discretization points but not the collocation points within each finite element. If the ContinuousSet has not been discretized or a finite difference discretization was used, this method returns a list of all the discretization points in the ContinuousSet.

Return type
list of *floats*

get_lower_element_boundary(*point*)

Returns the first finite element point that is less than or equal to 'point'

Parameters
point (*float*)

Return type
float

get_upper_element_boundary(*point*)

Returns the first finite element point that is greater or equal to 'point'

Parameters
point (*float*)

Return type
float

set_changed(*newvalue*)

Sets the _changed flag to 'newvalue'

Parameters
newvalue (*boolean*)

The following code snippet shows examples of declaring a ContinuousSet component on a concrete Pyomo model:

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.dae import ContinuousSet

>>> model = pyo.ConcreteModel()

Declaration by providing bounds
>>> model.t = ContinuousSet(bounds=(0,5))
```

(continues on next page)

(continued from previous page)

```
Declaration by initializing with desired discretization points
>>> model.x = ContinuousSet(initialize=[0,1,2,5])
```

Note

A `ContinuousSet` may not be constructed unless at least two numeric points are provided to bound the continuous domain.

The following code snippet shows an example of declaring a `ContinuousSet` component on an abstract Pyomo model using the example data file.

```
set t := 0 0.5 2.25 3.75 5;
```

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.dae import ContinuousSet

>>> model = pyo.AbstractModel()

The ContinuousSet below will be initialized using the points
in the data file when a model instance is created.
>>> model.t = ContinuousSet()
```

Note

If a separate data file is used to initialize a `ContinuousSet`, it is done using the 'set' command and not 'continuousset'

Note

Most valid ways to declare and initialize a *Set* can be used to declare and initialize a `ContinuousSet`. See the documentation for *Set* for additional options.

Warning

Be careful using a `ContinuousSet` as an implicit index in an expression, i.e. `sum(m.v[i] for i in m.myContinuousSet)`. The expression will be generated using the discretization points contained in the `ContinuousSet` at the time the expression was constructed and will not be updated if additional points are added to the set during discretization.

Note

`ContinuousSet` components are always ordered (sorted) therefore the `first()` and `last()` *Set* methods can be used to access the lower and upper boundaries of the `ContinuousSet` respectively

DerivativeVar

class pyomo.dae.**DerivativeVar**(**args*, ***kwargs*)

Represents derivatives in a model and defines how a `Var` is differentiated

The `DerivativeVar` component is used to declare a derivative of a `Var`. The constructor accepts a single positional argument which is the `Var` that's being differentiated. A `Var` may only be differentiated with respect to a `ContinuousSet` that it is indexed by. The indexing sets of a `DerivativeVar` are identical to those of the `Var` it is differentiating.

Parameters

- **sVar** (`pyomo.environ.Var`) – The variable being differentiated
- **wrt** (`pyomo.dae.ContinuousSet` or tuple) – Equivalent to *withrespectto* keyword argument. The `ContinuousSet` that the derivative is being taken with respect to. Higher order derivatives are represented by including the `ContinuousSet` multiple times in the tuple sent to this keyword. i.e. `wrt=(m.t, m.t)` would be the second order derivative with respect to `m.t`

get_continuousset_list()

Return the a list of `ContinuousSet` components the derivative is being taken with respect to.

Return type
list

get_derivative_expression()

Returns the current discretization expression for this derivative or creates an access function to its `Var` the first time this method is called. The expression gets built up as the discretization transformations are sequentially applied to each `ContinuousSet` in the model.

get_state_var()

Return the `Var` that is being differentiated.

Return type
`Var`

is_fully_discretized()

Check to see if all the `ContinuousSets` this derivative is taken with respect to have been discretized.

Return type
boolean

set_derivative_expression(*expr*)

Sets``_expr``, an expression representing the discretization equations linking the `DerivativeVar` to its state `Var`

The code snippet below shows examples of declaring `DerivativeVar` components on a Pyomo model. In each case, the variable being differentiated is supplied as the only positional argument and the type of derivative is specified using the 'wrt' (or the more verbose 'withrespectto') keyword argument. Any keyword argument that is valid for a Pyomo *Var* component may also be specified.

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.dae import ContinuousSet, DerivativeVar

>>> model = pyo.ConcreteModel()
>>> model.s = pyo.Set(initialize=['a','b'])
>>> model.t = ContinuousSet(bounds=(0,5))
```

(continues on next page)

(continued from previous page)

```
>>> model.l = ContinuousSet(bounds=(-10,10))

>>> model.x = pyo.Var(model.t)
>>> model.y = pyo.Var(model.s,model.t)
>>> model.z = pyo.Var(model.t,model.l)

Declare the first derivative of model.x with respect to model.t
>>> model.dxdt = DerivativeVar(model.x, withrespectto=model.t)

Declare the second derivative of model.y with respect to model.t
Note that this DerivativeVar will be indexed by both model.s and model.t
>>> model.dydt2 = DerivativeVar(model.y, wrt=(model.t,model.t))

Declare the partial derivative of model.z with respect to model.l
Note that this DerivativeVar will be indexed by both model.t and model.l
>>> model.dzdl = DerivativeVar(model.z, wrt=(model.l), initialize=0)

Declare the mixed second order partial derivative of model.z with respect
to model.t and model.l and set bounds
>>> model.dz2 = DerivativeVar(model.z, wrt=(model.t, model.l), bounds=(-10, 10))
```

Note

The 'initialize' keyword argument will initialize the value of a derivative and is **not** the same as specifying an initial condition. Initial or boundary conditions should be specified using a *Constraint* or *ConstraintList* or by fixing the value of a *Var* at a boundary point.

Declaring Differential Equations

A differential equations is declared as a standard Pyomo *Constraint* and is not required to have any particular form. The following code snippet shows how one might declare an ordinary or partial differential equation.

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.dae import ContinuousSet, DerivativeVar, Integral

>>> model = pyo.ConcreteModel()
>>> model.s = pyo.Set(initialize=['a', 'b'])
>>> model.t = ContinuousSet(bounds=(0, 5))
>>> model.l = ContinuousSet(bounds=(-10, 10))

>>> model.x = pyo.Var(model.s, model.t)
>>> model.y = pyo.Var(model.t, model.l)
>>> model.dxdt = DerivativeVar(model.x, wrt=model.t)
>>> model.dydt = DerivativeVar(model.y, wrt=model.t)
>>> model.dydl2 = DerivativeVar(model.y, wrt=(model.l, model.l))

An ordinary differential equation
>>> def _ode_rule(m, s, t):
...     if t == 0:
...         return pyo.Constraint.Skip
```

(continues on next page)

(continued from previous page)

```
...     return m.dxdt[s, t] == m.x[s, t]**2
>>> model.ode = pyo.Constraint(model.s, model.t, rule=_ode_rule)

A partial differential equation
>>> def _pde_rule(m, t, l):
...     if t == 0 or l == m.l.first() or l == m.l.last():
...        return pyo.Constraint.Skip
...     return m.dydt[t, l] == m.dydl2[t, l]
>>> model.pde = pyo.Constraint(model.t, model.l, rule=_pde_rule)
```

By default, a *Constraint* declared over a ContinuousSet will be applied at every discretization point contained in the set. Often a modeler does not want to enforce a differential equation at one or both boundaries of a continuous domain. This may be addressed explicitly in the *Constraint* declaration using Constraint.Skip as shown above. Alternatively, the desired constraints can be deactivated just before the model is sent to a solver as shown below.

```
>>> def _ode_rule(m, s, t):
...     return m.dxdt[s, t] == m.x[s, t]**2
>>> model.ode = pyo.Constraint(model.s, model.t, rule=_ode_rule)

>>> def _pde_rule(m, t, l):
...     return m.dydt[t, l] == m.dydl2[t, l]
>>> model.pde = pyo.Constraint(model.t, model.l, rule=_pde_rule)

Declare other model components and apply a discretization transformation
...

Deactivate the differential equations at certain boundary points
>>> for con in model.ode[:, model.t.first()]:
...     con.deactivate()

>>> for con in model.pde[0, :]:
...     con.deactivate()

>>> for con in model.pde[:, model.l.first()]:
...     con.deactivate()

>>> for con in model.pde[:, model.l.last()]:
...    con.deactivate()

Solve the model
...
```

Note

If you intend to use the pyomo.DAE Simulator on your model then you **must** use **constraint deactivation** instead of **constraint skipping** in the differential equation rule.

Declaring Integrals

Warning

The `Integral` component is still under development and considered a prototype. It currently includes only basic functionality for simple integrals. We welcome feedback on the interface and functionality but **we do not recommend using it** on general models. Instead, integrals should be reformulated as differential equations.

class `pyomo.dae.Integral`(**args*, ***kwds*)

Represents an integral over a continuous domain

The `Integral` component can be used to represent an integral taken over the entire domain of a `ContinuousSet`. Once every `ContinuousSet` in a model has been discretized, any integrals in the model will be converted to algebraic equations using the trapezoid rule. Future development will include more sophisticated numerical integration methods.

Parameters

- ***args** – Every indexing set needed to evaluate the integral expression
- **wrt** (`ContinuousSet`) – The continuous domain over which the integral is being taken
- **rule** (*function*) – Function returning the expression being integrated

get_continuousset()

Return the `ContinuousSet` the integral is being taken over

Declaring an `Integral` component is similar to declaring an *Expression* component. A simple example is shown below:

```
>>> model = pyo.ConcreteModel()
>>> model.time = ContinuousSet(bounds=(0,10))
>>> model.X = pyo.Var(model.time)
>>> model.scale = pyo.Param(initialize=1E-3)

>>> def _intX(m,t):
...     return m.X[t]
>>> model.intX = Integral(model.time,wrt=model.time,rule=_intX)

>>> def _obj(m):
...     return m.scale*m.intX
>>> model.obj = pyo.Objective(rule=_obj)
```

Notice that the positional arguments supplied to the `Integral` declaration must include all indices needed to evaluate the integral expression. The integral expression is defined in a function and supplied to the 'rule' keyword argument. Finally, a user must specify a `ContinuousSet` that the integral is being evaluated over. This is done using the 'wrt' keyword argument.

Note

The `ContinuousSet` specified using the 'wrt' keyword argument must be explicitly specified as one of the indexing sets (meaning it must be supplied as a positional argument). This is to ensure consistency in the ordering and dimension of the indexing sets

After an `Integral` has been declared, it can be used just like a Pyomo *Expression* component and can be included in constraints or the objective function as shown above.

If an Integral is specified with multiple positional arguments, i.e. multiple indexing sets, the final component will be indexed by all of those sets except for the ContinuousSet that the integral was taken over. In other words, the ContinuousSet specified with the 'wrt' keyword argument is removed from the indexing sets of the Integral even though it must be specified as a positional argument. This should become more clear with the following example showing a double integral over the ContinuousSet components model.t1 and model.t2. In addition, the expression is also indexed by the *Set* model.s. The mathematical representation and implementation in Pyomo are shown below:

$$\sum_s \int_{t_2} \int_{t_1} X(t_1, t_2, s) \, dt_1 \, dt_2$$

```
>>> model = pyo.ConcreteModel()
>>> model.t1 = ContinuousSet(bounds=(0, 10))
>>> model.t2 = ContinuousSet(bounds=(-1, 1))
>>> model.s = pyo.Set(initialize=['A', 'B', 'C'])

>>> model.X = pyo.Var(model.t1, model.t2, model.s)

>>> def _intX1(m, t1, t2, s):
...     return m.X[t1, t2, s]
>>> model.intX1 = Integral(model.t1, model.t2, model.s, wrt=model.t1,
...                        rule=_intX1)

>>> def _intX2(m, t2, s):
...     return m.intX1[t2, s]
>>> model.intX2 = Integral(model.t2, model.s, wrt=model.t2, rule=_intX2)

>>> def _obj(m):
...     return sum(m.intX2[k] for k in m.s)
>>> model.obj = pyo.Objective(rule=_obj)
```

Discretization Transformations

Before a Pyomo model with DerivativeVar or Integral components can be sent to a solver it must first be sent through a discretization transformation. These transformations approximate any derivatives or integrals in the model by using a numerical method. The numerical methods currently included in pyomo.DAE discretize the continuous domains in the problem and introduce equality constraints which approximate the derivatives and integrals at the discretization points. Two families of discretization schemes have been implemented in pyomo.DAE, Finite Difference and Collocation. These schemes are described in more detail below.

Note

The schemes described here are for derivatives only. All integrals will be transformed using the trapezoid rule.

The user must write a Python script in order to use these discretizations, they have not been tested on the pyomo command line. Example scripts are shown below for each of the discretization schemes. The transformations are applied to Pyomo model objects which can be further manipulated before being sent to a solver. Examples of this are also shown below.

Finite Difference Transformation

This transformation includes implementations of several finite difference methods. For example, the Backward Difference method (also called Implicit or Backward Euler) has been implemented. The discretization equations for this

method are shown below:

$$
\begin{array}{l}
\text{Given :} \\
\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0 \\
\text{discretize } t \text{ and } x \text{ such that} \\
x(t_0 + kh) = x_k \\
x_{k+1} = x_k + h * f(t_{k+1}, x_{k+1}) \\
t_{k+1} = t_k + h
\end{array}
$$

where h is the step size between discretization points or the size of each finite element. These equations are generated automatically as *Constraints* when the backward difference method is applied to a Pyomo model.

There are several discretization options available to a `dae.finite_difference` transformation which can be specified as keyword arguments to the `.apply_to()` function of the transformation object. These keywords are summarized below:

Keyword arguments for applying a finite difference transformation:

'nfe'
: The desired number of finite element points to be included in the discretization. The default value is 10.

'wrt'
: Indicates which `ContinuousSet` the transformation should be applied to. If this keyword argument is not specified then the same scheme will be applied to every `ContinuousSet` .

'scheme'
: Indicates which finite difference method to apply. Options are 'BACKWARD', 'CENTRAL', or 'FORWARD'. The default scheme is the backward difference method.

If the existing number of finite element points in a `ContinuousSet` is less than the desired number, new discretization points will be added to the set. If a user specifies a number of finite element points which is less than the number of points already included in the `ContinuousSet` then the transformation will ignore the specified number and proceed with the larger set of points. Discretization points will never be removed from a `ContinuousSet` during the discretization.

The following code is a Python script applying the backward difference method. The code also shows how to add a constraint to a discretized model.

```
Discretize model using Backward Difference method
>>> discretizer = pyo.TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,nfe=20,wrt=model.time,scheme='BACKWARD')

Add another constraint to discretized model
>>> def _sum_limit(m):
...     return sum(m.x1[i] for i in m.time) <= 50
>>> model.con_sum_limit = pyo.Constraint(rule=_sum_limit)

Solve discretized model
>>> solver = pyo.SolverFactory('ipopt')
>>> results = solver.solve(model)
```

Collocation Transformation

This transformation uses orthogonal collocation to discretize the differential equations in the model. Currently, two types of collocation have been implemented. They both use Lagrange polynomials with either Gauss-Radau roots or Gauss-Legendre roots. For more information on orthogonal collocation and the discretization equations associated with this method please see chapter 10 of the book "Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes" by L.T. Biegler.

The discretization options available to a `dae.collocation` transformation are the same as those described above for the finite difference transformation with different available schemes and the addition of the 'ncp' option.

Additional keyword arguments for collocation discretizations:

'scheme'
The desired collocation scheme, either 'LAGRANGE-RADAU' or 'LAGRANGE-LEGENDRE'. The default is 'LAGRANGE-RADAU'.

'ncp'
The number of collocation points within each finite element. The default value is 3.

Note

If the user's version of Python has access to the package Numpy then any number of collocation points may be specified, otherwise the maximum number is 10.

Note

Any points that exist in a `ContinuousSet` before discretization will be used as finite element boundaries and not as collocation points. The locations of the collocation points cannot be specified by the user, they must be generated by the transformation.

The following code is a Python script applying collocation with Lagrange polynomials and Radau roots. The code also shows how to add an objective function to a discretized model.

```
Discretize model using Radau Collocation
>>> discretizer = pyo.TransformationFactory('dae.collocation')
>>> discretizer.apply_to(model,nfe=20,ncp=6,scheme='LAGRANGE-RADAU')

Add objective function after model has been discretized
>>> def obj_rule(m):
...     return sum((m.x[i]-m.x_ref)**2 for i in m.time)
>>> model.obj = pyo.Objective(rule=obj_rule)

Solve discretized model
>>> solver = pyo.SolverFactory('ipopt')
>>> results = solver.solve(model)
```

Restricting Optimal Control Profiles

When solving an optimal control problem a user may want to restrict the number of degrees of freedom for the control input by forcing, for example, a piecewise constant profile. Pyomo.DAE provides the `reduce_collocation_points` function to address this use-case. This function is used in conjunction with the `dae.collocation` discretization transformation to reduce the number of free collocation points within a finite element for a particular variable.

class pyomo.dae.plugins.colloc.**Collocation_Discretization_Transformation**

reduce_collocation_points(*instance*, *var=None*, *ncp=None*, *contset=None*)

This method will add additional constraints to a model to reduce the number of free collocation points (degrees of freedom) for a particular variable.

Parameters

- **instance** (*Pyomo model*) – The discretized Pyomo model to add constraints to

- **var** (pyomo.environ.Var) – The Pyomo variable for which the degrees of freedom will be reduced
- **ncp** (*int*) – The new number of free collocation points for *var*. Must be less that the number of collocation points used in discretizing the model.
- **contset** (pyomo.dae.ContinuousSet) – The ContinuousSet that was discretized and for which the *var* will have a reduced number of degrees of freedom

An example of using this function is shown below:

```
>>> discretizer = pyo.TransformationFactory('dae.collocation')
>>> discretizer.apply_to(model, nfe=10, ncp=6)
>>> model = discretizer.reduce_collocation_points(model,
...                                               var=model.u,
...                                               ncp=1,
...                                               contset=model.time)
```

In the above example, the `reduce_collocation_points` function restricts the variable `model.u` to have only **1** free collocation point per finite element, thereby enforcing a piecewise constant profile. Fig. 3.1 shows the solution profile before and after applying the `reduce_collocation_points` function.

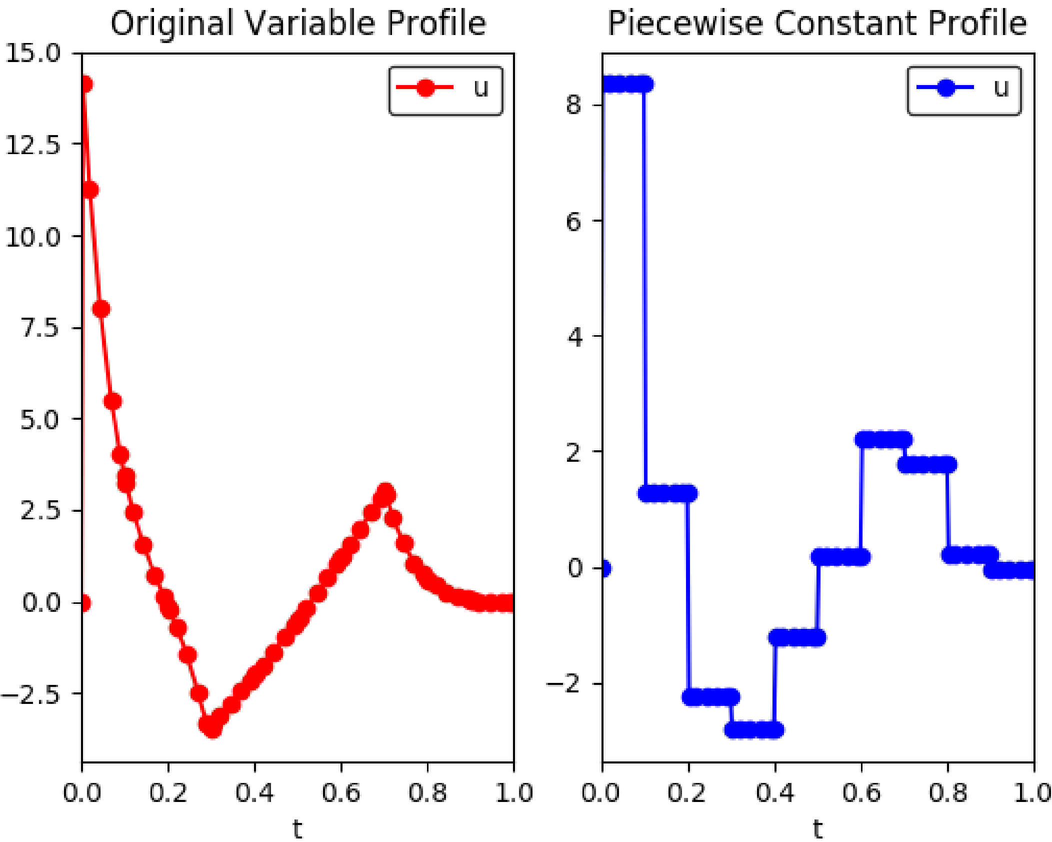


Fig. 3.1: (left) Profile before applying the `reduce_collocation_points` function (right) Profile after applying the function, restricting `model.u` to have a piecewise constant profile.

Applying Multiple Discretization Transformations

Discretizations can be applied independently to each ContinuousSet in a model. This allows the user great flexibility in discretizing their model. For example the same numerical method can be applied with different resolutions:

```
>>> discretizer = pyo.TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,wrt=model.t1,nfe=10)
>>> discretizer.apply_to(model,wrt=model.t2,nfe=100)
```

This also allows the user to combine different methods. For example, applying the forward difference method to one ContinuousSet and the central finite difference method to another ContinuousSet:

```
>>> discretizer = pyo.TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,wrt=model.t1,scheme='FORWARD')
>>> discretizer.apply_to(model,wrt=model.t2,scheme='CENTRAL')
```

In addition, the user may combine finite difference and collocation discretizations. For example:

```
>>> disc_fe = pyo.TransformationFactory('dae.finite_difference')
>>> disc_fe.apply_to(model,wrt=model.t1,nfe=10)
>>> disc_col = pyo.TransformationFactory('dae.collocation')
>>> disc_col.apply_to(model,wrt=model.t2,nfe=10,ncp=5)
```

If the user would like to apply the same discretization to all ContinuousSet components in a model, just specify the discretization once without the 'wrt' keyword argument. This will apply that scheme to all ContinuousSet components in the model that haven't already been discretized.

Custom Discretization Schemes

A transformation framework along with certain utility functions has been created so that advanced users may easily implement custom discretization schemes other than those listed above. The transformation framework consists of the following steps:

1. Specify Discretization Options
2. Discretize the ContinuousSet(s)
3. Update Model Components
4. Add Discretization Equations
5. Return Discretized Model

If a user would like to create a custom finite difference scheme then they only have to worry about step (4) in the framework. The discretization equations for a particular scheme have been isolated from of the rest of the code for implementing the transformation. The function containing these discretization equations can be found at the top of the source code file for the transformation. For example, below is the function for the forward difference method:

```
def _forward_transform(v,s):
"""
Applies the Forward Difference formula of order O(h) for first derivatives
"""
    def _fwd_fun(i):
        tmp = sorted(s)
        idx = tmp.index(i)
        return 1/(tmp[idx+1]-tmp[idx])*(v(tmp[idx+1])-v(tmp[idx]))
    return _fwd_fun
```

In this function, 'v' represents the continuous variable or function that the method is being applied to. 's' represents the set of discrete points in the continuous domain. In order to implement a custom finite difference method, a user would have to copy the above function and just replace the equation next to the first return statement with their method.

After implementing a custom finite difference method using the above function template, the only other change that must be made is to add the custom method to the 'all_schemes' dictionary in the `dae.finite_difference` class.

In the case of a custom collocation method, changes will have to be made in steps (2) and (4) of the transformation framework. In addition to implementing the discretization equations, the user would also have to ensure that the desired collocation points are added to the ContinuousSet being discretized.

Dynamic Model Simulation

The pyomo.dae Simulator class can be used to simulate systems of ODEs and DAEs. It provides an interface to integrators available in other Python packages.

Note

The pyomo.dae Simulator does not include integrators directly. The user must have at least one of the supported Python packages installed in order to use this class.

class `pyomo.dae.Simulator`(*m*, *package='scipy'*)

Simulator objects allow a user to simulate a dynamic model formulated using pyomo.dae.

Parameters

- **m** (*Pyomo Model*) – The Pyomo model to be simulated should be passed as the first argument
- **package** (*string*) – The Python simulator package to use. Currently 'scipy' and 'casadi' are the only supported packages

`get_variable_order`(*vartype=None*)

This function returns the ordered list of differential variable names. The order corresponds to the order being sent to the integrator function. Knowing the order allows users to provide initial conditions for the differential equations using a list or map the profiles returned by the simulate function to the Pyomo variables.

Parameters

vartype (*string* or None) – Optional argument for specifying the type of variables to return the order for. The default behavior is to return the order of the differential variables. 'time-varying' will return the order of all the time-dependent algebraic variables identified in the model. 'algebraic' will return the order of algebraic variables used in the most recent call to the simulate function. 'input' will return the order of the time-dependent algebraic variables that were treated as inputs in the most recent call to the simulate function.

Return type

list

`initialize_model`()

This function will initialize the model using the profile obtained from simulating the dynamic model.

`simulate`(*numpoints=None*, *tstep=None*, *integrator=None*, *varying_inputs=None*, *initcon=None*, *integrator_options=None*)

Simulate the model. Integrator-specific options may be specified as keyword arguments and will be passed on to the integrator.

Parameters

- **numpoints** (*int*) – The number of points for the profiles returned by the simulator. Default is 100
- **tstep** (*int or float*) – The time step to use in the profiles returned by the simulator. This is not the time step used internally by the integrators. This is an optional parameter that may be specified in place of 'numpoints'.
- **integrator** (*string*) – The string name of the integrator to use for simulation. The default is 'lsoda' when using Scipy and 'idas' when using CasADi
- **varying_inputs** (`pyomo.environ.Suffix`) – A `Suffix` object containing the piecewise constant profiles to be used for certain time-varying algebraic variables.
- **initcon** (*list of floats*) – The initial conditions for the the differential variables. This is an optional argument. If not specified then the simulator will use the current value of the differential variables at the lower bound of the ContinuousSet for the initial condition.
- **integrator_options** (*dict*) – Dictionary containing options that should be passed to the integrator. See the documentation for a specific integrator for a list of valid options.

Returns

The first return value is a 1D array of time points corresponding to the second return value which is a 2D array of the profiles for the simulated differential and algebraic variables.

Return type

numpy array, numpy array

Note

Any keyword options supported by the integrator may be specified as keyword options to the simulate function and will be passed to the integrator.

Supported Simulator Packages

The Simulator currently includes interfaces to SciPy and CasADi. ODE simulation is supported in both packages however, DAE simulation is only supported by CasADi. A list of available integrators for each package is given below. Please refer to the SciPy and CasADi documentation directly for the most up-to-date information about these packages and for more information about the various integrators and options.

SciPy Integrators:

- **'vode'** : Real-valued Variable-coefficient ODE solver, options for non-stiff and stiff systems
- **'zvode'** : Complex-values Variable-coefficient ODE solver, options for non-stiff and stiff systems
- **'lsoda'** : Real-values Variable-coefficient ODE solver, automatic switching of algorithms for non-stiff or stiff systems
- **'dopri5'** : Explicit runge-kutta method of order (4)5 ODE solver
- **'dop853'** : Explicit runge-kutta method of order 8(5,3) ODE solver

CasADi Integrators:

- **'cvodes'** : CVodes from the Sundials suite, solver for stiff or non-stiff ODE systems
- **'idas'** : IDAS from the Sundials suite, DAE solver
- **'collocation'** : Fixed-step implicit runge-kutta method, ODE/DAE solver
- **'rk'** : Fixed-step explicit runge-kutta method, ODE solver

Using the Simulator

We now show how to use the Simulator to simulate the following system of ODEs:

$$\frac{d\theta}{dt} = \omega$$
$$\frac{d\omega}{dt} = -b * \omega - c * sin(\theta)$$

We begin by formulating the model using pyomo.DAE

```
>>> m = pyo.ConcreteModel()

>>> m.t = ContinuousSet(bounds=(0.0, 10.0))

>>> m.b = pyo.Param(initialize=0.25)
>>> m.c = pyo.Param(initialize=5.0)

>>> m.omega = pyo.Var(m.t)
>>> m.theta = pyo.Var(m.t)

>>> m.domegadt = DerivativeVar(m.omega, wrt=m.t)
>>> m.dthetadt = DerivativeVar(m.theta, wrt=m.t)

Setting the initial conditions
>>> m.omega[0].fix(0.0)
>>> m.theta[0].fix(3.14 - 0.1)

>>> def _diffeq1(m, t):
...     return m.domegadt[t] == -m.b * m.omega[t] - m.c * pyo.sin(m.theta[t])
>>> m.diffeq1 = pyo.Constraint(m.t, rule=_diffeq1)

>>> def _diffeq2(m, t):
...     return m.dthetadt[t] == m.omega[t]
>>> m.diffeq2 = pyo.Constraint(m.t, rule=_diffeq2)
```

Notice that the initial conditions are set by *fixing* the values of `m.omega` and `m.theta` at t=0 instead of being specified as extra equality constraints. Also notice that the differential equations are specified without using `Constraint.Skip` to skip enforcement at t=0. The Simulator cannot simulate any constraints that contain if-statements in their construction rules.

To simulate the model you must first create a Simulator object. Building this object prepares the Pyomo model for simulation with a particular Python package and performs several checks on the model to ensure compatibility with the Simulator. Be sure to read through the list of limitations at the end of this section to understand the types of models supported by the Simulator.

```
>>> sim = Simulator(m, package='scipy')
```

After creating a Simulator object, the model can be simulated by calling the simulate function. Please see the API documentation for the `Simulator` for more information about the valid keyword arguments for this function.

```
>>> tsim, profiles = sim.simulate(numpoints=100, integrator='vode')
```

The `simulate` function returns numpy arrays containing time points and the corresponding values for the dynamic variable profiles.

***Simulator Limitations*:**

- Differential equations must be first-order and separable

- Model can only contain a single ContinuousSet
- Can't simulate constraints with if-statements in the construction rules
- Need to provide initial conditions for dynamic states by setting the value or using fix()

Specifying Time-Varying Inputs

The Simulator supports simulation of a system of ODE's or DAE's with time-varying parameters or control inputs. Time-varying inputs can be specified using a Pyomo Suffix. We currently only support piecewise constant profiles. For more complex inputs defined by a continuous function of time we recommend adding an algebraic variable and constraint to your model.

The profile for a time-varying input should be specified using a Python dictionary where the keys correspond to the switching times and the values correspond to the value of the input at a time point. A Suffix is then used to associate this dictionary with the appropriate Var or Param and pass the information to the Simulator. The code snippet below shows an example.

```
>>> m = pyo.ConcreteModel()

>>> m.t = ContinuousSet(bounds=(0.0, 20.0))

Time-varying inputs
>>> m.b = pyo.Var(m.t)
>>> m.c = pyo.Param(m.t, default=5.0)

>>> m.omega = pyo.Var(m.t)
>>> m.theta = pyo.Var(m.t)

>>> m.domegadt = DerivativeVar(m.omega, wrt=m.t)
>>> m.dthetadt = DerivativeVar(m.theta, wrt=m.t)

Setting the initial conditions
>>> m.omega[0] = 0.0
>>> m.theta[0] = 3.14 - 0.1

>>> def _diffeq1(m, t):
...     return m.domegadt[t] == -m.b[t] * m.omega[t] - \
...                              m.c[t] * pyo.sin(m.theta[t])
>>> m.diffeq1 = pyo.Constraint(m.t, rule=_diffeq1)

>>> def _diffeq2(m, t):
...     return m.dthetadt[t] == m.omega[t]
>>> m.diffeq2 = pyo.Constraint(m.t, rule=_diffeq2)

Specifying the piecewise constant inputs
>>> b_profile = {0: 0.25, 15: 0.025}
>>> c_profile = {0: 5.0, 7: 50}

Declaring a Pyomo Suffix to pass the time-varying inputs to the Simulator
>>> m.var_input = pyo.Suffix(direction=pyo.Suffix.LOCAL)
>>> m.var_input[m.b] = b_profile
>>> m.var_input[m.c] = c_profile

Simulate the model using scipy
```

(continues on next page)

(continued from previous page)

```
>>> sim = Simulator(m, package='scipy')
>>> tsim, profiles = sim.simulate(numpoints=100,
...                               integrator='vode',
...                               varying_inputs=m.var_input)
```

Note

The Simulator does not support multi-indexed inputs (i.e. if `m.b` in the above example was indexed by another set besides `m.t`)

Dynamic Model Initialization

Providing a good initial guess is an important factor in solving dynamic optimization problems. There are several model initialization tools under development in pyomo.DAE to help users initialize their models. These tools will be documented here as they become available.

From Simulation

The `Simulator` includes a function for initializing discretized dynamic optimization models using the profiles returned from the simulator. An example using this function is shown below

```
Simulate the model using scipy
>>> sim = Simulator(m, package='scipy')
>>> tsim, profiles = sim.simulate(numpoints=100, integrator='vode',
...                               varying_inputs=m.var_input)

Discretize the model using Orthogonal Collocation
>>> discretizer = pyo.TransformationFactory('dae.collocation')
>>> discretizer.apply_to(m, nfe=10, ncp=3)

Initialize the discretized model using the simulator profiles
>>> sim.initialize_model()
```

Note

A model must be simulated before it can be initialized using this function

3.2.3 Generalized Disjunctive Programming

The Pyomo.GDP modeling extension [PyomoGDP-proceedings] [PyomoGDP-paper] provides support for Generalized Disjunctive Programming (GDP) [RG94], an extension of Disjunctive Programming [Bal85] from the operations

research community to include nonlinear relationships. The classic form for a GDP is given by:

$$\begin{aligned}
\min\quad & f(x,z) \\
\text{s.t.}\quad & Ax + Bz \leq d \\
& g(x,z) \leq 0 \\
& \bigvee_{i\in D_k} \begin{bmatrix} Y_{ik} \\ M_{ik}x + N_{ik}z \leq e_{ik} \\ r_{ik}(x,z) \leq 0 \end{bmatrix} \quad k \in K \\
& \Omega(Y) = True \\
& x \in X \subseteq \mathbb{R}^n \\
& Y \in \{True, False\}^p \\
& z \in Z \subseteq \mathbb{Z}^m
\end{aligned}$$

Here, we have the minimization of an objective $f(x, z)$ subject to global linear constraints $Ax + Bz \leq d$ and nonlinear constraints $g(x, z) \leq 0$, with conditional linear constraints $M_{ik}x + N_{ik}z \leq e_{ik}$ and nonlinear constraints $r_{ik}(x, z) \leq 0$. These conditional constraints are collected into disjuncts D_k, organized into disjunctions K. Finally, there are logical propositions $\Omega(Y) = True$. Decision/state variables can be continuous x, Boolean Y, and/or integer z.

GDP is useful to model discrete decisions that have implications on the system behavior [GT13]. For example, in process design, a disjunction may model the choice between processes A and B. If A is selected, then its associated equations and inequalities will apply; otherwise, if B is selected, then its respective constraints should be enforced.

Modelers often ask to model if-then-else relationships. These can be expressed as a disjunction as follows:

$$\begin{bmatrix} Y_1 \\ \text{constraints} \\ \text{for } \textit{then} \end{bmatrix} \vee \begin{bmatrix} Y_2 \\ \text{constraints} \\ \text{for } \textit{else} \end{bmatrix}$$
$$Y_1 \veebar Y_2$$

Here, if the Boolean Y_1 is `True`, then the constraints in the first disjunct are enforced; otherwise, the constraints in the second disjunct are enforced. The following sections describe the key concepts, modeling, and solution approaches available for Generalized Disjunctive Programming.

Key Concepts

Generalized Disjunctive Programming (GDP) provides a way to bridge high-level propositional logic and algebraic constraints. The GDP standard form from the *index page* is repeated below.

$$\begin{aligned}
\min\quad & f(x,z) \\
\text{s.t.}\quad & Ax + Bz \leq d \\
& g(x,z) \leq 0 \\
& \bigvee_{i\in D_k} \begin{bmatrix} Y_{ik} \\ M_{ik}x + N_{ik}z \leq e_{ik} \\ r_{ik}(x,z) \leq 0 \end{bmatrix} \quad k \in K \\
& \Omega(Y) = True \\
& x \in X \subseteq \mathbb{R}^n \\
& Y \in \{True, False\}^p \\
& z \in Z \subseteq \mathbb{Z}^m
\end{aligned}$$

Original support in Pyomo.GDP focused on the disjuncts and disjunctions, allowing the modelers to group relational expressions in disjuncts, with disjunctions describing logical-OR relationships between the groupings. As a result, we implemented the `Disjunct` and `Disjunction` objects before `BooleanVar` and the rest of the logical expression system. Accordingly, we also describe the disjuncts and disjunctions first below.

Disjuncts

Disjuncts represent groupings of relational expressions (e.g. algebraic constraints) summarized by a Boolean indicator variable Y through implication:

$$\left.\begin{aligned}Y_{ik} &\Rightarrow & M_{ik}x + N_{ik}z &\leq e_{ik}\\ Y_{ik} &\Rightarrow & r_{ik}(x,z) &\leq 0\end{aligned}\right. \qquad \forall i \in D_k, \forall k \in K$$

Logically, this means that if $Y_{ik} = True$, then the constraints $M_{ik}x + N_{ik}z \leq e_{ik}$ and $r_{ik}(x,z) \leq 0$ must be satisfied. However, if $Y_{ik} = False$, then the corresponding constraints are ignored. Note that $Y_{ik} = False$ does **not** imply that the corresponding constraints are *violated*.

Disjunctions

Disjunctions describe a logical *OR* relationship between two or more Disjuncts. The simplest and most common case is a 2-term disjunction:

$$\begin{bmatrix}Y_1\\ \exp(x_2) - 1 = x_1\\ x_3 = x_4 = 0\end{bmatrix} \bigvee \begin{bmatrix}Y_2\\ \exp\left(\frac{x_4}{1.2}\right) - 1 = x_3\\ x_1 = x_2 = 0\end{bmatrix}$$

The disjunction above describes the selection between two units in a process network. Y_1 and Y_2 are the Boolean variables corresponding to the selection of process units 1 and 2, respectively. The continuous variables x_1, x_2, x_3, x_4 describe flow in and out of the first and second units, respectively. If a unit is selected, the nonlinear equality in the corresponding disjunct enforces the input/output relationship in the selected unit. The final equality in each disjunct forces flows for the absent unit to zero.

Boolean Variables

Boolean variables are decision variables that may take a value of `True` or `False`. These are most often encountered as the indicator variables of disjuncts. However, they can also be independently defined to represent other problem decisions.

Note

Boolean variables are not intended to participate in algebraic expressions. That is, $3 \times \text{True}$ does not make sense; hence, $x = 3Y_1$ does not make sense. Instead, you may have the disjunction

$$\begin{bmatrix}Y_1\\ x = 3\end{bmatrix} \bigvee \begin{bmatrix}\neg Y_1\\ x = 0\end{bmatrix}$$

Logical Propositions

Logical propositions are constraints describing relationships between the Boolean variables in the model.

These logical propositions can include:

Operator	Example	Y_1	Y_2	Result
Negation	$\neg Y_1$			
		`True`		`False`
		`False`		`True`
Equivalence	$Y_1 \Leftrightarrow Y_2$			
		`True`	`True`	`True`
		`True`	`False`	`False`
		`False`	`True`	`False`
		`False`	`False`	`True`
Conjunction	$Y_1 \wedge Y_2$			
		`True`	`True`	`True`
		`True`	`False`	`False`
		`False`	`True`	`False`
		`False`	`False`	`False`
Disjunction	$Y_1 \vee Y_2$			
		`True`	`True`	`True`
		`True`	`False`	`True`
		`False`	`True`	`True`
		`False`	`False`	`False`
Exclusive OR	$Y_1 \veebar Y_2$			
		`True`	`True`	`False`
		`True`	`False`	`True`
		`False`	`True`	`True`
		`False`	`False`	`False`
Implication	$Y_1 \Rightarrow Y_2$			
		`True`	`True`	`True`
		`True`	`False`	`False`
		`False`	`True`	`True`
		`False`	`False`	`True`

Modeling in Pyomo.GDP

Disjunctions

To demonstrate modeling with disjunctions in Pyomo.GDP, we revisit the small example from *the previous page*.

$$\begin{bmatrix} Y_1 \\ \exp(x_2) - 1 = x_1 \\ x_3 = x_4 = 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_2 \\ \exp\left(\frac{x_4}{1.2}\right) - 1 = x_3 \\ x_1 = x_2 = 0 \end{bmatrix}$$

Explicit syntax: more descriptive

Pyomo.GDP explicit syntax (see below) provides more clarity in the declaration of each modeling object, and gives the user explicit control over the `Disjunct` names. Assuming the `ConcreteModel` object `m` and variables have been defined, lines 1 and 5 declare the `Disjunct` objects corresponding to selection of unit 1 and 2, respectively. Lines 2 and 6 define the input-output relations for each unit, and lines 3-4 and 7-8 enforce zero flow through the unit that is not selected. Finally, line 9 declares the logical disjunction between the two disjunctive terms.

```
m.unit1 = Disjunct()
m.unit1.inout = Constraint(expr=exp(m.x[2]) - 1 == m.x[1])
m.unit1.no_unit2_flow1 = Constraint(expr=m.x[3] == 0)
m.unit1.no_unit2_flow2 = Constraint(expr=m.x[4] == 0)
m.unit2 = Disjunct()
m.unit2.inout = Constraint(expr=exp(m.x[4] / 1.2) - 1 == m.x[3])
m.unit2.no_unit1_flow1 = Constraint(expr=m.x[1] == 0)
m.unit2.no_unit1_flow2 = Constraint(expr=m.x[2] == 0)
m.use_unit1or2 = Disjunction(expr=[m.unit1, m.unit2])
```

The indicator variables for each disjunct Y_1 and Y_2 are automatically generated by Pyomo.GDP, accessible via `m.unit1.indicator_var` and `m.unit2.indicator_var`.

Compact syntax: more concise

For more advanced users, a compact syntax is also available below, taking advantage of the ability to declare disjuncts and constraints implicitly. When the `Disjunction` object constructor is passed a list of lists, the outer list defines the disjuncts and the inner list defines the constraint expressions associated with the respective disjunct.

```
m.use1or2 = Disjunction(expr=[
    # First disjunct
    [exp(m.x[2])-1 == m.x[1],
     m.x[3] == 0, m.x[4] == 0],
    # Second disjunct
    [exp(m.x[4]/1.2)-1 == m.x[3],
     m.x[1] == 0, m.x[2] == 0]])
```

Note

By default, Pyomo.GDP `Disjunction` objects enforce an implicit "exactly one" relationship among the selection of the disjuncts (generalization of exclusive-OR). That is, exactly one of the `Disjunct` indicator variables should take a `True` value. This can be seen as an implicit logical proposition, in our example, $Y_1 \veebar Y_2$.

Logical Propositions

Pyomo.GDP also supports the use of logical propositions through the use of the `BooleanVar` and `LogicalConstraint` objects. The `BooleanVar` object in Pyomo represents Boolean variables, analogous to `Var` for numeric variables. `BooleanVar` can be indexed over a Pyomo `Set`, as below:

```
>>> m = ConcreteModel()
>>> m.my_set = RangeSet(4)
>>> m.Y = BooleanVar(m.my_set)
>>> m.Y.display()
Y : Size=4, Index=my_set
    Key : Value : Fixed : Stale
      1 :  None : False :  True
      2 :  None : False :  True
      3 :  None : False :  True
      4 :  None : False :  True
```

Using these Boolean variables, we can define `LogicalConstraint` objects, analogous to algebraic `Constraint` objects.

```
>>> m.p = LogicalConstraint(expr=m.Y[1].implies(m.Y[2] & m.Y[3]) | m.Y[4])
>>> m.p.pprint()
p : Size=1, Index=None, Active=True
    Key  : Body                           : Active
    None : (Y[1] --> Y[2] ∧ Y[3]) ∨ Y[4] :   True
```

Supported Logical Operators

Pyomo.GDP logical expression system supported operators and their usage are listed in the table below.

Operator	Operator	Method	Function
Negation	`~Y[1]`		`lnot(Y[1])`
Conjunction	`Y[1] & Y[2]`	`Y[1].land(Y[2])`	`land(Y[1],Y[2])`
Disjunction	`Y[1] | Y[2]`	`Y[1].lor(Y[2])`	`lor(Y[1],Y[2])`
Exclusive OR	`Y[1] ^ Y[2]`	`Y[1].xor(Y[2])`	`xor(Y[1], Y[2])`
Implication		`Y[1].implies(Y[2])`	`implies(Y[1], Y[2])`
Equivalence		`Y[1].equivalent_to(Y[2])`	`equivalent(Y[1], Y[2])`

Note

We omit support for some infix operators, e.g. `Y[1] >> Y[2]`, due to concerns about non-intuitive Python operator precedence. That is `Y[1] | Y[2] >> Y[3]` would translate to $Y_1 \lor (Y_2 \Rightarrow Y_3)$ rather than $(Y_1 \lor Y_2) \Rightarrow Y_3$

In addition, the following constraint-programming-inspired operators are provided: `exactly`, `atmost`, and `atleast`. These predicates enforce, respectively, that exactly, at most, or at least N of their `BooleanVar` arguments are `True`.

Usage:

- `atleast(3, Y[1], Y[2], Y[3])`
- `atmost(3, Y)`
- `exactly(3, Y)`

```
>>> m = ConcreteModel()
>>> m.my_set = RangeSet(4)
>>> m.Y = BooleanVar(m.my_set)
>>> m.p = LogicalConstraint(expr=atleast(3, m.Y))
>>> m.p.pprint()
p : Size=1, Index=None, Active=True
    Key  : Body                                 : Active
    None : atleast(3: [Y[1], Y[2], Y[3], Y[4]]) :   True
>>> TransformationFactory('core.logical_to_linear').apply_to(m)
>>> # constraint auto-generated by transformation
>>> m.logic_to_linear.transformed_constraints.pprint()
transformed_constraints : Size=1, Index={1}, Active=True
    Key : Lower : Body                                                          : Upper␣
↪: Active
      1 :   3.0 : Y_asbinary[1] + Y_asbinary[2] + Y_asbinary[3] + Y_asbinary[4] :  +Inf␣
↪:   True
```

We elaborate on the `logical_to_linear` transformation *on the next page*.

Indexed logical constraints

Like `Constraint` objects for algebraic expressions, `LogicalConstraint` objects can be indexed. An example of this usage may be found below for the expression:

$$Y_{i+1} \Rightarrow Y_i, \quad i \in \{1, 2, \ldots, n-1\}$$

```
>>> m = ConcreteModel()
>>> n = 5
>>> m.I = RangeSet(n)
>>> m.Y = BooleanVar(m.I)

>>> @m.LogicalConstraint(m.I)
... def p(m, i):
...     return m.Y[i+1].implies(m.Y[i]) if i < n else Constraint.Skip

>>> m.p.pprint()
p : Size=4, Index=I, Active=True
    Key : Body           : Active
      1 : Y[2] --> Y[1] :   True
      2 : Y[3] --> Y[2] :   True
      3 : Y[4] --> Y[3] :   True
      4 : Y[5] --> Y[4] :   True
```

Integration with Disjunctions

Note

Historically, the `indicator_var` on `Disjunct` objects was implemented as a binary `Var`. Beginning in Pyomo 6.0, that has been changed to the more mathematically correct `BooleanVar`, with the associated binary variable available as `binary_indicator_var`.

The logical expression system is designed to augment the previously introduced `Disjunct` and `Disjunction` components. Mathematically, the disjunct indicator variable is Boolean, and can be used directly in logical propositions.

Here, we demonstrate this capability with a toy example:

$$
\begin{aligned}
\min \quad & x \\
\text{s.t.} \quad & \begin{bmatrix} Y_1 \\ x \geq 2 \end{bmatrix} \vee \begin{bmatrix} Y_2 \\ x \geq 3 \end{bmatrix} \\
& \begin{bmatrix} Y_3 \\ x \leq 8 \end{bmatrix} \vee \begin{bmatrix} Y_4 \\ x = 2.5 \end{bmatrix} \\
& Y_1 \veebar Y_2 \\
& Y_3 \veebar Y_4 \\
& Y_1 \Rightarrow Y_4
\end{aligned}
$$

```
>>> m = ConcreteModel()
>>> m.s = RangeSet(4)
>>> m.ds = RangeSet(2)
>>> m.d = Disjunct(m.s)
>>> m.djn = Disjunction(m.ds)
>>> m.djn[1] = [m.d[1], m.d[2]]
>>> m.djn[2] = [m.d[3], m.d[4]]
>>> m.x = Var(bounds=(-2, 10))
>>> m.d[1].c = Constraint(expr=m.x >= 2)
>>> m.d[2].c = Constraint(expr=m.x >= 3)
>>> m.d[3].c = Constraint(expr=m.x <= 8)
>>> m.d[4].c = Constraint(expr=m.x == 2.5)
>>> m.o = Objective(expr=m.x)

>>> # Add the logical proposition
>>> m.p = LogicalConstraint(
...    expr=m.d[1].indicator_var.implies(m.d[4].indicator_var))
>>> # Note: the implicit XOR enforced by m.djn[1] and m.djn[2] still apply

>>> # Apply the Big-M reformulation: It will convert the logical
>>> # propositions to algebraic expressions.
>>> TransformationFactory('gdp.bigm').apply_to(m)

>>> # Before solve, Boolean vars have no value
>>> Reference(m.d[:].indicator_var).display()
IndexedBooleanVar : Size=4, Index=s, ReferenceTo=d[:].indicator_var
    Key : Value : Fixed : Stale
      1 :  None : False :  True
      2 :  None : False :  True
      3 :  None : False :  True
      4 :  None : False :  True

>>> # Solve the reformulated model
>>> run_data = SolverFactory('glpk').solve(m)
>>> Reference(m.d[:].indicator_var).display()
IndexedBooleanVar : Size=4, Index=s, ReferenceTo=d[:].indicator_var
    Key : Value : Fixed : Stale
      1 :  True : False : False
      2 : False : False : False
```

(continues on next page)

(continued from previous page)

```
    3 : False : False : False
    4 :  True : False : False
```

Advanced LogicalConstraint Examples

Support for complex nested expressions is a key benefit of the logical expression system. Below are examples of expressions that we support, and with some, an explanation of their implementation.

Composition of standard operators

$$Y_1 \vee Y_2 \implies Y_3 \wedge \neg Y_4 \wedge (Y_5 \vee Y_6)$$

```
m.p = LogicalConstraint(expr=(m.Y[1] | m.Y[2]).implies(
    m.Y[3] & ~m.Y[4] & (m.Y[5] | m.Y[6]))
)
```

Expressions within CP-type operators

$$\text{atleast}(3, Y_1, Y_2 \vee Y_3, Y_4 \Rightarrow Y_5, Y_6)$$

Here, augmented variables may be automatically added to the model as follows:

$$\begin{aligned} \text{atleast}(3, Y_1, Y_A, Y_B, Y_6) \\ Y_A \Leftrightarrow Y_2 \vee Y_3 \\ Y_B \Leftrightarrow (Y_4 \Rightarrow Y_5) \end{aligned}$$

```
m.p = LogicalConstraint(
    expr=atleast(3, m.Y[1], Or(m.Y[2], m.Y[3]), m.Y[4].implies(m.Y[5]), m.Y[6]))
```

Nested CP-style operators

$$\text{atleast}(2, Y_1, \text{exactly}(2, Y_2, Y_3, Y_4), Y_5, Y_6)$$

Here, we again need to add augmented variables:

$$\begin{aligned} \text{atleast}(2, Y_1, Y_A, Y_5, Y_6) \\ Y_A \Leftrightarrow \text{exactly}(2, Y_2, Y_3, Y_4) \end{aligned}$$

However, we also need to further interpret the second statement as a disjunction:

$$\text{atleast}(2, Y_1, Y_A, Y_5, Y_6)$$

$$\begin{bmatrix} Y_A \\ \text{exactly}(2, Y_2, Y_3, Y_4) \end{bmatrix} \vee \begin{bmatrix} \neg Y_A \\ \begin{bmatrix} Y_B \\ \text{atleast}(3, Y_2, Y_3, Y_4) \end{bmatrix} \vee \begin{bmatrix} Y_C \\ \text{atmost}(1, Y_2, Y_3, Y_4) \end{bmatrix} \end{bmatrix}$$

or equivalently,

$$\text{atleast}(2, Y_1, Y_A, Y_5, Y_6)$$
$$\text{exactly}(1, Y_A, Y_B, Y_C)$$
$$\begin{bmatrix} Y_A \\ \text{exactly}(2, Y_2, Y_3, Y_4) \end{bmatrix} \vee \begin{bmatrix} Y_B \\ \text{atleast}(3, Y_2, Y_3, Y_4) \end{bmatrix} \vee \begin{bmatrix} Y_C \\ \text{atmost}(1, Y_2, Y_3, Y_4) \end{bmatrix}$$

```
m.p = LogicalConstraint(
    expr=atleast(2, m.Y[1], exactly(2, m.Y[2], m.Y[3], m.Y[4]), m.Y[5], m.Y[6]))
```

In the `logical_to_linear` transformation, we automatically convert these special disjunctions to linear form using a Big M reformulation.

Additional Examples

The following models all work and are equivalent for $[x = 0] \veebar [y = 0]$:

```
Option 1: Rule-based construction

>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunct, Disjunction
>>> model = pyo.ConcreteModel()

>>> model.x = pyo.Var()
>>> model.y = pyo.Var()

>>> # Two conditions
>>> def _d(disjunct, flag):
...     model = disjunct.model()
...     if flag:
...         # x == 0
...         disjunct.c = pyo.Constraint(expr=model.x == 0)
...     else:
...         # y == 0
...         disjunct.c = pyo.Constraint(expr=model.y == 0)
>>> model.d = Disjunct([0,1], rule=_d)

>>> # Define the disjunction
>>> def _c(model):
...     return [model.d[0], model.d[1]]
>>> model.c = Disjunction(rule=_c)

Option 2: Explicit disjuncts

>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunct, Disjunction
>>> model = pyo.ConcreteModel()

>>> model.x = pyo.Var()
>>> model.y = pyo.Var()

>>> model.fix_x = Disjunct()
```

(continues on next page)

(continued from previous page)

```
>>> model.fix_x.c = pyo.Constraint(expr=model.x == 0)

>>> model.fix_y = Disjunct()
>>> model.fix_y.c = pyo.Constraint(expr=model.y == 0)

>>> model.c = Disjunction(expr=[model.fix_x, model.fix_y])

Option 3: Implicit disjuncts (disjunction rule returns a list of
expressions or a list of lists of expressions)

>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunction
>>> model = pyo.ConcreteModel()

>>> model.x = pyo.Var()
>>> model.y = pyo.Var()

>>> model.c = Disjunction(expr=[model.x == 0, model.y == 0])
```

Solving Logic-based Models with Pyomo.GDP

Flexible Solution Suite

Once a model is formulated as a GDP model, a range of solution strategies are available to manipulate and solve it.

The traditional approach is reformulation to a MI(N)LP, but various other techniques are possible, including direct solution via the *GDPopt solver*. Below, we describe some of these capabilities.

Reformulations

Logical constraints

Note

Historically users needed to explicitly convert logical propositions to algebraic form prior to invoking the GDP MI(N)LP reformulations or the GDPopt solver. However, this is mathematically incorrect since the GDP MI(N)LP reformulations themselves convert logical formulations to algebraic formulations. The current recommended practice is to pass the entire (mixed logical / algebraic) model to the MI(N)LP reformulations or GDPopt directly.

There are several approaches to convert logical constraints into algebraic form.

Conjunctive Normal Form

The first transformation (*core.logical_to_linear*) leverages the *sympy* package to generate the conjunctive normal form of the logical constraints and then adds the equivalent as a list algebraic constraints. The following transforms logical propositions on the model to algebraic form:

```
TransformationFactory('core.logical_to_linear').apply_to(model)
```

The transformation creates a constraint list with a unique name starting with `logic_to_linear`, within which the algebraic equivalents of the logical constraints are placed. If not already associated with a binary variable, each `BooleanVar` object will receive a generated binary counterpart. These associated binary variables may be accessed via the `get_associated_binary()` method.

```
m.Y[1].get_associated_binary()
```

Additional augmented variables and their corresponding constraints may also be created, as described in *Advanced LogicalConstraint Examples*.

Following solution of the GDP model, values of the Boolean variables may be updated from their algebraic binary counterparts using the `update_boolean_vars_from_binary()` function.

`pyomo.core.plugins.transform.logical_to_linear.`**`update_boolean_vars_from_binary`**`(`*model*, *integer_tolerance=1e-05*`)`

Updates all Boolean variables based on the value of their linked binary variables.

Factorable Programming

The second transformation (*contrib.logical_to_disjunctive*) leverages ideas from factorable programming to first generate an equivalent set of "factored" logical constraints form by traversing each logical proposition and replacing each logical operator with an additional Boolean variable and then adding the "simple" logical constraint that equates the new Boolean variable with the single logical operator.

The resulting "simple" logical constraints are converted to either MIP or GDP form: if the constraint contains only Boolean variables, then then MIP representation is emitted. Logical constraints with mixed integer-Boolean arguments (e.g., *atmost*, *atleast*, *exactly*, etc.) are converted to a disjunctive representation.

As this transformation both avoids the conversion into *sympy* and only requires a single traversal of each logical constraint, *contrib.logical_to_disjunctive* is significantly faster than *core.logical_to_linear* at the cost of a larger model. In practice, the cost of the larger model is negated by the effectiveness of the MIP presolve in most solvers.

Reformulation to MI(N)LP

To use standard commercial solvers, you must convert the disjunctive model to a standard MILP/MINLP model. The two classical strategies for doing so are the (included) Big-M and Hull reformulations.

Big-M (BM) Reformulation

The Big-M reformulation[NW88] results in a smaller transformed model, avoiding the need to add extra variables; however, it yields a looser continuous relaxation. By default, the BM transformation will estimate reasonably tight M values for you if variables are bounded. For nonlinear models where finite expression bounds may be inferred from variable bounds, the BM transformation may also be able to automatically compute M values for you. For all other models, you will need to provide the M values through a "BigM" Suffix, or through the *bigM* argument to the transformation. We will raise a `GDP_Error` for missing M values.

To apply the BM reformulation within a python script, use:

```
TransformationFactory('gdp.bigm').apply_to(model)
```

From the Pyomo command line, include the `--transform pyomo.gdp.bigm` option.

Multiple Big-M (MBM) Reformulation

We also implement the multiple-parameter Big-M (MBM) approach described in literature[TG15]. By default, the MBM transformation will solve continuous subproblems in order to calculate M values. This process can be time-consuming, so the transformation also provides a method to export the M values used as a dictionary and allows for M values to be provided through the *bigM* argument.

For example, to apply the transformation and store the M values, use:

```
mbigm = TransformationFactory('gdp.mbigm')
mbigm.apply_to(model)

# These can be stored...
M_values = mbigm.get_all_M_values(model)
# ...so that in future runs, you can write:
mbigm.apply_to(m, bigM=M_values)
```

From the Pyomo command line, include the `--transform pyomo.gdp.mbigm` option.

Warning

The Multiple Big-M transformation does not currently support Suffixes and will ignore "BigM" Suffixes.

Hull Reformulation (HR)

The Hull Reformulation requires a lifting into a higher-dimensional space and consequently introduces disaggregated variables and their corresponding constraints.

Note

- All variables that appear in disjuncts need upper and lower bounds.
- The hull reformulation is an exact reformulation at the solution points even for nonconvex GDP models, but the resulting MINLP will also be nonconvex.

To apply the Hull reformulation within a python script, use:

```
TransformationFactory('gdp.hull').apply_to(model)
```

From the Pyomo command line, include the `--transform pyomo.gdp.hull` option.

Hybrid BM/HR Reformulation

An experimental (for now) implementation of the cutting plane approach described in literature[SG03] is provided for linear GDP models. The transformation augments the BM reformulation by a set of cutting planes generated from the HR model by solving separation problems. This gives a model that is not as large as the HR, but with a stronger continuous relaxation than the BM.

This transformation is accessible via:

```
TransformationFactory('gdp.cuttingplane').apply_to(model)
```

Direct GDP solvers

Pyomo includes the contributed GDPopt solver, which can directly solve GDP models. Its usage is described within the *contributed packages documentation*.

3.2.4 MPEC

`pyomo.mpec` supports modeling complementarity conditions and optimization problems with equilibrium constraints.

3.2.5 Pyomo Network

Pyomo Network is a package that allows users to easily represent their model as a connected network of units. Units are blocks that contain ports, which contain variables, that are connected to other ports via arcs. The connection of two ports to each other via an arc typically represents a set of constraints equating each member of each port to each other, however there exist other connection rules as well, in addition to support for custom rules. Pyomo Network also includes a model transformation that will automatically expand the arcs and generate the appropriate constraints to produce an algebraic model that a solver can handle. Furthermore, the package also introduces a generic sequential decomposition tool that can leverage the modeling components to decompose a model and compute each unit in the model in a logically ordered sequence.

Modeling Components

Pyomo Network introduces two new modeling components to Pyomo:

`pyomo.network.Port`	A collection of variables, which may be connected to other ports
`pyomo.network.Arc`	Component used for connecting the members of two Port objects

Port

class pyomo.network.**Port**(**args*, ***kwds*)

A collection of variables, which may be connected to other ports

The idea behind Ports is to create a bundle of variables that can be manipulated together by connecting them to other ports via Arcs. A preprocess transformation will look for Arcs and expand them into a series of constraints that involve the original variables contained within the Port. The way these constraints are built can be specified for each Port member when adding members to the port, but by default the Port members will be equated to each other. Additionally, other objects such as expressions can be added to Ports as long as they, or their indexed members, can be manipulated within constraint expressions.

Parameters

- **rule** (*function*) – A function that returns a dict of (name: var) pairs to be initially added to the Port. Instead of var it could also be a tuples of (var, rule). Or it could return an iterable of either vars or tuples of (var, rule) for implied names.
- **initialize** – Follows same specifications as rule's return value, gets initially added to the Port
- **implicit** – An iterable of names to be initially added to the Port as implicit vars
- **extends** (*Port*) – A Port whose vars will be added to this Port upon construction

static Equality(*port*, *name*, *index_set*)

Arc Expansion procedure to generate simple equality constraints

static Extensive(*port*, *name*, *index_set*, *include_splitfrac=None*, *write_var_sum=True*)

Arc Expansion procedure for extensive variable properties

This procedure is the rule to use when variable quantities should be conserved; that is, split for outlets and combined for inlets.

This will first go through every destination of the port (i.e., arcs whose source is this Port) and create a new variable on the arc's expanded block of the same index as the current variable being processed to store the amount of the variable that flows over the arc. For ports that have multiple outgoing arcs, this procedure will create a single splitfrac variable on the arc's expanded block as well. Then it will generate constraints for the new variable that relate it to the port member variable using the split fraction, ensuring that all extensive variables in the Port are split using the same ratio. The generation of the split fraction variable and constraint can be suppressed by setting the *include_splitfrac* argument to *False*.

Once all arc-specific variables are created, this procedure will create the "balancing constraint" that ensures that the sum of all the new variables equals the original port member variable. This constraint can be suppressed by setting the *write_var_sum* argument to *False*; in which case, a single constraint will be written that states the sum of the split fractions equals 1.

Finally, this procedure will go through every source for this port and create a new arc variable (unless it already exists), before generating the balancing constraint that ensures the sum of all the incoming new arc variables equals the original port variable.

Model simplifications:

> If the port has a 1-to-1 connection on either side, it will not create the new variables and instead write a simple equality constraint for that side.
>
> If the outlet side is not 1-to-1 but there is only one outlet, it will not create a splitfrac variable or write the split constraint, but it will still write the outsum constraint which will be a simple equality.
>
> If the port only contains a single Extensive variable, the splitfrac variables and the splitting constraints will be skipped since they will be unnecessary. However, they can be still included by passing *include_splitfrac=True*.

Note

If split fractions are skipped, the *write_var_sum=False* option is not allowed.

class pyomo.network.port.**_PortData**(**args*, ***kwargs*)

The following code snippet shows examples of declaring and using a `Port` component on a concrete Pyomo model:

```
>>> import pyomo.environ as pyo
>>> from pyomo.network import Port
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var(['a', 'b']) # can be indexed
>>> m.z = pyo.Var()
>>> m.e = 5 * m.z # you can add Pyomo expressions too
>>> m.w = pyo.Var()

>>> m.p = Port()
>>> m.p.add(m.x) # implicitly name the port member "x"
>>> m.p.add(m.y, "foo") # name the member "foo"
```

(continues on next page)

(continued from previous page)

```
>>> m.p.add(m.e, rule=Port.Extensive) # specify a rule
>>> m.p.add(m.w, rule=Port.Extensive, write_var_sum=False) # keyword arg
```

Arc

class pyomo.network.**Arc**(**args*, ***kwds*)

Component used for connecting the members of two Port objects

Parameters

- **source** (*Port*) – A single Port for a directed arc. Aliases to src.
- **destination** (*Port*) – A single`Port for a directed arc. Aliases to dest.
- **ports** – A two-member list or tuple of single Ports for an undirected arc
- **directed** (*bool*) – Set True for directed. Use along with *rule* to be able to return an implied (source, destination) tuple.
- **rule** (*function*) – A function that returns either a dictionary of the arc arguments or a two-member iterable of ports

class pyomo.network.arc.**_ArcData**(**args*, ***kwargs*)

The following code snippet shows examples of declaring and using an `Arc` component on a concrete Pyomo model:

```
>>> import pyomo.environ as pyo
>>> from pyomo.network import Port, Arc
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var(['a', 'b'])
>>> m.u = pyo.Var()
>>> m.v = pyo.Var(['a', 'b'])
>>> m.w = pyo.Var()
>>> m.z = pyo.Var(['a', 'b']) # indexes need to match

>>> m.p = Port(initialize=[m.x, m.y])
>>> m.q = Port(initialize={"x": m.u, "y": m.v})
>>> m.r = Port(initialize={"x": m.w, "y": m.z}) # names need to match
>>> m.a = Arc(source=m.p, destination=m.q) # directed
>>> m.b = Arc(ports=(m.p, m.q)) # undirected
>>> m.c = Arc(ports=(m.p, m.q), directed=True) # directed
>>> m.d = Arc(src=m.p, dest=m.q) # aliases work
>>> m.e = Arc(source=m.r, dest=m.p) # ports can have both in and out
```

Arc Expansion Transformation

The examples above show how to declare and instantiate a `Port` and an `Arc`. These two components form the basis of the higher level representation of a connected network with sets of related variable quantities. Once a network model has been constructed, Pyomo Network implements a transformation that will expand all (active) arcs on the model and automatically generate the appropriate constraints. The constraints created for each port member will be indexed by the same indexing set as the port member itself.

During transformation, a new block is created on the model for each arc (located on the arc's parent block), which serves to contain all of the auto generated constraints for that arc. At the end of the transformation, a reference is created on the arc that points to this new block, available via the arc property *arc.expanded_block*.

The constraints produced by this transformation depend on the rule assigned for each port member and can be different between members on the same port. For example, you can have two different members on a port where one member's rule is `Port.Equality` and the other member's rule is `Port.Extensive`.

`Port.Equality` is the default rule for port members. This rule simply generates equality constraints on the expanded block between the source port's member and the destination port's member. Another implemented expansion method is `Port.Extensive`, which essentially represents implied splitting and mixing of certain variable quantities. Users can refer to the documentation of the static method itself for more details on how this implicit splitting and mixing is implemented. Additionally, should users desire, the expansion API supports custom rules that can be implemented to generate whatever is needed for special cases.

The following code demonstrates how to call the transformation to expand the arcs on a model:

```
>>> import pyomo.environ as pyo
>>> from pyomo.network import Port, Arc
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var(['a', 'b'])
>>> m.u = pyo.Var()
>>> m.v = pyo.Var(['a', 'b'])

>>> m.p = Port(initialize=[m.x, (m.y, Port.Extensive)]) # rules must match
>>> m.q = Port(initialize={"x": m.u, "y": (m.v, Port.Extensive)})
>>> m.a = Arc(source=m.p, destination=m.q)

>>> pyo.TransformationFactory("network.expand_arcs").apply_to(m)
```

Sequential Decomposition

Pyomo Network implements a generic `SequentialDecomposition` tool that can be used to compute each unit in a network model in a logically ordered sequence.

The sequential decomposition procedure is commenced via the `run` method.

Creating a Graph

To begin this procedure, the Pyomo Network model is first utilized to create a networkx *MultiDiGraph* by adding edges to the graph for every arc on the model, where the nodes of the graph are the parent blocks of the source and destination ports. This is done via the `create_graph` method, which requires all arcs on the model to be both directed and already expanded. The *MultiDiGraph* class of networkx supports both direccted edges as well as having multiple edges between the same two nodes, so users can feel free to connect as many ports as desired between the same two units.

Computation Order

The order of computation is then determined by treating the resulting graph as a tree, starting at the roots of the tree, and making sure by the time each node is reached, all of its predecessors have already been computed. This is implemented through the `calculation_order` and `tree_order` methods. Before this, however, the procedure will first select a set of tear edges, if necessary, such that every loop in the graph is torn, while minimizing both the number of times any single loop is torn as well as the total number of tears.

Tear Selection

A set of tear edges can be selected in one of two ways. By default, a Pyomo MIP model is created and optimized resulting in an optimal set of tear edges. The implementation of this MIP model is based on a set of binary "torn" variables for every edge in the graph, and constraints on every loop in the graph that dictate that there must be at least one tear on the loop. Then there are two objectives (represented by a doubly weighted objective). The primary

objective is to minimize the number of times any single loop is torn, and then secondary to that is to minimize the total number of tears. This process is implemented in the `select_tear_mip` method, which uses the model returned from the `select_tear_mip_model` method.

Alternatively, there is the `select_tear_heuristic` method. This uses a heuristic procedure that walks back and forth on the graph to find every optimal tear set, and returns each equally optimal tear set it finds. This method is much slower than the MIP method on larger models, but it maintains some use in the fact that it returns every possible optimal tear set.

A custom tear set can be assigned before calling the `run` method. This is useful so users can know what their tear set will be and thus what arcs will require guesses for uninitialized values. See the `set_tear_set` method for details.

Running the Sequential Decomposition Procedure

After all of this computational order preparation, the sequential decomposition procedure will then run through the graph in the order it has determined. Thus, the *function* that was passed to the `run` method will be called on every unit in sequence. This function can perform any arbitrary operations the user desires. The only thing that `SequentialDecomposition` expects from the function is that after returning from it, every variable on every outgoing port of the unit will be specified (i.e. it will have a set current value). Furthermore, the procedure guarantees to the user that for every unit, before the function is called, every variable on every incoming port of the unit will be fixed.

In between computing each of these units, port member values are passed across existing arcs involving the unit currently being computed. This means that after computing a unit, the expanded constraints from each arc coming out of this unit will be satisfied, and the values on the respective destination ports will be fixed at these new values. While running the computational order, values are not passed across tear edges, as tear edges represent locations in loops to stop computations (during iterations). This process continues until all units in the network have been computed. This concludes the “first pass run” of the network.

Guesses and Fixing Variables

When passing values across arcs while running the computational order, values at the destinations of each of these arcs will be fixed at the appropriate values. This is important to the fact that the procedure guarantees every inlet variable will be fixed before calling the function. However, since values are not passed across torn arcs, there is a need for user-supplied guesses for those values. See the `set_guesses_for` method for details on how to supply these values.

In addition to passing dictionaries of guesses for certain ports, users can also assign current values to the variables themselves and the procedure will pick these up and fix the variables in place. Alternatively, users can utilize the *default_guess* option to specify a value to use as a default guess for all free variables if they have no guess or current value. If a free variable has no guess or current value and there is no default guess option, then an error will be raised.

Similarly, if the procedure attempts to pass a value to a destination port member but that port member is already fixed and its fixed value is different from what is trying to be passed to it (by a tolerance specified by the *almost_equal_tol* option), then an error will be raised. Lastly, if there is more than one free variable in a constraint while trying to pass values across an arc, an error will be raised asking the user to fix more variables by the time values are passed across said arc.

Tear Convergence

After completing the first pass run of the network, the sequential decomposition procedure will proceed to converge all tear edges in the network (unless the user specifies not to, or if there are no tears). This process occurs separately for every strongly connected component (SCC) in the graph, and the SCCs are computed in a logical order such that each SCC is computed before other SCCs downstream of it (much like `tree_order`).

There are two implemented methods for converging tear edges: direct substitution and Wegstein acceleration. Both of these will iteratively run the computation order until every value in every tear arc has converged to within the specified tolerance. See the `SequentialDecomposition` parameter documentation for details on what can be controlled about this procedure.

The following code demonstrates basic usage of the `SequentialDecomposition` class:

```
>>> import pyomo.environ as pyo
>>> from pyomo.network import Port, Arc, SequentialDecomposition
>>> m = pyo.ConcreteModel()
>>> m.unit1 = pyo.Block()
>>> m.unit1.x = pyo.Var()
>>> m.unit1.y = pyo.Var(['a', 'b'])
>>> m.unit2 = pyo.Block()
>>> m.unit2.x = pyo.Var()
>>> m.unit2.y = pyo.Var(['a', 'b'])
>>> m.unit1.port = Port(initialize=[m.unit1.x, (m.unit1.y, Port.Extensive)])
>>> m.unit2.port = Port(initialize=[m.unit2.x, (m.unit2.y, Port.Extensive)])
>>> m.a = Arc(source=m.unit1.port, destination=m.unit2.port)
>>> pyo.TransformationFactory("network.expand_arcs").apply_to(m)

>>> m.unit1.x.fix(10)
>>> m.unit1.y['a'].fix(15)
>>> m.unit1.y['b'].fix(20)

>>> seq = SequentialDecomposition(tol=1.0E-3) # options can go to init
>>> seq.options.select_tear_method = "heuristic" # or set them like so
>>> # seq.set_tear_set([...]) # assign a custom tear set
>>> # seq.set_guesses_for(m.unit.inlet, {...}) # choose guesses
>>> def initialize(b):
...     # b.initialize()
...     pass
...
>>> seq.run(m, initialize)
```

class `pyomo.network.`**`SequentialDecomposition`**(***kwds*)

A sequential decomposition tool for Pyomo Network models

The following parameters can be set upon construction of this class or via the *options* attribute.

Parameters

- **graph** (*MultiDiGraph*) – A networkx graph representing the model to be solved.

  default=None (will compute it)

- **tear_set** (*list*) – A list of indexes representing edges to be torn. Can be set with a list of edge tuples via set_tear_set.

  default=None (will compute it)

- **select_tear_method** (*str*) – Which method to use to select a tear set, either "mip" or "heuristic".

  default="mip"

- **run_first_pass** (*bool*) – Boolean indicating whether or not to run through network before running the tear stream convergence procedure.

  default=True

- **solve_tears** (*bool*) – Boolean indicating whether or not to run iterations to converge tear streams.

  default=True

- **guesses** (*ComponentMap*) – ComponentMap of guesses to use for first pass (see set_guesses_for method).

  default=ComponentMap()

- **default_guess** (*float*) – Value to use if a free variable has no guess.

  default=None

- **almost_equal_tol** (*float*) – Difference below which numbers are considered equal when checking port value agreement.

  default=1.0E-8

- **log_info** (*bool*) – Set logger level to INFO during run.

  default=False

- **tear_method** (*str*) – Method to use for converging tear streams, either "Direct" or "Wegstein".

  default="Direct"

- **iterLim** (*int*) – Limit on the number of tear iterations.

  default=40

- **tol** (*float*) – Tolerance at which to stop tear iterations.

  default=1.0E-5

- **tol_type** (*str*) – Type of tolerance value, either "abs" (absolute) or "rel" (relative to current value).

  default="abs"

- **report_diffs** (*bool*) – Report the matrix of differences across tear streams for every iteration.

  default=False

- **accel_min** (*float*) – Min value for Wegstein acceleration factor.

  default=-5

- **accel_max** (*float*) – Max value for Wegstein acceleration factor.

  default=0

- **tear_solver** (*str*) – Name of solver to use for select_tear_mip.

  default="cplex"

- **tear_solver_io** (*str*) – Solver IO keyword for the above solver.

  default=None

- **tear_solver_options** (*dict*) – Keyword options to pass to solve method.

  default={}

calculation_order(*G*, *roots=None*, *nodes=None*)

Rely on tree_order to return a calculation order of nodes

Parameters

- **roots** – List of nodes to consider as tree roots, if None then the actual roots are used
- **nodes** – Subset of nodes to consider in the tree, if None then all nodes are used

`create_graph(model)`

Returns a networkx MultiDiGraph of a Pyomo network model

The nodes are units and the edges follow Pyomo Arc objects. Nodes that get added to the graph are determined by the parent blocks of the source and destination Ports of every Arc in the model. Edges are added for each Arc using the direction specified by source and destination. All Arcs in the model will be used whether or not they are active (since this needs to be done after expansion), and they all need to be directed.

`indexes_to_arcs(G, lst)`

Converts a list of edge indexes to the corresponding Arcs

Parameters

- `G` – A networkx graph corresponding to lst
- `lst` – A list of edge indexes to convert to tuples

Returns

A list of arcs

`run(model, function)`

Compute a Pyomo Network model using sequential decomposition

Parameters

- `model` – A Pyomo model
- `function` – A function to be called on each block/node in the network

`select_tear_heuristic(G)`

This finds optimal sets of tear edges based on two criteria. The primary objective is to minimize the maximum number of times any cycle is broken. The secondary criteria is to minimize the number of tears.

This function uses a branch and bound type approach.

Returns

- *tsets* – List of lists of tear sets. All the tear sets returned are equally good. There are often a very large number of equally good tear sets.
- *upperbound_loop* – The max number of times any single loop is torn
- *upperbound_total* – The total number of loops

Improvements for the future

I think I can improve the efficiency of this, but it is good enough for now. Here are some ideas for improvement:

1. Reduce the number of redundant solutions. It is possible to find tears sets [1,2] and [2,1]. I eliminate redundant solutions from the results, but they can occur and it reduces efficiency.

2. Look at strongly connected components instead of whole graph. This would cut back on the size of graph we are looking at. The flowsheets are rarely one strongly connected component.

3. When you add an edge to a tear set you could reduce the size of the problem in the branch by only looking at strongly connected components with that edge removed.

4. This returns all equally good optimal tear sets. That may not really be necessary. For very large flowsheets, there could be an extremely large number of optimal tear edge sets.

select_tear_mip(*G*, *solver*, *solver_io=None*, *solver_options={}*)

This finds optimal sets of tear edges based on two criteria. The primary objective is to minimize the maximum number of times any cycle is broken. The secondary criteria is to minimize the number of tears.

This function creates a MIP problem in Pyomo with a doubly weighted objective and solves it with the solver arguments.

select_tear_mip_model(*G*)

Generate a model for selecting tears from the given graph

Returns

- *model*
- *bin_list* – A list of the binary variables representing each edge, indexed by the edge index of the graph

set_guesses_for(*port*, *guesses*)

Set the guesses for the given port

These guesses will be checked for all free variables that are encountered during the first pass run. If a free variable has no guess, its current value will be used. If its current value is None, the default_guess option will be used. If that is None, an error will be raised.

All port variables that are downstream of a non-tear edge will already be fixed. If there is a guess for a fixed variable, it will be silently ignored.

The guesses should be a dict that maps the following:

Port Member Name -> Value

Or, for indexed members, multiple dicts that map:

Port Member Name -> Index -> Value

For extensive members, "Value" must be a list of tuples of the form (arc, value) to guess a value for the expanded variable of the specified arc. However, if the arc connecting this port is a 1-to-1 arc with its peer, then there will be no expanded variable for the single arc, so a regular "Value" should be provided.

This dict cannot be used to pass guesses for variables within expression type members. Guesses for those variables must be assigned to the variable's current value before calling run.

While this method makes things more convenient, all it does is:

self.options["guesses"][port] = guesses

set_tear_set(*tset*)

Set a custom tear set to be used when running the decomposition

The procedure will use this custom tear set instead of finding its own, thus it can save some time. Additionally, this will be useful for knowing which edges will need guesses.

Parameters

tset – A list of Arcs representing edges to tear

While this method makes things more convenient, all it does is:

self.options["tear_set"] = tset

tear_set_arcs(*G*, *method='mip'*, ***kwds*)

Call the specified tear selection method and return a list of arcs representing the selected tear edges.

The kwds will be passed to the method.

tree_order(*adj*, *adjR*, *roots=None*)

This function determines the ordering of nodes in a directed tree. This is a generic function that can operate on any given tree represented by the adjaceny and reverse adjacency lists. If the adjacency list does not represent a tree the results are not valid.

In the returned order, it is sometimes possible for more than one node to be calculated at once. So a list of lists is returned by this function. These represent a bredth first search order of the tree. Following the order, all nodes that lead to a particular node will be visited before it.

Parameters

- **adj** – An adjeceny list for a directed tree. This uses generic integer node indexes, not node names from the graph itself. This allows this to be used on sub-graphs and graps of components more easily.
- **adjR** – The reverse adjacency list coresponing to adj
- **roots** – List of node indexes to start from. These do not need to be the root nodes of the tree, in some cases like when a node changes the changes may only affect nodes reachable in the tree from the changed node, in the case that roots are supplied not all the nodes in the tree may appear in the ordering. If no roots are supplied, the roots of the tree are used.

3.2.6 Units Handling in Pyomo

Pyomo Units Container Module

This module provides support for including units within Pyomo expressions. This module can be used to define units on a model, and to check the consistency of units within the underlying constraints and expressions in the model. The module also supports conversion of units within expressions using the *convert* method to support construction of constraints that contain embedded unit conversions.

To use this package within your Pyomo model, you first need an instance of a PyomoUnitsContainer. You can use the module level instance already defined as 'units'. This object 'contains' the units - that is, you can access units on this module using common notation.

```
>>> from pyomo.environ import units as u
>>> print(3.0*u.kg)
3.0*kg
```

Units can be assigned to Var, Param, and ExternalFunction components, and can be used directly in expressions (e.g., defining constraints). You can also verify that the units are consistent on a model, or on individual components like the objective function, constraint, or expression using *assert_units_consistent* (from pyomo.util.check_units). There are other methods there that may be helpful for verifying correct units on a model.

```
>>> from pyomo.environ import ConcreteModel, Var, Objective
>>> from pyomo.environ import units as u
>>> from pyomo.util.check_units import assert_units_consistent, assert_units_
↪equivalent, check_units_equivalent
>>> model = ConcreteModel()
>>> model.acc = Var(initialize=5.0, units=u.m/u.s**2)
>>> model.obj = Objective(expr=(model.acc - 9.81*u.m/u.s**2)**2)
>>> assert_units_consistent(model.obj) # raise exc if units invalid on obj
>>> assert_units_consistent(model) # raise exc if units invalid anywhere on␣
↪the model
>>> assert_units_equivalent(model.obj.expr, u.m**2/u.s**4) # raise exc if␣
↪units not equivalent
>>> print(u.get_units(model.obj.expr)) # print the units on the objective
```

(continues on next page)

(continued from previous page)

```
m**2/s**4
>>> print(check_units_equivalent(model.acc, u.m/u.s**2))
True
```

The implementation is currently based on the pint package and supports all the units that are supported by pint. The list of units that are supported by pint can be found at the following url: https://github.com/hgrecco/pint/blob/master/pint/default_en.txt.

If you need a unit that is not in the standard set of defined units, you can create your own units by adding to the unit definitions within pint. See *PyomoUnitsContainer.load_definitions_from_file()* or *PyomoUnitsContainer.load_definitions_from_strings()* for more information.

Note

In this implementation of units, "offset" units for temperature are not supported within expressions (i.e. the non-absolute temperature units including degrees C and degrees F). This is because there are many non-obvious combinations that are not allowable. This concern becomes clear if you first convert the non-absolute temperature units to absolute and then perform the operation. For example, if you write 30 degC + 30 degC == 60 degC, but convert each entry to Kelvin, the expression is not true (i.e., 303.15 K + 303.15 K is not equal to 333.15 K). Therefore, there are several operations that are not allowable with non-absolute units, including addition, multiplication, and division.

This module does support conversion of offset units to absolute units numerically, using convert_value_K_to_C, convert_value_C_to_K, convert_value_R_to_F, convert_value_F_to_R. These are useful for converting input data to absolute units, and for converting data to convenient units for reporting.

Please see the pint documentation here for more discussion. While pint implements "delta" units (e.g., delta_degC) to support correct unit conversions, it can be difficult to identify and guarantee valid operations in a general algebraic modeling environment. While future work may support units with relative scale, the current implementation requires use of absolute temperature units (i.e. K and R) within expressions and a direct conversion of numeric values using specific functions for converting input data and reporting.

PyomoUnitsContainer([pint_registry])	Class that is used to create and contain units in Pyomo.
UnitsError(msg)	An exception class for all general errors/warnings associated with units
InconsistentUnitsError(exp1, exp2, msg)	An exception indicating that inconsistent units are present on an expression.

3.3 Solvers

3.3.1 Persistent Solvers

The purpose of the persistent solver interfaces is to efficiently notify the solver of incremental changes to a Pyomo model. The persistent solver interfaces create and store model instances from the Python API for the corresponding solver. For example, the *GurobiPersistent* class maintains a pointer to a gurobipy Model object. Thus, we can make small changes to the model and notify the solver rather than recreating the entire model using the solver Python API (or rewriting an entire model file - e.g., an lp file) every time the model is solved.

Warning

Users are responsible for notifying persistent solver interfaces when changes to a model are made!

Using Persistent Solvers

The first step in using a persistent solver is to create a Pyomo model as usual.

```
>>> import pyomo.environ as pyo
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
```

You can create an instance of a persistent solver through the SolverFactory.

```
>>> opt = pyo.SolverFactory('gurobi_persistent')
```

This returns an instance of `GurobiPersistent`. Now we need to tell the solver about our model.

```
>>> opt.set_instance(m)
```

This will create a gurobipy Model object and include the appropriate variables and constraints. We can now solve the model.

```
>>> results = opt.solve()
```

We can also add or remove variables, constraints, blocks, and objectives. For example,

```
>>> m.c2 = pyo.Constraint(expr=m.y >= m.x)
>>> opt.add_constraint(m.c2)
```

This tells the solver to add one new constraint but otherwise leave the model unchanged. We can now resolve the model.

```
>>> results = opt.solve()
```

To remove a component, simply call the corresponding remove method.

```
>>> opt.remove_constraint(m.c2)
>>> del m.c2
>>> results = opt.solve()
```

If a pyomo component is replaced with another component with the same name, the first component must be removed from the solver. Otherwise, the solver will have multiple components. For example, the following code will run without error, but the solver will have an extra constraint. The solver will have both y >= -2*x + 5 and y <= x, which is not what was intended!

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pyo.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> # WRONG:
>>> del m.c
```

(continues on next page)

(continued from previous page)

```
>>> m.c = pyo.Constraint(expr=m.y <= m.x)
>>> opt.add_constraint(m.c)
```

The correct way to do this is:

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pyo.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> # Correct:
>>> opt.remove_constraint(m.c)
>>> del m.c
>>> m.c = pyo.Constraint(expr=m.y <= m.x)
>>> opt.add_constraint(m.c)
```

Warning

Components removed from a pyomo model must be removed from the solver instance by the user.

Additionally, unexpected behavior may result if a component is modified before being removed.

```
>>> m = pyo.ConcreteModel()
>>> m.b = pyo.Block()
>>> m.b.x = pyo.Var()
>>> m.b.y = pyo.Var()
>>> m.b.c = pyo.Constraint(expr=m.b.y >= -2*m.b.x + 5)
>>> opt = pyo.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> m.b.c2 = pyo.Constraint(expr=m.b.y <= m.b.x)
>>> # ERROR: The constraint referenced by m.b.c2 does not
>>> # exist in the solver model.
>>> opt.remove_block(m.b)
```

In most cases, the only way to modify a component is to remove it from the solver instance, modify it with Pyomo, and then add it back to the solver instance. The only exception is with variables. Variables may be modified and then updated with with solver:

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pyo.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> m.x.setlb(1.0)
>>> opt.update_var(m.x)
```

Working with Indexed Variables and Constraints

The examples above all used simple variables and constraints; in order to use indexed variables and/or constraints, the code must be slightly adapted:

```
>>> for v in indexed_var.values():
...     opt.add_var(v)
>>> for v in indexed_con.values():
...     opt.add_constraint(v)
```

This must be done when removing variables/constraints, too. Not doing this would result in AttributeError exceptions, for example:

```
>>> opt.add_var(indexed_var)
>>> # ERROR: AttributeError: 'IndexedVar' object has no attribute 'is_binary'
>>> opt.add_constraint(indexed_con)
>>> # ERROR: AttributeError: 'IndexedConstraint' object has no attribute 'body'
```

The method "is_indexed" can be used to automate the process, for example:

```
>>> def add_variable(opt, variable):
...     if variable.is_indexed():
...         for v in variable.values():
...             opt.add_var(v)
...     else:
...         opt.add_var(v)
```

Persistent Solver Performance

In order to get the best performance out of the persistent solvers, use the "save_results" flag:

```
>>> import pyomo.environ as pyo
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pyo.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> results = opt.solve(save_results=False)
```

Note that if the "save_results" flag is set to False, then the following is not supported.

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     m.solutions.load_from(results)
```

However, the following will work:

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     opt.load_vars()
```

Additionally, a subset of variable values may be loaded back into the model:

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     opt.load_vars(m.x)
```

3.3.2 GDPopt logic-based solver

The GDPopt solver in Pyomo allows users to solve nonlinear Generalized Disjunctive Programming (GDP) models using logic-based decomposition approaches, as opposed to the conventional approach via reformulation to a Mixed Integer Nonlinear Programming (MINLP) model.

The main advantage of these techniques is their ability to solve subproblems in a reduced space, including nonlinear constraints only for `True` logical blocks. As a result, GDPopt is most effective for nonlinear GDP models.

Four algorithms are available in GDPopt:

1. Logic-based outer approximation (LOA) [Turkay & Grossmann, 1996]
2. Global logic-based outer approximation (GLOA) [Lee & Grossmann, 2001]
3. Logic-based branch-and-bound (LBB) [Lee & Grossmann, 2001]
4. Logic-based discrete steepest descent algorithm (LD-SDA) [Ovalle et al., 2025]

Usage and implementation details for GDPopt can be found in the PSE 2018 paper (Chen et al., 2018), or via its preprint.

Credit for prototyping and development can be found in the `GDPopt` class documentation, below.

GDPopt can be used to solve a Pyomo.GDP concrete model in two ways. The simplest is to instantiate the generic GDPopt solver and specify the desired algorithm as an argument to the `solve` method:

```
>>> pyo.SolverFactory('gdpopt').solve(model, algorithm='LOA')
```

The alternative is to instantiate an algorithm-specific GDPopt solver:

```
>>> pyo.SolverFactory('gdpopt.loa').solve(model)
```

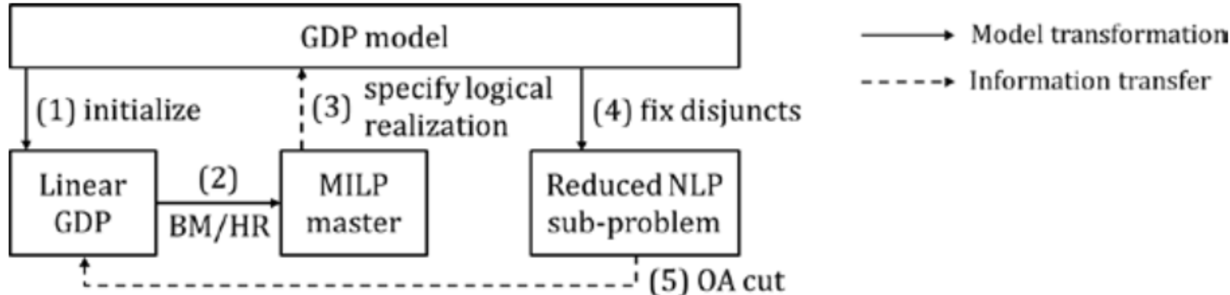
In the above examples, GDPopt uses the GDPopt-LOA algorithm. Other algorithms may be used by specifying them in the `algorithm` argument when using the generic solver or by instantiating the algorithm-specific GDPopt solvers. All GDPopt options are listed below.

Note

The generic GDPopt solver allows minimal configuration outside of the arguments to the `solve` method. To avoid repeatedly specifying the same configuration options to the `solve` method, use the algorithm-specific solvers.

Logic-based Outer Approximation (LOA)

Chen et al., 2018 contains the following flowchart, taken from the preprint version:



An example that includes the modeling approach may be found below.

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunct, Disjunction

Create a simple model
>>> model = pyo.ConcreteModel(name='LOA example')

>>> model.x = pyo.Var(bounds=(-1.2, 2))
>>> model.y = pyo.Var(bounds=(-10,10))
>>> model.c = pyo.Constraint(expr= model.x + model.y == 1)

>>> model.fix_x = Disjunct()
>>> model.fix_x.c = pyo.Constraint(expr=model.x == 0)

>>> model.fix_y = Disjunct()
>>> model.fix_y.c = pyo.Constraint(expr=model.y == 0)

>>> model.d = Disjunction(expr=[model.fix_x, model.fix_y])
>>> model.objective = pyo.Objective(expr=model.x + 0.1*model.y, sense=pyo.minimize)

Solve the model using GDPopt
>>> results = pyo.SolverFactory('gdpopt.loa').solve(
...     model, mip_solver='glpk')

Display the final solution
>>> model.display()
Model LOA example

  Variables:
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  -1.2 :     0 :     2 : False : False :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :   -10 :     1 :    10 : False : False :  Reals

  Objectives:
    objective : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True :   0.1
```

(continues on next page)

(continued from previous page)

```
Constraints:
  c : Size=1
      Key  : Lower : Body : Upper
      None :   1.0 :    1 :   1.0
```

Note

When troubleshooting, it can often be helpful to turn on verbose output using the `tee` flag.

```
>>> pyo.SolverFactory('gdpopt.loa').solve(model, tee=True)
```

Global Logic-based Outer Approximation (GLOA)

The same algorithm can be used to solve GDPs involving nonconvex nonlinear constraints by solving the subproblems globally:

```
>>> pyo.SolverFactory('gdpopt.gloa').solve(model)
```

Warning

The `nlp_solver` option must be set to a global solver for the solution returned by GDPopt to also be globally optimal.

Relaxation with Integer Cuts (RIC)

Instead of outer approximation, GDPs can be solved using the same MILP relaxation as in the previous two algorithms, but instead of using the subproblems to generate outer-approximation cuts, the algorithm adds only no-good cuts for every discrete solution encountered:

```
>>> pyo.SolverFactory('gdpopt.ric').solve(model)
```

Again, this is a global algorithm if the subproblems are solved globally, and is not otherwise.

Note

The RIC algorithm will not necessarily enumerate all discrete solutions as it is possible for the bounds to converge first. However, full enumeration is not uncommon.

Logic-based Branch-and-Bound (LBB)

The GDPopt-LBB solver branches through relaxed subproblems with inactive disjunctions. It explores the possibilities based on best lower bound, eventually activating all disjunctions and presenting the globally optimal solution.

To use the GDPopt-LBB solver, define your Pyomo GDP model as usual:

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunct, Disjunction
```

(continues on next page)

(continued from previous page)

```
Create a simple model
>>> m = pyo.ConcreteModel()
>>> m.x1 = pyo.Var(bounds = (0,8))
>>> m.x2 = pyo.Var(bounds = (0,8))
>>> m.obj = pyo.Objective(expr=m.x1 + m.x2, sense=pyo.minimize)
>>> m.y1 = Disjunct()
>>> m.y2 = Disjunct()
>>> m.y1.c1 = pyo.Constraint(expr=m.x1 >= 2)
>>> m.y1.c2 = pyo.Constraint(expr=m.x2 >= 2)
>>> m.y2.c1 = pyo.Constraint(expr=m.x1 >= 3)
>>> m.y2.c2 = pyo.Constraint(expr=m.x2 >= 3)
>>> m.djn = Disjunction(expr=[m.y1, m.y2])

Invoke the GDPopt-LBB solver

>>> results = pyo.SolverFactory('gdpopt.lbb').solve(m)
WARNING: 09/06/22: The GDPopt LBB algorithm currently has known issues. Please
    use the results with caution and report any bugs!

>>> print(results)
>>> print(results.solver.status)
ok
>>> print(results.solver.termination_condition)
optimal

>>> print([pyo.value(m.y1.indicator_var), pyo.value(m.y2.indicator_var)])
[True, False]
```

Logic-based Discrete-Steepest Descent Algorithm (LD-SDA)

The GDPopt-LDSDA solver exploits the ordered Boolean variables in the disjunctions to solve the GDP model. It requires an **exclusive OR (XOR) logical constraint** to ensure that exactly one disjunct is active in each disjunction. The solver also requires a **starting point** for the discrete variables and allows users to choose between two **direction norms**, *'L2'* and *'Linf'*, to guide the search process.

Note

The current implementation of the GDPopt-LDSDA requires an explicit LogicalConstraint to enforce the exclusive OR condition for each disjunction.

To use the GDPopt-LDSDA solver, define your Pyomo GDP model as usual:

```
Required imports
>>> import pyomo.environ as pyo
>>> from pyomo.gdp import Disjunct, Disjunction

Create a simple model
>>> m = pyo.ConcreteModel()

Define sets
```

(continues on next page)

(continued from previous page)

```
>>> I = [1, 2, 3, 4, 5]
>>> J = [1, 2, 3, 4, 5]

Define variables
>>> m.a = pyo.Var(bounds=(-0.3, 0.2))
>>> m.b = pyo.Var(bounds=(-0.9, -0.5))

Define disjuncts for Y1
>>> m.Y1_disjuncts = Disjunct(I)
>>> for i in I:
...     m.Y1_disjuncts[i].y1_constraint = pyo.Constraint(expr=m.a == -0.3 + 0.1 * (i -
↪1))

Define disjuncts for Y2
>>> m.Y2_disjuncts = Disjunct(J)
>>> for j in J:
...     m.Y2_disjuncts[j].y2_constraint = pyo.Constraint(expr=m.b == -0.9 + 0.1 * (j -
↪1))

Define disjunctions
>>> m.y1_disjunction = Disjunction(expr=[m.Y1_disjuncts[i] for i in I])
>>> m.y2_disjunction = Disjunction(expr=[m.Y2_disjuncts[j] for j in J])

Logical constraints to enforce exactly one selection
>>> m.Y1_limit = pyo.LogicalConstraint(expr=pyo.exactly(1, [m.Y1_disjuncts[i].indicator_
↪var for i in I]))
>>> m.Y2_limit = pyo.LogicalConstraint(expr=pyo.exactly(1, [m.Y2_disjuncts[j].indicator_
↪var for j in J]))

Define objective function
>>> m.obj = pyo.Objective(
...     expr=4 * m.a**2 - 2.1 * m.a**4 + (1 / 3) * m.a**6 + m.a * m.b - 4 * m.b**2 + 4 *
↪m.b**4,
...     sense=pyo.minimize
... )

Invoke the GDPopt-LDSDA solver
>>> results = pyo.SolverFactory('gdpopt.ldsda').solve(m,
...     starting_point=[1,1],
...     logical_constraint_list=[m.Y1_limit, m.Y2_limit],
...     direction_norm='Linf',
... )
```

GDPopt implementation and optional arguments

> **Warning**
>
> GDPopt optional arguments should be considered beta code and are subject to change.

GDPoptSolver()	Decomposition solver for Generalized Disjunctive Programming (GDP) problems.
GDP_LOA_Solver(**kwds)	The GDPopt (Generalized Disjunctive Programming optimizer) logic-based outer approximation (LOA) solver.
GDP_GLOA_Solver(**kwds)	The GDPopt (Generalized Disjunctive Programming optimizer) global logic-based outer approximation (GLOA) solver.
GDP_RIC_Solver(**kwds)	The GDPopt (Generalized Disjunctive Programming optimizer) relaxation with integer cuts (RIC) solver.
GDP_LBB_Solver(**kwds)	The GDPopt (Generalized Disjunctive Programming optimizer) logic-based branch and bound (LBB) solver.
GDP_LDSDA_Solver(**kwds)	The GDPopt (Generalized Disjunctive Programming optimizer) LD-SDA (Logic-based Discrete-Steepest Descent Algorithm) solver.

3.3.3 PyROS Solver

PyROS (Pyomo Robust Optimization Solver) is a Pyomo-based meta-solver for non-convex, two-stage adjustable robust optimization problems.

It was developed by **Natalie M. Isenberg**, **Jason A. F. Sherman**, and **Chrysanthos E. Gounaris** of Carnegie Mellon University, in collaboration with **John D. Siirola** of Sandia National Laboratories. The developers gratefully acknowledge support from the U.S. Department of Energy's Institute for the Design of Advanced Energy Systems (IDAES) and Carbon Capture Simulation for Industry Impact (CCSI2) projects.

PyROS Methodology Overview

PyROS can accommodate optimization models with:

- **Continuous variables** only
- **Nonlinearities** (including **nonconvexities**) in both the variables and uncertain parameters
- **First-stage degrees of freedom** and **second-stage degrees of freedom**
- **Equality constraints** defining state variables, including implicitly defined state variables that cannot be eliminated from the model via reformulation
- **Uncertain parameters** participating in the inequality constraints, equality constraints, and/or objective function

Supported deterministic models are nonlinear programs (NLPs) of the general form

$$\begin{array}{lll} \displaystyle\min_{\substack{x\in\mathcal{X},\\ z\in\mathbb{R}^{n_z},\\ y\in\mathbb{R}^{n_y}}} & f_1(x) + f_2(x,z,y;q^{\text{nom}}) & \\ \text{s.t.} & g_i(x,z,y;q^{\text{nom}}) \leq 0 & \forall\, i\in\mathcal{I} \\ & h_j(x,z,y;q^{\text{nom}}) = 0 & \forall\, j\in\mathcal{J} \end{array}$$

where:

- $x \in \mathcal{X}$ denotes the first-stage degree of freedom variables (or design variables), of which the feasible space $\mathcal{X} \subseteq \mathbb{R}^{n_x}$ is defined by the model constraints (including variable bounds specifications) referencing x only
- $z \in \mathbb{R}^{n_z}$ denotes the second-stage degree of freedom variables (or control variables)
- $y \in \mathbb{R}^{n_y}$ denotes the state variables
- $q \in \mathbb{R}^{n_q}$ denotes the model parameters considered uncertain, and q^{nom} is the vector of nominal values associated with those

- $f_1(x)$ is the summand of the objective function that depends only on the first-stage degree of freedom variables
- $f_2(x, z, y; q)$ is the summand of the objective function that depends on all variables and the uncertain parameters
- $g_i(x, z, y; q)$ is the ith inequality constraint function in set $\mathcal{I}$ (see *first Note*)
- $h_j(x, z, y; q)$ is the jth equality constraint function in set $\mathcal{J}$ (see *second Note*)

Note

PyROS accepts and automatically reformulates models with:

1. Interval bounds on components of (x, z, y)
2. Ranged inequality constraints

Note

A key assumption of PyROS is that for every $x \in \mathcal{X}$, $z \in \mathbb{R}^{n_z}$, $q \in \mathcal{Q}$, there exists a unique $y \in \mathbb{R}^{n_y}$ for which (x, z, y, q) satisfies the equality constraints $h_j(x, z, y, q) = 0 \, \forall j \in \mathcal{J}$. If this assumption is not met, then the selection of state (i.e., not degree of freedom) variables y is incorrect, and one or more entries of y should be appropriately redesignated to be part of either x or z.

In order to cast the robust optimization counterpart of the *deterministic model*, we now assume that the uncertain parameters q may attain any realization in a compact uncertainty set $\mathcal{Q} \subseteq \mathbb{R}^{n_q}$ containing the nominal value q^{nom}. The set $\mathcal{Q}$ may be **either continuous or discrete**.

Based on the above notation, the form of the robust counterpart addressed by PyROS is

$$\begin{array}{llll} \min\limits_{x \in \mathcal{X}} & \max\limits_{q \in \mathcal{Q}} & \min\limits_{\substack{z \in \mathbb{R}^{n_z}, \\ y \in \mathbb{R}^{n_y}}} & f_1(x) + f_2(x, z, y, q) & \\ & & \text{s.t.} & g_i(x, z, y, q) \leq 0 & \forall\, i \in \mathcal{I} \\ & & & h_j(x, z, y, q) = 0 & \forall\, j \in \mathcal{J} \end{array}$$

PyROS accepts a deterministic model and accompanying uncertainty set and then, using the Generalized Robust Cutting-Set algorithm developed in [IAE+21], seeks a solution to the robust counterpart.

Getting Started with PyROS

Table of Contents

- *Installation*
- *Quickstart*
  - *Step 0: Import Pyomo and the PyROS Module*
  - *Step 1: Define the Solver Inputs*
    - *Deterministic Model*
    - *First-Stage and Second-Stage Variables*
    - *Uncertain Parameters*
    - *Uncertainty Set*

      * *Subordinate NLP Solvers*
    - *Step 2: Solve With PyROS*
      * *Invoke PyROS*
      * *Inspect the Results*
      * *Try Higher-Order Decision Rules*
    - *Analyzing the Price of Robustness*
- *Beyond the Basics*

Installation

In advance of using PyROS to solve robust optimization problems, you will need (at least) one local nonlinear programming (NLP) solver (e.g., CONOPT, IPOPT, Knitro) and (at least) one global NLP solver (e.g., BARON, COUENNE, SCIP) installed and licensed on your system.

PyROS can be installed as follows:

1. *Install Pyomo*. PyROS is included in the Pyomo software package, at pyomo/contrib/pyros.
2. Install NumPy and SciPy with your preferred package manager; both NumPy and SciPy are required dependencies of PyROS. You may install NumPy and SciPy with, for example, `conda`:

   ```
   conda install numpy scipy
   ```

   or `pip`:

   ```
   pip install numpy scipy
   ```

3. (*Optional*) Test your installation: install `pytest` and `parameterized` with your preferred package manager (as in the previous step):

   ```
   pip install pytest parameterized
   ```

   You may then run the PyROS tests as follows:

   ```
   python -c 'import os, pytest, pyomo.contrib.pyros as p; pytest.main([os.path.
   ↪dirname(p.__file__)])'
   ```

   Some tests involving deterministic NLP solvers may be skipped if IPOPT, BARON, or SCIP is not pre-installed and licensed on your system.

Quickstart

We now provide a quick overview of how to use PyROS to solve a robust optimization problem.

Consider the nonconvex deterministic QCQP

$$\begin{array}{cl}
\min\limits_{\substack{x\in[-100,100],\\ z\in[-100,100],\\ (y_1,y_2)\in[-100,100]^2}} & x^2 - y_1 z + y_2 \\
\text{s.t.} & xy_1 \geq 150(q_1+1)^2 \\
& x + y_2^2 \leq 600 \\
& xz - q_2 y_1 = 2 \\
& y_1^2 - 2y_2 = 15
\end{array}$$

in which x is the sole first-stage variable, z is the sole second-stage variable, y_1, y_2 are the state variables, and q_1, q_2 are the uncertain parameters.

The uncertain parameters q_1, q_2 each have a nominal value of 1. We assume that q_1, q_2 can independently deviate from their nominal values by up to $\pm 10\%$, so that (q_1, q_2) is constrained in value to the interval uncertainty set $\mathcal{Q} = [0.9, 1.1]^2$.

Note

Per our analysis, our selections of first-stage variables and second-stage variables in the present example satisfy our *assumption that the state variable values are uniquely defined*.

Step 0: Import Pyomo and the PyROS Module

In anticipation of using the PyROS solver and building the deterministic Pyomo model:

```
>>> import pyomo.environ as pyo
>>> import pyomo.contrib.pyros as pyros
```

Step 1: Define the Solver Inputs

Deterministic Model

The model can be implemented as follows:

```
>>> m = pyo.ConcreteModel()
>>> # parameters
>>> m.q1 = pyo.Param(initialize=1, mutable=True)
>>> m.q2 = pyo.Param(initialize=1, mutable=True)
>>> # variables
>>> m.x = pyo.Var(bounds=[-100, 100])
>>> m.z = pyo.Var(bounds=[-100, 100])
>>> m.y1 = pyo.Var(bounds=[-100, 100])
>>> m.y2 = pyo.Var(bounds=[-100, 100])
>>> # objective
>>> m.obj = pyo.Objective(expr=m.x ** 2 - m.y1 * m.z + m.y2)
>>> # constraints
>>> m.ineq1 = pyo.Constraint(expr=m.x * m.y1 >= 150 * (m.q1 + 1) ** 2)
>>> m.ineq2 = pyo.Constraint(expr=m.x + m.y2 ** 2 <= 600)
>>> m.eq1 = pyo.Constraint(expr=m.x * m.z - m.y1 * m.q2 == 2)
>>> m.eq2 = pyo.Constraint(expr=m.y1 ** 2 - 2 * m.y2 == 15)
```

Observe that the uncertain parameters q_1, q_2 are implemented as mutable `Param` objects. See the *Uncertain parameters section of the Solver Interface documentation* for further guidance.

First-Stage and Second-Stage Variables

We take `m.x` to be the sole first-stage variable and `m.z` to be the sole second-stage variable:

```
>>> first_stage_variables = [m.x]
>>> second_stage_variables = [m.z]
```

Uncertain Parameters

The uncertain parameters are represented by `m.q1` and `m.q2`:

```
>>> uncertain_params = [m.q1, m.q2]
```

Uncertainty Set

As previously discussed, we take the uncertainty set to be the interval $[0.9, 1.1]^2$, which we can implement as a *BoxSet* object:

```
>>> box_uncertainty_set = pyros.BoxSet(bounds=[(0.9, 1.1)] * 2)
```

Further information on PyROS uncertainty sets is presented in the *Uncertainty Sets documentation*.

Subordinate NLP Solvers

We will use IPOPT as the subordinate local NLP solver and BARON as the subordinate global NLP solver:

```
>>> local_solver = pyo.SolverFactory("ipopt")
>>> global_solver = pyo.SolverFactory("baron")
```

Note

Additional NLP optimizers can be automatically used in the event the primary subordinate local or global optimizer passed to the PyROS *solve()* method does not successfully solve a subproblem to an appropriate termination condition. These alternative solvers can be provided through the optional keyword arguments `backup_local_solvers` and `backup_global_solvers` to the PyROS *solve()* method.

In advance of using PyROS, we check that the model can be solved to optimality with the subordinate global solver:

```
>>> pyo.assert_optimal_termination(global_solver.solve(m))
>>> deterministic_obj = pyo.value(m.obj)
>>> print(f"Optimal deterministic objective value: {deterministic_obj:.2f}")
Optimal deterministic objective value: 5407.94
```

Step 2: Solve With PyROS

PyROS can be instantiated through the Pyomo `SolverFactory`:

```
>>> pyros_solver = pyo.SolverFactory("pyros")
```

Invoke PyROS

We now use PyROS to solve the model to robust optimality by invoking the *solve()* method of the PyROS solver object:

```
>>> results_1 = pyros_solver.solve(
...     # required arguments
...     model=m,
...     first_stage_variables=first_stage_variables,
...     second_stage_variables=second_stage_variables,
```

(continues on next page)

(continued from previous page)

```
...     uncertain_params=uncertain_params,
...     uncertainty_set=box_uncertainty_set,
...     local_solver=local_solver,
...     global_solver=global_solver,
...     # optional arguments: passed directly to
...     #  solve to robust optimality
...     objective_focus="worst_case",
...     solve_master_globally=True,
... )
==============================================================================
PyROS: The Pyomo Robust Optimization Solver...
...
Robust optimal solution identified.
...
All done. Exiting PyROS.
==============================================================================
```

PyROS, by default, logs to the output console the progress of the optimization and, upon termination, a summary of the final result. The summary includes the iteration and solve time requirements, the final objective function value, and the termination condition. For further information on the output log, see the *Solver Output Log documentation*.

Note

PyROS, like other Pyomo solvers, accepts optional arguments passed indirectly through the keyword argument `options`. This is discussed further in the *Optional Arguments section of the Solver Interface documentation*. Thus, the PyROS solver invocation in the *preceding code snippet* is equivalent to:

```
results_1 = pyros_solver.solve(
    model=m,
    first_stage_variables=first_stage_variables,
    second_stage_variables=second_stage_variables,
    uncertain_params=uncertain_params,
    uncertainty_set=box_uncertainty_set,
    local_solver=local_solver,
    global_solver=global_solver,
    # optional arguments: passed indirectly to
    #  solve to robust optimality
    options={
        "objective_focus": "worst_case",
        "solve_master_globally": True,
    },
)
```

Inspect the Results

The PyROS *solve()* method returns a results object, of type *ROSolveResults*, that summarizes the outcome of invoking PyROS on a robust optimization problem. By default, a printout of the results object is included at the end of the solver output log. Alternatively, we can display the results object ourselves using:

```
>>> print(results_1)  # output may vary
Termination stats:
```

(continues on next page)

(continued from previous page)

```
Iterations             : 3
Solve time (wall s)    : 0.917
Final objective value : 9.6616e+03
Termination condition : pyrosTerminationCondition.robust_optimal
```

We can also query the results object's individual attributes:

```
>>> results_1.iterations  # total number of iterations
3
>>> results_1.time  # total wallclock time; may vary
0.839
>>> results_1.final_objective_value  # final objective value; may vary
9661.6...
>>> results_1.pyros_termination_condition  # termination condition
<pyrosTerminationCondition.robust_optimal: 1>
```

Since PyROS has successfully solved our problem, the final solution has been automatically loaded to the model. We can inspect the resulting state of the model by invoking, for example, `m.display()` or `m.pprint()`.

For a general discussion of the PyROS solver outputs, see the *Overview of Outputs section of the Solver Interface documentation*.

Try Higher-Order Decision Rules

PyROS uses polynomial decision rules to approximate the adjustability of the second-stage variables to the uncertain parameters. The degree of the decision rule polynomials is specified through the optional keyword argument `decision_rule_order` to the PyROS *solve()* method. By default, `decision_rule_order` is set to 0, so that static decision rules are used. Increasing the decision rule order may yield a solution with better quality:

```
>>> results_2 = pyros_solver.solve(
...     model=m,
...     first_stage_variables=first_stage_variables,
...     second_stage_variables=second_stage_variables,
...     uncertain_params=uncertain_params,
...     uncertainty_set=box_uncertainty_set,
...     local_solver=local_solver,
...     global_solver=global_solver,
...     objective_focus="worst_case",
...     solve_master_globally=True,
...     decision_rule_order=1,  # use affine decision rules
... )
==============================================================================
PyROS: The Pyomo Robust Optimization Solver...
...
Robust optimal solution identified.
...
All done. Exiting PyROS.
==============================================================================
```

Inspecting the results:

```
>>> print(results_2)  # output may vary
Termination stats:
```

(continues on next page)

(continued from previous page)

```
 Iterations              : 5
 Solve time (wall s)     : 1.956
 Final objective value : 6.5403e+03
 Termination condition : pyrosTerminationCondition.robust_optimal
```

Notice that when we switch from optimizing over static decision rules to optimizing over affine decision rules, there is a ~32% decrease in the final objective value, albeit at some additional computational expense.

Analyzing the Price of Robustness

In conjunction with standard Pyomo control flow tools, PyROS facilitates an analysis of the “price of robustness”, which we define to be the increase in the robust optimal objective value relative to the deterministically optimal objective value.

Let us, for example, consider optimizing robustly against an interval uncertainty set $[1 - p, 1 + p]^2$, where p is the half-length of the interval. We can optimize against intervals of increasing half-length p by iterating over select values for p in a `for` loop, and in each iteration, solving a robust optimization problem subject to a corresponding *BoxSet* instance:

```
>>> results_dict = dict()
>>> for half_length in [0.0, 0.1, 0.2, 0.3, 0.4]:
...     print(f"Solving problem for {half_length=}:")
...     results_dict[half_length] = pyros_solver.solve(
...         model=m,
...         first_stage_variables=first_stage_variables,
...         second_stage_variables=second_stage_variables,
...         uncertain_params=uncertain_params,
...         uncertainty_set=pyros.BoxSet(
...             bounds=[(1 - half_length, 1 + half_length)] * 2
...         ),
...         local_solver=local_solver,
...         global_solver=global_solver,
...         objective_focus="worst_case",
...         solve_master_globally=True,
...         decision_rule_order=1,
...     )
...
Solving problem for half_length=0.0:
...
Solving problem for half_length=0.1:
...
Solving problem for half_length=0.2:
...
Solving problem for half_length=0.3:
...
Solving problem for half_length=0.4:
...
All done. Exiting PyROS.
==============================================================================
```

Using the `dict` populated in the loop, and the *previously evaluated deterministically optimal objective value*, we can print a tabular summary of the results:

```
>>> for idx, (half_length, res) in enumerate(results_dict.items()):
...     if idx == 0:
...         # print table header
...         print("=" * 71)
...         print(
...             f"{'Half-Length':15s}"
...             f"{'Termination Cond.':21s}"
...             f"{'Objective Value':18s}"
...             f"{'Price of Rob. (%)':17s}"
...         )
...         print("-" * 71)
...     # print table row
...     obj_value, percent_obj_increase = float("nan"), float("nan")
...     is_robust_optimal = (
...         res.pyros_termination_condition
...         == pyros.pyrosTerminationCondition.robust_optimal
...     )
...     is_robust_infeasible = (
...         res.pyros_termination_condition
...         == pyros.pyrosTerminationCondition.robust_infeasible
...     )
...     if is_robust_optimal:
...         # compute the price of robustness
...         obj_value = res.final_objective_value
...         price_of_robustness = (
...             (res.final_objective_value - deterministic_obj)
...             / deterministic_obj
...         )
...     elif is_robust_infeasible:
...         # infinite objective
...         obj_value, price_of_robustness = float("inf"), float("inf")
...     print(
...         f"{half_length:<15.1f}"
...         f"{res.pyros_termination_condition.name:21s}"
...         f"{obj_value:<18.2f}"
...         f"{100 * price_of_robustness:<.2f}"
...     )
...     print("-" * 71)
...
=======================================================================
Half-Length    Termination Cond.    Objective Value   Price of Rob. (%)
-----------------------------------------------------------------------
0.0            robust_optimal       5407.94           -0.00
-----------------------------------------------------------------------
0.1            robust_optimal       6540.31           20.94
-----------------------------------------------------------------------
0.2            robust_optimal       7838.50           44.94
-----------------------------------------------------------------------
0.3            robust_optimal       9316.88           72.28
-----------------------------------------------------------------------
0.4            robust_infeasible    inf               inf
-----------------------------------------------------------------------
```

The table shows the response of the PyROS termination condition, final objective value, and price of robustness to the half-length p. Observe that:

- The optimal objective value for the interval of half-length $p = 0$ is equal to the optimal deterministic objective value
- The objective value (and thus, the price of robustness) increases with the half-length
- For large enough half-length ($p = 0.4$), the problem is robust infeasible

Therefore, this example clearly illustrates the potential impact of the uncertainty set size on the robust optimal objective function value and the ease of analyzing the price of robustness for a given optimization problem under uncertainty.

Beyond the Basics

A more in-depth guide to incorporating PyROS into a Pyomo optimization workflow is given in the *Usage Tutorial*.

PyROS Solver Interface

Table of Contents

- *Instantiation*
- *Overview of Inputs*
  - *Deterministic Model*
  - *First-Stage and Second-Stage Variables*
  - *Uncertain Parameters*
  - *Uncertainty Set*
  - *Subordinate NLP Solvers*
  - *Optional Arguments*
  - *Separation Priority Ordering*
- *Overview of Outputs*
  - *Results Object*
  - *Final Solution*
  - *Solver Output Log*

Instantiation

The PyROS solver is invoked through the *solve()* method of an instance of the PyROS solver class, which can be instantiated as follows:

```
import pyomo.environ as pyo
import pyomo.contrib.pyros as pyros  # register the PyROS solver
pyros_solver = pyo.SolverFactory("pyros")
```

Overview of Inputs

Deterministic Model

PyROS is designed to operate on a single-objective deterministic model (implemented as a *ConcreteModel*), from which the robust optimization counterpart is automatically inferred. All variables of the model should be continuous, as mixed-integer problems are not supported.

First-Stage and Second-Stage Variables

A model may have either first-stage variables, second-stage variables, or both. PyROS automatically considers all other variables participating in the active model components to be state variables. Further, PyROS assumes that the state variables are *uniquely defined by the equality constraints*.

Uncertain Parameters

Uncertain parameters can be represented by either mutable *Param* or fixed *Var* objects. Uncertain parameters *cannot* be directly represented by Python literals that have been hard-coded into the deterministic model.

A *Param* object can be made mutable at construction by passing the argument `mutable=True` to the *Param* constructor. If specifying/modifying the `mutable` argument is not straightforward in your context, then add the following lines of code to your script before setting up your deterministic model:

```
import pyomo.environ as pyo
pyo.Param.DefaultMutable = True
```

All *Param* objects declared after the preceding code statements will be made mutable by default.

Uncertainty Set

The uncertainty set is represented by an *UncertaintySet* object. See the *Uncertainty Sets documentation* for more information.

Subordinate NLP Solvers

PyROS requires at least one subordinate local nonlinear programming (NLP) solver (e.g., Ipopt or CONOPT) and subordinate global NLP solver (e.g., BARON or SCIP) to solve subproblems.

Note

In advance of invoking the PyROS solver, check that your deterministic model can be solved to optimality by either your subordinate local or global NLP solver.

Optional Arguments

The optional arguments are enumerated in the documentation of the *solve()* method.

Like other Pyomo solver interface methods, the PyROS *solve()* method accepts the keyword argument `options`, which must be a `dict` mapping names of optional arguments to *solve()* to their desired values. If an argument is passed directly by keyword and indirectly through `options`, then the value passed directly takes precedence over the value passed through `options`.

Warning

All required arguments to the PyROS *solve()* method must be passed directly by position or keyword, or else an exception is raised. Required arguments passed indirectly through the `options` setting are ignored.

Separation Priority Ordering

The PyROS solver supports custom prioritization of the separation subproblems (and, thus, the constraints) that are automatically derived from a given model for robust optimization. Users may specify separation priorities through:

- (Recommended) *Suffix* components with local name `pyros_separation_priority`, declared on the model or any of its sub-blocks. Each entry of every such *Suffix* should map a *Var* or *Constraint* component to a value that specifies the separation priority of all constraints derived from that component
- The optional argument `separation_priority_order` to the PyROS *solve()* method. The argument should be castable to a `dict`, of which each entry maps the full name of a *Var* or *Constraint* component to a value that specifies the separation priority of all constraints derived from that component

Specification via *Suffix* components takes precedence over specification via the solver argument `separation_priority_order`. Moreover, the precedence ordering among *Suffix* components is handled by the Pyomo *SuffixFinder* utility.

A separation priority can be either a (real) number (i.e., of type `int`, `float`, etc.) or `None`. A higher number indicates a higher priority. The default priority for all constraints is 0. Therefore a constraint can be prioritized [or deprioritized] over the default by mapping the constraint to a positive [or negative] number. In practice, critical or dominant constraints are often prioritized over algorithmic or implied constraints.

Constraints that have been assigned a priority of `None` are enforced subject to only the nominal uncertain parameter realization provided by the user. Therefore, these constraints are not imposed robustly and, in particular, are excluded from the separation problems.

Overview of Outputs

Results Object

The *solve()* method returns an *ROSolveResults* object.

When the PyROS *solve()* method has successfully solved a given robust optimization problem, the *pyros_termination_condition* attribute of the returned *ROSolveResults* object is set to *robust_optimal* if and only if:

1. Master problems are solved to global optimality (by passing `solve_master_globally=True`)
2. A worst-case objective focus is chosen (by setting `objective_focus` to *worst_case*)

Otherwise, the termination condition is set to *robust_feasible*.

The *final_objective_value* attribute of the results object depends on the value of the optional `objective_focus` argument to the *solve()* method:

- If `objective_focus` is set to *nominal*, then the objective is evaluated subject to the nominal uncertain parameter realization
- If `objective_focus` is set to *worst_case*, then the objective is evaluated subject to the uncertain parameter realization that induces the worst-case objective value

The second-stage variable and state variable values in the *solution loaded to the model* are evaluated similarly.

Final Solution

PyROS automatically loads the final solution found to the model (i.e., updates the values of the variables of the deterministic model) if and only if:

1. The argument `load_solution=True` has been passed to PyROS (occurs by default)
2. The *pyros_termination_condition* attribute of the returned *ROSolveResults* object is either *robust_optimal* or *robust_feasible*

Otherwise, the solution is lost.

If a solution is loaded to the model, then, as mentioned in our discussion of the *results object*, the second-stage variables and state variables of the model are updated according to the value of the optional `objective_focus` argument to the *solve()* method. The uncertain parameter objects are left unchanged.

Solver Output Log

When the PyROS *solve()* method is invoked to solve an RO problem, the progress and final result are reported through a highly configurable logging system. See the *Solver Output Log documentation* for more information.

PyROS Uncertainty Sets

Table of Contents

- *Overview*
- *Pre-Implemented Uncertainty Set Types*
- *Custom Uncertainty Set Types*

Overview

In PyROS, the uncertainty set of a robust optimization problem is represented by an instance of a subclass of the *UncertaintySet* abstract base class. PyROS provides a suite of *pre-implemented concrete subclasses* to facilitate instantiation of uncertainty sets that are commonly used in the optimization literature. *Custom uncertainty set types* can be implemented by subclassing *UncertaintySet*.

Note

The *UncertaintySet* class is an abstract class and therefore cannot be directly instantiated.

Pre-Implemented Uncertainty Set Types

The pre-implemented *UncertaintySet* subclasses are enumerated below:

AxisAlignedEllipsoidalSet(center, half_lengths)	An axis-aligned ellipsoidal region.
BoxSet(bounds)	A hyperrectangle (or box).
BudgetSet(budget_membership_mat, rhs_vec[, ...])	A budget set.
CardinalitySet(origin, positive_deviation, gamma)	A cardinality-constrained (i.e., "gamma") set.
CartesianProductSet(all_sets)	A Cartesian product of uncertainty sets.
DiscreteScenarioSet(scenarios)	A set of finitely many distinct points (or scenarios).

continues on next page

Table 3.5 – continued from previous page

EllipsoidalSet(center, shape_matrix[, ...])	A general ellipsoidal region.
FactorModelSet(origin, number_of_factors, ...)	A factor model (i.e., "net-alpha" model) set.
IntersectionSet(**unc_sets)	An intersection of two or more uncertainty sets, each of which is represented by an *UncertaintySet* object.
PolyhedralSet(lhs_coefficients_mat, rhs_vec)	A bounded convex polyhedron or polytope.

Mathematical definitions of the pre-implemented *UncertaintySet* subclasses are provided below.

Table 3.6: Mathematical definitions of PyROS uncertainty sets of dimension n.

Uncertainty Set Type	Input Data	Mathematical Definition
AxisAlignedEllipsoidalSet	$q^0 \in \mathbb{R}^n$, $\alpha \in \mathbb{R}^n_+$	$\left\{ q \in \mathbb{R}^n \,\middle|\, \begin{array}{l} \sum\limits_{\substack{i=1: \\ \alpha_i > 0}}^{n} \left(\frac{q_i - q_i^0}{\alpha_i}\right)^2 \leq 1 \\ q_i = q_i^0 \, \forall \, i : \alpha_i = 0 \end{array} \right\}$
BoxSet	$q^{\mathrm{L}} \in \mathbb{R}^n$, $q^{\mathrm{U}} \in \mathbb{R}^n$	$\{q \in \mathbb{R}^n \mid q^{\mathrm{L}} \leq q \leq q^{\mathrm{U}}\}$
BudgetSet	$q^0 \in \mathbb{R}^n$, $b \in \mathbb{R}^L_+$, $B \in \{0,1\}^{L \times n}$	$\left\{ q \in \mathbb{R}^n \,\middle|\, \begin{pmatrix} B \\ -I \end{pmatrix} q \leq \begin{pmatrix} b + Bq^0 \\ -q^0 \end{pmatrix} \right\}$
CardinalitySet	$q^0 \in \mathbb{R}^n$, $\hat{q}^+ \in \mathbb{R}^n_+$, $\hat{q}^- \in \mathbb{R}^n_+$, $\Gamma \in [0, n]$	$\left\{ q \in \mathbb{R}^n \,\middle|\, \begin{array}{l} \exists \, \xi^+, \xi^- \in [0,1]^n : \\ \quad q = q^0 + \hat{q}^+ \circ \xi^+ - \hat{q}^- \circ \xi^- \\ \quad \sum\limits_{i=1}^{n} (\xi_i^+ + \xi_i^-) \leq \Gamma \\ \quad \xi_i^+ = 0 \quad \forall \, i : \hat{q}_i^+ = 0 \\ \quad \xi_i^- = 0 \quad \forall \, i : \hat{q}_i^- = 0 \end{array} \right\}$
CartesianProductSet	$\mathcal{Q}_1 \subset \mathbb{R}^{n_1}$, $\mathcal{Q}_2 \subset \mathbb{R}^{n_2}$, $\vdots$ $\mathcal{Q}_m \subset \mathbb{R}^{n_m}$	$\mathcal{Q}_1 \times \mathcal{Q}_2 \times \cdots \times \mathcal{Q}_m$
DiscreteScenarioSet	$q^1, q^2, \ldots, q^S \in \mathbb{R}^n$	$\{q^1, q^2, \ldots, q^S\}$
EllipsoidalSet	$q^0 \in \mathbb{R}^n$, $P \in \mathbb{S}^n_{++}$, $s \in \mathbb{R}_+$	$\{q \in \mathbb{R}^n \mid (q - q^0)^\intercal P^{-1} (q - q^0) \leq s\}$
FactorModelSet	$q^0 \in \mathbb{R}^n$, $\Psi \in \mathbb{R}^{n \times F}$, $\beta \in [0,1]$	$\left\{ q \in \mathbb{R}^n \,\middle|\, \begin{array}{l} \exists \, \xi \in [-1,1]^F : \\ \quad q = q^0 + \Psi \xi \\ \quad \left\lvert \sum\limits_{j=1}^{F} \xi_j \right\rvert \leq \beta F \end{array} \right\}$
IntersectionSet	$\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_m \subset \mathbb{R}^n$	$\bigcap\limits_{i=1}^{m} \mathcal{Q}_i$
PolyhedralSet	$A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$	$\{q \in \mathbb{R}^n \mid Aq \leq b\}$

Custom Uncertainty Set Types

A custom uncertainty set type in which all uncertain parameters are modeled as continuous quantities can be implemented by subclassing *UncertaintySet*. For discrete sets, we recommend using the pre-implemented *DiscreteScenarioSet* subclass instead of implementing a new set type. PyROS does not support mixed-integer uncertainty set types.

PyROS Solver Output Log

Table of Contents

- *Default Format*
- *Configuring the Output Log*

Default Format

When the PyROS `solve()` method is called to solve a robust optimization problem, your console output will, by default, look like this:

Listing 3.1: PyROS solver output log for the *Quickstart example*.

```
==============================================================================
PyROS: The Pyomo Robust Optimization Solver, v1.3.11.
       Pyomo version: 6.9.5
       Commit hash: unknown
       Invoked at UTC 2025-10-18T00:00:00.000000+00:00

Developed by: Natalie M. Isenberg (1), Jason A. F. Sherman (1),
              John D. Siirola (2), Chrysanthos E. Gounaris (1)
(1) Carnegie Mellon University, Department of Chemical Engineering
(2) Sandia National Laboratories, Center for Computing Research

The developers gratefully acknowledge support from the U.S. Department
of Energy's Institute for the Design of Advanced Energy Systems (IDAES).
==============================================================================
================================= DISCLAIMER =================================
PyROS is currently under active development.
Please provide feedback and/or report any issues by creating a ticket at
https://github.com/Pyomo/pyomo/issues/new/choose
==============================================================================
User-provided solver options:
 objective_focus=<ObjectiveType.worst_case: 1>
 decision_rule_order=1
 solve_master_globally=True
------------------------------------------------------------------------------
Model Statistics (before preprocessing):
  Number of variables : 4
    First-stage variables : 1
    Second-stage variables : 1
    State variables : 2
  Number of uncertain parameters : 2
  Number of constraints : 4
    Equality constraints : 2
    Inequality constraints : 2
------------------------------------------------------------------------------
Preprocessing...
Done preprocessing; required wall time of 0.004s.
------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Itn  Objective    1-Stg Shift  2-Stg Shift  #CViol  Max Viol     Wall Time (s)
------------------------------------------------------------------------------
0     5.4079e+03  -            -            3       7.9226e+00   0.209
1     5.4079e+03  6.0451e-10   1.0717e-10   2       1.0250e-01   0.476
2     6.5403e+03  1.0018e-01   7.4564e-03   1       1.0249e-02   0.786
3     6.5403e+03  1.9372e-16   2.0321e-05   2       8.7074e-03   1.132
4     6.5403e+03  0.0000e+00   2.0311e-05   0       1.2310e-06g  1.956
------------------------------------------------------------------------------
Robust optimal solution identified.
------------------------------------------------------------------------------
Termination stats:
 Iterations            : 5
 Solve time (wall s)   : 1.956
 Final objective value : 6.5403e+03
 Termination condition : pyrosTerminationCondition.robust_optimal
------------------------------------------------------------------------------
All done. Exiting PyROS.
==============================================================================
```

Observe that the log contains the following information (listed in order of appearance):

- **Introductory information and disclaimer** (lines 1–19): Includes the version number, author information, (UTC) time at which the solver was invoked, and, if available, information on the local Git branch and commit hash.
- **Summary of solver options** (lines 20–24): Enumeration of specifications for optional arguments to the solver.
- **Model component statistics** (lines 25–34): Breakdown of component statistics for the user-provided model and variable selection (before preprocessing).
- **Preprocessing information** (lines 35–37): Wall time required for preprocessing the deterministic model and associated components, i.e., standardizing model components and adding the decision rule variables and equations.
- **Iteration log table** (lines 38–45): Summary information on the problem iterates and subproblem outcomes. The constituent columns are defined in detail in the table that follows.
- **Termination message** (lines 46–47): One-line message briefly summarizing the reason the solver has terminated.
- **Final result** (lines 48–53): A printout of the `ROSolveResults` object that is finally returned.
- **Exit message** (lines 54–55): Confirmation that the solver has been exited properly.

The iteration log table is designed to provide, in a concise manner, important information about the progress of the iterative algorithm for the problem of interest. The constituent columns are defined in the table below.

Table 3.7: PyROS iteration log table columns.

Column Name	Definition
Itn	Iteration number, equal to one less than the total number of elapsed iterations.
Objective	Master solution objective function value. If the objective of the deterministic model provided has a maximization sense, then the negative of the objective function value is displayed. Expect this value to trend upward as the iteration number increases. A dash ("-") is produced in lieu of a value if the master problem of the current iteration is not solved successfully.
1-Stg Shift	Infinity norm of the relative difference between the first-stage variable vectors of the master solutions of the current and previous iterations. Expect this value to trend downward as the iteration number increases. A dash ("-") is produced in lieu of a value if the current iteration number is 0, there are no first-stage variables, or the master problem of the current iteration is not solved successfully.
2-Stg Shift	Infinity norm of the relative difference between the second-stage variable vectors (evaluated subject to the nominal uncertain parameter realization) of the master solutions of the current and previous iterations. Expect this value to trend downward as the iteration number increases. A dash ("-") is produced in lieu of a value if the current iteration number is 0, there are no second-stage variables, or the master problem of the current iteration is not solved successfully. An asterisk ("*") is appended to the value if decision rule polishing was unsuccessful.
#CViol	Number of second-stage inequality constraints found to be violated during the separation step of the current iteration. Unless a custom prioritization of the model's second-stage inequality constraints is specified (through the `separation_priority_order` argument), expect this number to trend downward as the iteration number increases. A "+" is appended if not all of the separation problems were solved successfully, either due to custom prioritization, a time out, or an issue encountered by the subordinate optimizers. A dash ("-") is produced in lieu of a value if the separation routine is not invoked during the current iteration.
Max Viol	Maximum scaled second-stage inequality constraint violation. Expect this value to trend downward as the iteration number increases. A 'g' is appended to the value if the separation problems were solved globally during the current iteration. A dash ("-") is produced in lieu of a value if the separation routine is not invoked during the current iteration, or if there are no second-stage inequality constraints.
Wall time (s)	Total time elapsed by the solver, in seconds, up to the end of the current iteration.

Configuring the Output Log

The PyROS solver output log is produced by the Python logger (`logging.Logger`) object derived from the optional argument `progress_logger` to the PyROS *`solve()`* method. By default, the PyROS solver argument `progress_logger` is taken to be the `logging.INFO`-level logger with name `"pyomo.contrib.pyros"`. The verbosity level of the output log can be adjusted by setting the `logging` level of the progress logger. For example, the level of the default logger can be adjusted to `logging.DEBUG` as follows:

```
import logging
logging.getLogger("pyomo.contrib.pyros").setLevel(logging.DEBUG)
```

We refer the reader to the official Python logging library documentation for further guidance on (customization of) Python logger objects.

The *following table* describes the information logged by PyROS at the various `logging` levels. Messages of a lower logging level than that of the progress logger are excluded from the solver log.

Table 3.8: PyROS solver log output at the various standard Python `logging` levels.

Logging Level	Output Messages
`logging.ERROR`	• Elaborations of exceptions stemming from expression evaluation errors or issues encountered by the subordinate solvers
`logging.WARNING`	• Elaboration of unacceptable subproblem termination statuses for critical subproblems • Caution about solution robustness guarantees in event that user passes `bypass_global_separation=True`
`logging.INFO`	• PyROS version, author, and disclaimer information • Summary of user options • Model component statistics (before preprocessing) • Summary of preprocessing outcome • Iteration log table • Termination message and summary statistics • Exit message
`logging.DEBUG`	• Detailed progress through the various preprocessing subroutines • Detailed component statistics for the preprocessed model • Termination outcomes, backup solver invocation statements, and summaries of results for all subproblems • Summary of separation subroutine overall outcomes: second-stage inequality constraints violated and uncertain parameter realization(s) added to the master problem • Solve time profiling statistics

PyROS Usage Tutorial

Table of Contents

- *Setup*
- *Prepare the Deterministic Model*
  - *Formulate the Model*
  - *Implement the Model*
  - *Solve the Model Deterministically*
- *Assess Impact of Parametric Uncertainty*
- *Use PyROS to Obtain Robust Solutions*
  - *Import PyROS*
  - *Construct PyROS Solver Arguments*
    - *Deterministic Model*
    - *First-stage Variables and Second-Stage Variables*

- *Uncertain Parameters and Uncertainty Set*
- *Subsolvers*
- *Invoke PyROS*
- *Inspect the Solution*
- *Try Higher-Order Decision Rules to Improve Solution Quality*
- *Assess Impact of Uncertainty Set on the Solution Obtained*
  - *Invoke PyROS in a `for` Loop*
  - *Visualize the Results*
  - *Assess Robust Feasibility of the Solutions*

This tutorial is an in-depth guide on how to use PyROS to solve a two-stage robust optimization problem. The problem is taken from the area of chemical process systems design.

Setup

To successfully run this tutorial, you will need to *install PyROS* along with at least one local nonlinear programming (NLP) solver and at least one global NLP solver. In particular, this tutorial uses IPOPT as the local solver, BARON as the global solver, and COUENNE as a backup global solver.

You will also need to install Matplotlib, which is used to generate plots in this tutorial. Further, we recommend installing an interactive Matplotlib backend for quick and easy viewing of plots.

Prepare the Deterministic Model

PyROS is designed to operate on a user-supplied deterministic NLP. We now set out to prepare a deterministic NLP that can be solved tractably with subordinate NLP optimizers.

Formulate the Model

Consider the reactor-cooler system below.

A stream of chemical species E enters the reactor with a molar flow rate $F_0 = 45.36\,\text{kmol/h}$, absolute temperature $T_0 = 333\,\text{K}$, concentration $\boldsymbol{c}_{E0} = 32.04\,\text{kmol}/\text{m}^3$, and heat capacity $\boldsymbol{c}_p = 167.4\,\text{kJ}/\text{kmol}\,\text{K}$. Inside the reactor, the exothermic reaction $E \to F$ occurs at temperature T_1 and with a conversion of 90%, so that $\boldsymbol{c}_{E1} = 0.1\boldsymbol{c}_{E0}$. We assume that the reaction follows first-order kinetics, with a rate constant $\boldsymbol{k}_\text{R}$ of nominal value $10\,\text{hr}^{-1}$ and normalized activation energy $\boldsymbol{E}/\boldsymbol{R} = 555.6\,\text{K}$. Further, the molar heat of reaction is $\boldsymbol{\Delta H}_\text{R} = -23260\,\text{kJ/kmol}$.

A portion of the product is cooled to a temperature T_2 then recycled to the reactor. Cooling water, with heat capacity $\boldsymbol{c}_{w,p} = 4.184\,\text{kJ}/\text{kg}\,\text{K}$ and inlet temperature $\boldsymbol{T}_{w1} = 300\,\text{K}$, is used as the cooling medium. The heat transfer coefficient of the cooling unit is of nominal value $\boldsymbol{U} = 1635\,\text{kJ}/\text{m}^2\,\text{kg}\,\text{h}$.

We are interested in optimizing the design and operation of the system:

- The design is specified through the reactor volume $\hat{V}$ and the area A of the heat exchanger used to cool the recycle stream.
- The operation of the system can be adjusted via the reactor outlet temperature T_1 and recycle stream flow rate F_1.

Modified from the formulations presented in [HG83], [YLH18], [Dje20], and [IAE+21], the deterministic optimization

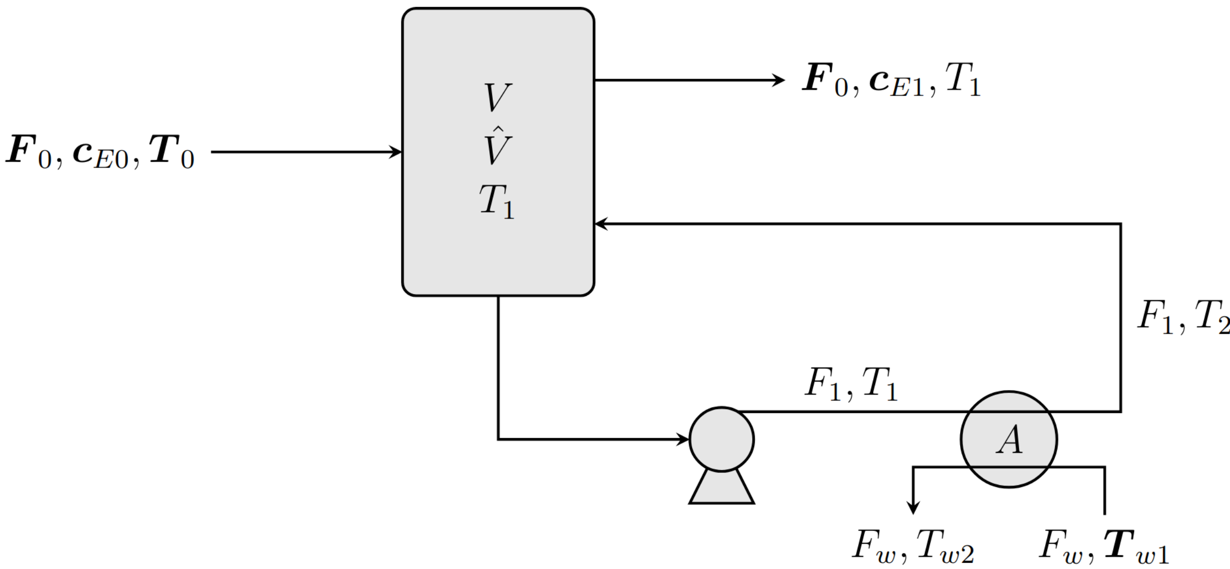


Fig. 3.2: Reactor-cooler system process flowsheet, adapted from [Dje20]. Constants are set in boldface.

model for the system of interest can be written as

$$
\begin{aligned}
\min_{\substack{(\hat{V}, A, c_{E1}) \in [0,10] \times [0,20] \\ (F_1, T_1) \in \mathbb{R}^3 \\ (T_2, T_{w2}, \dot{Q}_{\text{HE}}, F_w, V) \in \mathbb{R}^5}} \quad & 691.2\hat{V}^{0.7} + 873.6A^{0.6} + 7.056F_1 + 1.760F_w \\
\text{s.t.} \quad & \boldsymbol{F}_0 \left(\frac{\boldsymbol{c}_{E0} - \boldsymbol{c}_{E1}}{\boldsymbol{c}_{E0}} \right) = V \boldsymbol{k}_{\text{R}} \exp\left(-\frac{\boldsymbol{E}}{\boldsymbol{R}T_1} \right) \boldsymbol{c}_{E1} \\
& -\boldsymbol{\Delta H}_{\text{R}} \boldsymbol{F}_0 \left(\frac{\boldsymbol{c}_{E0} - \boldsymbol{c}_{E1}}{\boldsymbol{c}_{E0}} \right) = \boldsymbol{F}_0 \boldsymbol{c}_p (T_1 - \boldsymbol{T}_0) + \dot{Q}_{\text{HE}} \\
& \dot{Q}_{\text{HE}} = F_1 \boldsymbol{c}_p (T_1 - T_2) \\
& \dot{Q}_{\text{HE}} = F_w \boldsymbol{c}_{w,p} (T_{w2} - \boldsymbol{T}_{w1}) \\
& \dot{Q}_{\text{HE}} = \boldsymbol{U} A \left(\frac{(T_1 - T_{w2}) - (T_2 - \boldsymbol{T}_{w1})}{\ln(T_1 - T_{w2}) - \ln(T_2 - \boldsymbol{T}_{w1})} \right) \\
& 0 \leq V \leq \hat{V} \\
& 311 \leq T_1, T_2 \leq 389 \\
& 301 \leq T_{w2} \leq 355 \\
& 1 \leq T_1 - T_2 \\
& 1 \leq T_{w2} - \boldsymbol{T}_{w1} \\
& 11.1 \leq T_1 - T_{w2} \\
& 11.1 \leq T_2 - \boldsymbol{T}_{w1}
\end{aligned}
$$

in which:

- V is the reactor holdup volume during operation (in m^3)
- F_w is the mass flow rate of cooling water (in kg/hr)
- T_2 is the temperature to which the recycle stream is cooled (in K)
- T_{w2} is the cooling water return temperature (in K)
- $\dot{Q}_{\text{HE}}$ is the cooling duty of the heat exchanger (in kW)

The objective function yields the total annualized cost, with units of $/yr.

Once the design $(\hat{V}, A)$ and operation (F_1, T_1) of the system are specified, the state variables $(V, \dot{Q}_{\text{HE}}, F_w, T_{w2}, T_2)$ are calculated using the equality constraints, which comprise a square system of nonlinear equations.

Implement the Model

We now implement the deterministic model for the reactor-cooler system. First, we import Pyomo:

```
>>> import pyomo.environ as pyo
```

and write a function for building the model (with the variables uninitialized):

```
>>> def build_model():
...     m = pyo.ConcreteModel()
...
...     # certain parameters
...     m.cA0 = pyo.Param(initialize=32.040)
...     m.cA1 = pyo.Param(initialize=0.1 * m.cA0)
...     m.EovR = pyo.Param(initialize=555.6)
...     m.delHr = pyo.Param(initialize=-23260)
...     m.cp = pyo.Param(initialize=167.400)
...     m.cwp = pyo.Param(initialize=4.184)
...     m.F0 = pyo.Param(initialize=45.36)
...     m.T0 = pyo.Param(initialize=333)
...     m.Tw1 = pyo.Param(initialize=300)
...
...     # uncertain parameters
...     m.kR = pyo.Param(initialize=10, mutable=True)
...     m.U = pyo.Param(initialize=1635, mutable=True)
...
...     # first-stage variables
...     m.Vhat = pyo.Var(bounds=(0, 20))
...     m.A = pyo.Var(bounds=(0, 10))
...
...     # second-stage variables
...     m.F1 = pyo.Var()
...     m.T1 = pyo.Var(bounds=(311, 389))
...
...     # state variables
...     m.V = pyo.Var(bounds=(0, None))
...     m.Qhe = pyo.Var()
...     m.T2 = pyo.Var(bounds=(311, 389))
...     m.Tw2 = pyo.Var(bounds=(301, 355))
...     m.Fw = pyo.Var()
...
...     # objective and constituent expressions
...     m.capex = pyo.Expression(expr=691.2 * m.Vhat ** 0.7 + 873.6 * m.A ** 0.6)
...     m.opex = pyo.Expression(expr=1.76 * m.Fw + 7.056 * m.F1)
...     m.obj = pyo.Objective(expr=m.capex + m.opex)
...
...     # equality constraints
...     m.reactant_mol_bal = pyo.Constraint(
...         expr=(
...             m.F0 * ((m.cA0 - m.cA1) / m.cA0)
...             == m.V * m.kR * pyo.exp(-m.EovR / m.T1) * m.cA1
```

(continues on next page)

(continued from previous page)

```
...         ),
...     )
...     m.reactant_heat_bal = pyo.Constraint(
...         expr=(
...             -m.delHr * m.F0 * ((m.cA0 - m.cA1) / m.cA0)
...             == m.F0 * m.cp * (m.T1 - m.T0)
...             + m.Qhe
...         )
...     )
...     m.heat_bal_process = pyo.Constraint(
...         expr=m.Qhe == m.F1 * m.cp * (m.T1 - m.T2)
...     )
...     m.heat_bal_util = pyo.Constraint(
...         expr=m.Qhe == m.Fw * m.cwp * (m.Tw2 - m.Tw1)
...     )
...
...     @m.Constraint()
...     def hex_design_eq(mdl):
...         dt1 = mdl.T1 - mdl.Tw2
...         dt2 = mdl.T2 - mdl.Tw1
...         lmtd_expr = (dt1 - dt2) / (pyo.log(dt1) - pyo.log(dt2))
...         return m.Qhe == m.A * m.U * lmtd_expr
...
...     # inequality constraints
...     m.V_con = pyo.Constraint(expr=(m.V <= m.Vhat))
...     m.T1T2_con = pyo.Constraint(expr=(1 <= m.T1 - m.T2))
...     m.Tw1Tw2_con = pyo.Constraint(expr=(1 <= m.Tw2 - m.Tw1))
...     m.T1Tw2_con = pyo.Constraint(expr=(11.1 <= m.T1 - m.Tw2))
...     m.T2Tw1_con = pyo.Constraint(expr=(11.1 <= m.T2 - m.Tw1))
...
...     return m
...
```

Note

Observe that the *Param* objects representing the potentially uncertain parameters k_R and U are declared with the argument `mutable=True`, as PyROS requires that *Param* objects representing uncertain parameters be mutable. Alternatively, k_R and U may have instead been implemented as fixed *Var* objects, as follows:

```
m.kR = pyo.Var(initialize=10)
m.U = pyo.Var(initialize=1635)
m.kR.fix(); m.U.fix()
```

For more information on implementing uncertain parameters for PyROS, see the *Uncertain Parameters section of the Solver Interface documentation*.

For convenience, we also write a function to initialize the model's variable values:

```
>>> from pyomo.util.calc_var_value import calculate_variable_from_constraint
>>>
>>> def initialize_model(m, Vhat=20, A=10, F1=50, T1=389):
```

(continues on next page)

(continued from previous page)

```
...     # set first-stage and second-stage variable values
...     m.Vhat.set_value(Vhat)
...     m.A.set_value(A)
...     m.F1.set_value(F1)
...     m.T1.set_value(T1)
...
...     # solve equations for state variables
...     calculate_variable_from_constraint(
...         variable=m.V,
...         constraint=m.reactant_mol_bal,
...     )
...     calculate_variable_from_constraint(
...         variable=m.Qhe,
...         constraint=m.reactant_heat_bal,
...     )
...     calculate_variable_from_constraint(
...         variable=m.T2,
...         constraint=m.heat_bal_process,
...     )
...     calculate_variable_from_constraint(
...         variable=m.Tw2,
...         constraint=m.hex_design_eq,
...     )
...     calculate_variable_from_constraint(
...         variable=m.Fw,
...         constraint=m.heat_bal_util,
...     )
...
```

And finally, a function to build the model and initialize the variable values:

```
>>> def build_and_initialize_model(**init_kwargs):
...     m = build_model()
...     initialize_model(m, **init_kwargs)
...     return m
...
```

We may now instantiate and initialize the model as follows:

```
>>> m = build_and_initialize_model()
```

The following helper function will be useful for inspecting the current solution:

```
>>> def print_solution(model):
...     print(f"Objective      ($/yr)    : {pyo.value(model.obj):.2f}")
...     print(f"Reactor volume (m^3)     : {model.Vhat.value:.2f}")
...     print(f"Cooler area    (m^2)     : {model.A.value:.2f}")
...     print(f"F1             (kmol/hr) : {model.F1.value:.2f}")
...     print(f"T1             (K)       : {model.T1.value:.2f}")
...     print(f"Fw             (kg/hr)   : {model.Fw.value:.2f}")
...
```

Inspecting the initial model solution:

```
>>> print_solution(m)  # may vary
Objective      ($/yr)    : 13830.89
Reactor volume (m^3)     : 20.00
Cooler area    (m^2)     : 10.00
F1             (kmol/hr) : 50.00
T1             (K)       : 389.00
Fw             (kg/hr)   : 2484.43
```

Solve the Model Deterministically

We use IPOPT to solve the model to local optimality:

```
>>> ipopt = pyo.SolverFactory("ipopt")
>>> pyo.assert_optimal_termination(ipopt.solve(m, tee=True, load_solutions=True))
Ipopt ...
...
EXIT: Optimal Solution Found.
```

We are able to solve the model to local optimality. Inspecting the solution, we notice reductions in the objective and the main variables of interest compared to the initial point used:

```
>>> print_solution(m)  # may vary
Objective      ($/yr)    : 9774.58
Reactor volume (m^3)     : 5.32
Cooler area    (m^2)     : 7.45
F1             (kmol/hr) : 88.32
T1             (K)       : 389.00
Fw             (kg/hr)   : 2278.57
```

Assess Impact of Parametric Uncertainty

Suppose the reaction rate constant $\boldsymbol{k}_{\mathrm{R}}$ and heat transfer coefficient $\boldsymbol{U}$ are uncertain. We assume that each parameter may deviate from its nominal value by up to 5%, and that the deviations are independent. Thus, the joint realizations of the uncertain parameters are confined to a rectangular region, that is, a box.

Given a *fixed* design $(\hat{V}, A)$, we wish to assess whether we can guarantee that the operational variables (F_1, T_1), and concomitantly, the state $(V, \dot{Q}_{\mathrm{HE}}, T_2, T_{w2}, F_w)$, can be adjusted to a feasible solution under any plausible realization of the uncertain parameters. This assessment can be carried out with the following function:

```
>>> # module imports needed
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import matplotlib.patches as patches
>>>
>>> def plot_feasibility(solutions, solver, samples=200, test_set_size=10):
...     # seed the random number generator for deterministic sampling
...     rng = np.random.default_rng(123456)
...
...     # nominal uncertain parameter realizations
...     nom_vals = np.array([10, 1635])
...
...     # sample points from box uncertainty set of specified test size
...     point_samples = np.empty((samples, 2))
```

(continues on next page)

(continued from previous page)

```
...     point_samples[0] = nom_vals
...     point_samples[1:] = rng.uniform(
...         low=nom_vals * (1 - test_set_size / 100),
...         high=nom_vals * (1 + test_set_size / 100),
...         size=(samples - 1, 2),
...     )
...
...     costs = np.empty((len(solutions), point_samples.shape[0]), dtype=float)
...     mdl = build_model()
...     for sol_idx, (size, sol) in enumerate(solutions.items()):
...         # fix the first-stage variables
...         mdl.Vhat.fix(sol[0])
...         mdl.A.fix(sol[1])
...
...         for pt_idx, pt in enumerate(point_samples):
...             # update parameter realization to sampled point
...             mdl.kR.set_value(pt[0])
...             mdl.U.set_value(pt[1])
...
...             # update the values of the operational variables
...             initialize_model(mdl, Vhat=sol[0], A=sol[1])
...
...             # try solving the model to inspect for feasibility
...             res = solver.solve(mdl, load_solutions=False)
...             if pyo.check_optimal_termination(res):
...                 mdl.solutions.load_from(res)
...                 costs[sol_idx, pt_idx] = pyo.value(mdl.obj)
...             else:
...                 costs[sol_idx, pt_idx] = np.nan
...
...     # now generate the plot(s)
...     fig, axs = plt.subplots(
...         figsize=(0.5 * (len(solutions) - 1) + 5 * len(solutions), 4),
...         ncols=len(solutions),
...         squeeze=False,
...         sharey=True,
...         layout="constrained",
...     )
...     for sol_idx, (size, ax) in enumerate(zip(solutions, axs[0])):
...         # track realizations for which solution feasible
...         is_feas = ~np.isnan(costs[sol_idx])
...
...         # realizations under which design is feasible
...         # (colored by objective)
...         plot = ax.scatter(
...             point_samples[is_feas][:, 0],
...             point_samples[is_feas][:, 1],
...             c=costs[sol_idx, is_feas],
...             vmin=np.nanmin(costs),
...             vmax=np.nanmax(costs),
...             cmap="plasma_r",
...             marker="o",
```

(continues on next page)

(continued from previous page)

```
...         )
...         # realizations under which design is infeasible
...         ax.scatter(
...             point_samples[~is_feas][:, 0],
...             point_samples[~is_feas][:, 1],
...             color="none",
...             edgecolors="black",
...             label="infeasible",
...             marker="^",
...         )
...         if size != 0:
...             # boundary of the box uncertainty set mapped to the design
...             rect = patches.Rectangle(
...                 nom_vals * (1 - size / 100),
...                 *tuple(nom_vals * 2 * size / 100),
...                 facecolor="none",
...                 edgecolor="black",
...                 linestyle="dashed",
...                 label=f"{size}% box set",
...             )
...             ax.add_patch(rect)
...
...         ax.legend(bbox_to_anchor=(1, -0.15), loc="upper right")
...         ax.set_xlabel(r"$k_\mathrm{R}$ (per hr)")
...         ax.set_ylabel("$U$ (kJ/sqm-h-K)")
...
...         is_in_set = np.logical_and(
...             np.all(nom_vals * (1 - size / 100) <= point_samples, axis=1),
...             np.all(point_samples <= nom_vals * (1 + size / 100), axis=1),
...         )
...         feas_in_set = np.logical_and(is_feas, is_in_set)
...
...         # add plot title summarizing statistics of the results
...         ax.set_title(
...             f"Solution for {size}% box set\n"
...             "Avg ± SD objective "
...             f"{costs[sol_idx, is_feas].mean():.2f} ± {costs[sol_idx, is_feas].
↪std():.2f}\n"
...             f"Feas. for {feas_in_set.sum()}/{is_in_set.sum()} samples in set\n"
...             f"Feas. for {is_feas.sum()}/{len(point_samples)} samples overall"
...         )
...
...     cbar = fig.colorbar(plot, ax=axs.ravel().tolist(), pad=0.03)
...     cbar.ax.set_ylabel("Objective ($/yr)")
...
...     plt.show()
...
```

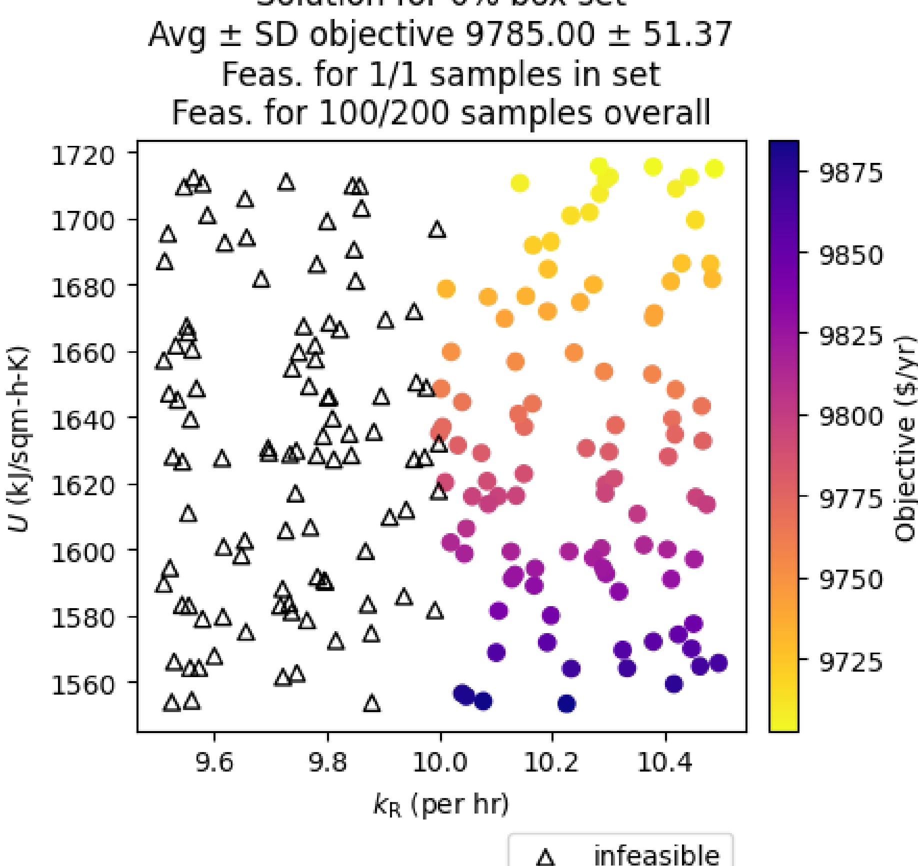
Applying this function to the design that was deterministically optimized subject to the nominal realization of the uncertain parameters:

```
>>> plot_feasibility(
```

(continues on next page)

(continued from previous page)

```
...     # design variable values
...     solutions={0: (m.Vhat.value, m.A.value)},
...     # solver to use for feasibility testing
...     solver=ipopt,
...     # size of the uncertainty set (percent maximum deviation from nominal)
...     test_set_size=5,
... )
```



Clearly, the nominally optimal $(\hat{V}, A)$ is robust infeasible, as the operation of the system cannot be feasibly adjusted subject to approximately half of the tested scenarios. Observe that infeasibility occurs subject to parameter realizations in which the rate constant $\boldsymbol{k}_{\mathrm{R}}$ is below its nominal value. This suggests that for such realizations, the design $(\hat{V}, A)$ is not sufficiently large to allow for the 90% reactor conversion requirement to be met.

Use PyROS to Obtain Robust Solutions

We have just confirmed that the nominally optimal design for the reactor cooler system is robust infeasible. Thus, we now use PyROS to optimize the design while explicitly accounting for the impact of parametric uncertainty.

Import PyROS

We will need to import the PyROS module in order to instantiate the solver and required arguments:

```
>>> import pyomo.contrib.pyros as pyros
```

Construct PyROS Solver Arguments

We now construct the required arguments to the PyROS solver. A general discussion on all PyROS solver arguments is given in the *Solver Interface documentation*.

Deterministic Model

We have already constructed the deterministic model.

First-stage Variables and Second-Stage Variables

As previously discussed, the first-stage variables comprise the design variables $(\hat{V}, A)$, whereas the second-stage variables comprise the operational decision variables (F_1, T_1). PyROS automatically infers the state variables of the model by inspecting the active objective and constraint components.

```
>>> first_stage_variables = [m.A, m.Vhat]
>>> second_stage_variables = [m.F1, m.T1]
```

Uncertain Parameters and Uncertainty Set

Following from our prior feasibility analysis, we take $\boldsymbol{k}_\mathrm{R}$ and $\boldsymbol{U}$ to be the uncertain parameters, confined in value to a box set, such that each parameter may deviate from its nominal value by up to 5%. Thus, we compile the *Param* objects representing $\boldsymbol{k}_\mathrm{R}$ and $\boldsymbol{U}$ into a list and represent the uncertainty set with an instance of the PyROS *BoxSet* class:

```
>>> uncertain_params = [m.kR, m.U]
>>> uncertainty_set = pyros.BoxSet(bounds=[
...     [param.value * (1 - 0.05), param.value * (1 + 0.05)] for param in uncertain_
↪params
... ])
```

Subsolvers

PyROS requires subordinate deterministic NLP optimizers to solve the subproblems of its underlying algorithm. At least one local NLP solver and one global NLP solver are required. We will use IPOPT (already constructed) as the local NLP subsolver and BARON as the global NLP subsolver. For subproblems not solved successfully by BARON, we use COUENNE as a backup.

```
>>> # already constructed local subsolver IPOPT.
>>> # global subsolvers:
>>> baron = pyo.SolverFactory("baron", options={"MaxTime": 10})
>>> couenne = pyo.SolverFactory("couenne", options={"max_cpu_time": 10})
```

Invoke PyROS

We are now ready to invoke PyROS on our model. We do so by instantiating the PyROS solver interface:

```
>>> pyros_solver = pyo.SolverFactory("pyros")
```

and invoking the *solve()* method:

```
>>> pyros_solver.solve(
...     # mandatory arguments
...     model=m,
...     first_stage_variables=first_stage_variables,
...     second_stage_variables=second_stage_variables,
...     uncertain_params=uncertain_params,
...     uncertainty_set=uncertainty_set,
...     local_solver=ipopt,
...     global_solver=baron,
...     # optional arguments
...     backup_global_solvers=[couenne],
... )
==============================================================================
PyROS: The Pyomo Robust Optimization Solver, ...
...
Robust feasible solution identified.
...
All done. Exiting PyROS.
==============================================================================
<pyomo.contrib.pyros.solve_data.ROSolveResults at ...>
```

By default, the progress and final result of the PyROS solver is logged to the console. The *Solver Output Log documentation* provides guidance on how the solver log is to be interpreted. The *solve()* method returns an *ROSolveResults* object summarizing the results.

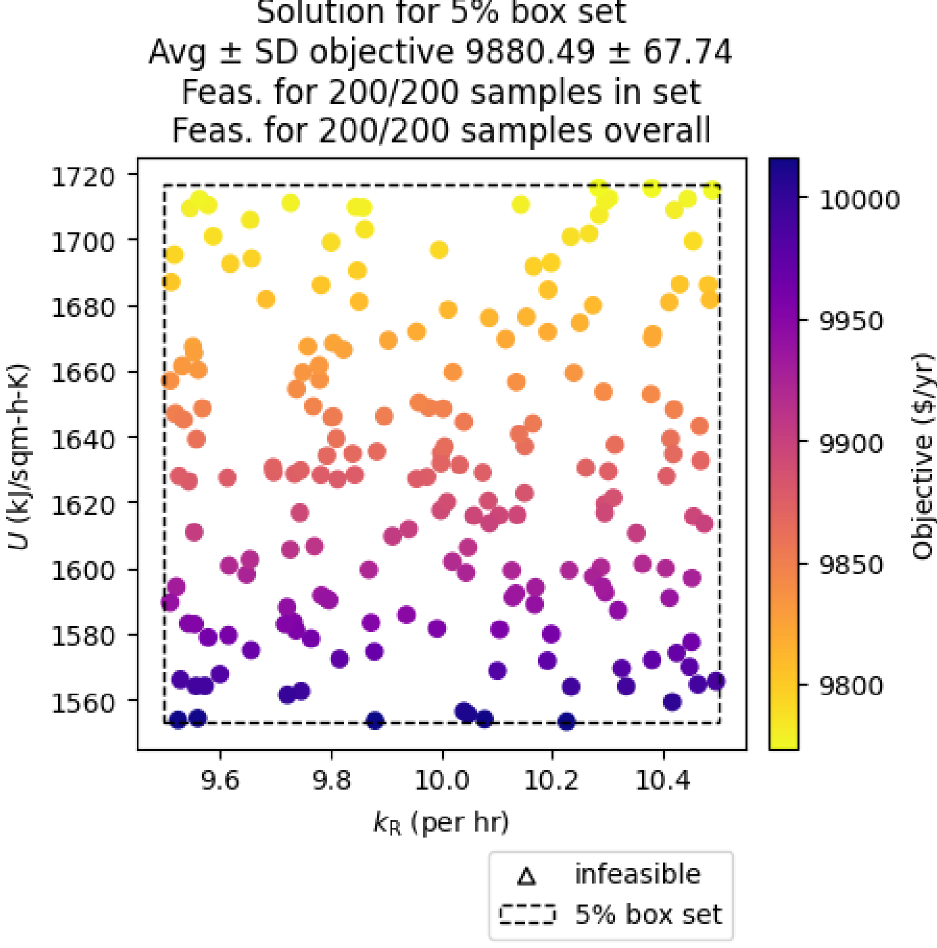
Inspect the Solution

Inspecting the solution, we see that the overall objective is increased compared to when the model is *solved deterministically*. The cooler area A and recycle stream flow F_1 are reduced, but the reactor volume $\hat{V}$ and cooling water utility flow rate F_w are increased:

```
>>> print_solution(m)  # may vary
Objective      ($/yr)    : 10064.11
Reactor volume (m^3)     : 5.59
Cooler area    (m^2)     : 7.19
F1             (kmol/hr) : 85.39
T1             (K)       : 389.00
Fw             (kg/hr)   : 2444.42
```

We can also confirm the robust feasibility of the solution empirically:

```
>>> plot_feasibility({5: (m.Vhat.value, m.A.value)}, solver=ipopt, test_set_size=5)
```



Try Higher-Order Decision Rules to Improve Solution Quality

For tractability purposes, the underlying algorithm of PyROS uses polynomial decision rules to approximate (restrict) the adjustability of the second-stage decision variables (that is, F_1 and T_1 for the present model) to the uncertain parameters. By default, a static approximation is used, such that the second-stage decisions are modeled as nonadjustable. A less restrictive approximation can be used by increasing the order of the decision rules to 1, through the optional argument `decision_rule_order`:

```
>>> pyros_solver.solve(
...     # mandatory arguments
...     model=m,
...     first_stage_variables=first_stage_variables,
...     second_stage_variables=second_stage_variables,
...     uncertain_params=uncertain_params,
...     uncertainty_set=uncertainty_set,
...     local_solver=ipopt,
...     global_solver=baron,
...     # optional arguments
...     backup_global_solvers=[couenne],
...     decision_rule_order=1,
... )
```

(continues on next page)

(continued from previous page)

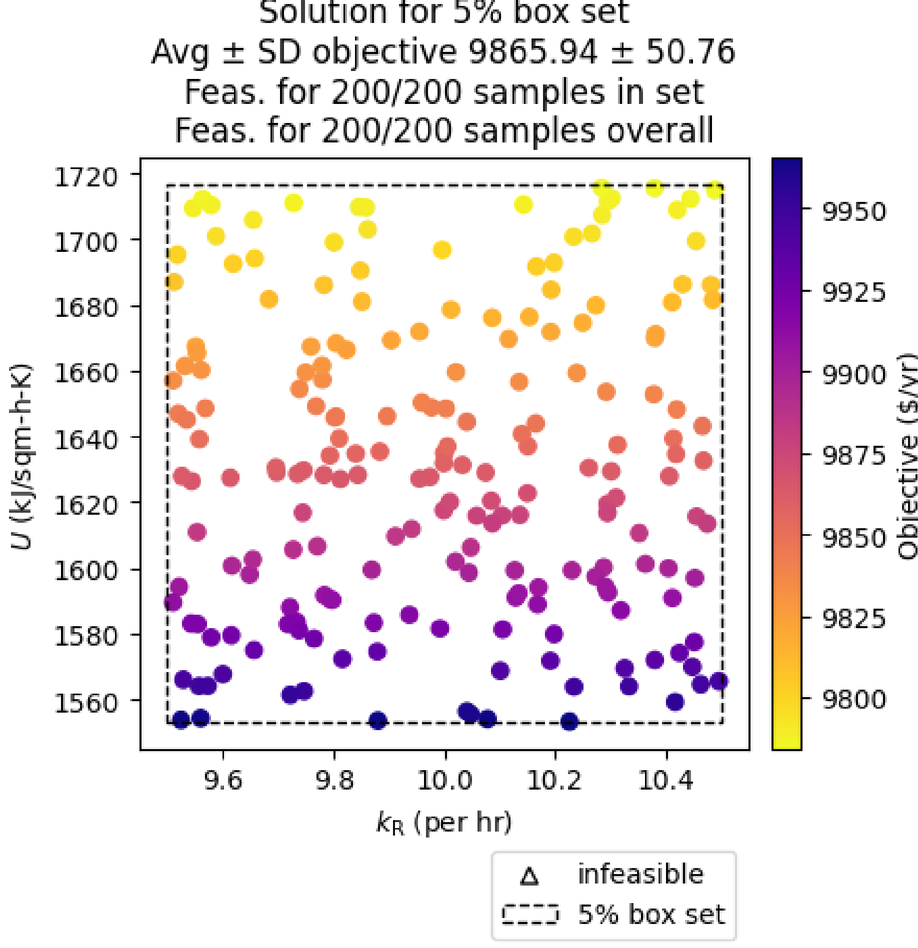
```
==============================================================================
PyROS: The Pyomo Robust Optimization Solver, ...
...
Robust feasible solution identified.
...
All done. Exiting PyROS.
==============================================================================
<pyomo.contrib.pyros.solve_data.ROSolveResults at ...>
```

Inspecting the solution, we see that the cost is reduced compared to when *a static decision rule approximation is used*, as a smaller cooling water flow rate F_w is required since the cooler area A is increased:

```
>>> print_solution(m)  # may vary
Objective      ($/yr)    : 9855.95
Reactor volume (m^3)     : 5.59
Cooler area    (m^2)     : 7.45
F1             (kmol/hr) : 88.32
T1             (K)       : 389.00
Fw             (kg/hr)   : 2278.57
```

Further, our empirical check confirms that the solution is robust:

```
>>> plot_feasibility({5: (m.Vhat.value, m.A.value)}, solver=ipopt,  test_set_size=5)
```



Assess Impact of Uncertainty Set on the Solution Obtained

We now perform a price-of-robustness study, in which we assess the response of the solution obtained to the size of the uncertainty set. This study can be easily performed by placing the PyROS solver invocation in a `for` loop and recording the result at each iteration.

Invoke PyROS in a `for` Loop

The PyROS solver invocation can easily be made in a `for` loop. At each iteration of the loop, we use PyROS to solve the RO problem subject to the uncertainty set of the corresponding size:

```
>>> res_dict = dict()
>>> obj_vals = dict()
>>> capex_vals = dict()
>>> opex_vals = dict()
>>> vhat_vals = dict()
>>> area_vals = dict()
>>> for percent_size in [0, 2.5, 5, 7.5, 10]:
...     mdl = build_and_initialize_model()
...     unc_set = pyros.BoxSet(bounds=[
...         [param.value * (1 - percent_size / 100), param.value * (1 + percent_size /␣
↪100)]
```

(continues on next page)

(continued from previous page)

```
...         for param in [mdl.kR, mdl.U]
...     ])
...     print(f"Solving RO problem for uncertainty set size {percent_size}:")
...     res_dict[percent_size] = res = pyros_solver.solve(
...         model=mdl,
...         first_stage_variables=[mdl.Vhat, mdl.A],
...         second_stage_variables=[mdl.F1, mdl.T1],
...         uncertain_params=[mdl.kR, mdl.U],
...         uncertainty_set=unc_set,
...         local_solver=ipopt,
...         global_solver=baron,
...         decision_rule_order=1,
...         backup_global_solvers=[couenne],
...     )
...     if res.pyros_termination_condition == pyros.pyrosTerminationCondition.robust_
↪feasible:
...         obj_vals[percent_size] = pyo.value(mdl.obj)
...         capex_vals[percent_size] = pyo.value(mdl.capex)
...         opex_vals[percent_size] = pyo.value(mdl.opex)
...         vhat_vals[percent_size] = pyo.value(mdl.Vhat)
...         area_vals[percent_size] = pyo.value(mdl.A)
...
Solving RO problem for uncertainty set size 0:
...
Solving RO problem for uncertainty set size 2.5:
...
Solving RO problem for uncertainty set size 5:
...
Solving RO problem for uncertainty set size 7.5:
...
Solving RO problem for uncertainty set size 10:
...
All done. Exiting PyROS.
==============================================================================
```

Visualize the Results

We can visualize the results of our price-of-robustness analysis, as follows:

```
>>> fig, (obj_ax, vhat_ax, area_ax) = plt.subplots(
...     ncols=3,
...     figsize=(19, 4),
...     layout="constrained",
... )
>>> plt.subplots_adjust(wspace=0.4, hspace=0.6)
>>>
>>> # plot costs
>>> obj_ax.plot(obj_vals.keys(), obj_vals.values(), label="total", marker="o")
>>> obj_ax.plot(capex_vals.keys(), capex_vals.values(), label="CAPEX", marker="s")
>>> obj_ax.plot(opex_vals.keys(), opex_vals.values(), label="OPEX", marker="^")
>>> obj_ax.set_xlabel("Deviation from Nominal Value (%)")
>>> obj_ax.set_ylabel("Cost ($/yr)")
```

(continues on next page)

(continued from previous page)

```
>>> obj_ax.legend()
>>>
>>> # plot reactor volume
>>> vhat_ax.plot(vhat_vals.keys(), vhat_vals.values(), marker="o")
>>> vhat_ax.set_xlabel("Deviation from Nominal Value (%)")
>>> vhat_ax.set_ylabel(r"Reactor Volume ($\mathrm{m}^3$)")
>>>
>>> # plot cooler area
>>> area_ax.plot(area_vals.keys(), area_vals.values(), marker="o")
>>> area_ax.set_xlabel("Deviation from Nominal Value (%)")
>>> area_ax.set_ylabel(r"Cooler Heat Transfer Area ($\mathrm{m}^2$)")
>>> area_ax.set_ylim([7.45, 7.46])
>>>
>>> plt.show()
```

Notice that the costs and reactor volume increase with the size of the uncertainty set, whereas the heat transfer area of the cooler does not vary.

Assess Robust Feasibility of the Solutions

We can also visualize the robustness of each solution:

```
>>> plot_feasibility(
...     {key: (vhat_vals[key], area_vals[key]) for key in vhat_vals},
...     solver=ipopt,
...     test_set_size=10,
... )
```

Notice that:

- Every solution is found to be robust feasible subject to its corresponding uncertainty set, but robust infeasible

subject to strict supersets.

- As the size of uncertainty set is increased, so is the average objective value.

Citing PyROS

If you use PyROS in your research, please acknowledge PyROS by citing [IAE+21].

Feedback and Reporting Issues

Please provide feedback and/or report any problems by opening an issue on the Pyomo GitHub page.

3.3.4 MindtPy Solver

The Mixed-Integer Nonlinear Decomposition Toolbox in Pyomo (MindtPy) solver allows users to solve Mixed-Integer Nonlinear Programs (MINLP) using decomposition algorithms. These decomposition algorithms usually rely on the solution of Mixed-Integer Linear Programs (MILP) and Nonlinear Programs (NLP).

The following algorithms are currently available in MindtPy:

- **Outer-Approximation (OA)** [Duran & Grossmann, 1986]
- **LP/NLP based Branch-and-Bound (LP/NLP BB)** [Quesada & Grossmann, 1992]
- **Extended Cutting Plane (ECP)** [Westerlund & Petterson, 1995]
- **Global Outer-Approximation (GOA)** [Kesavan & Allgor, 2004]
- **Regularized Outer-Approximation (ROA)** [Bernal & Peng, 2021, Kronqvist & Bernal, 2018]
- **Feasibility Pump (FP)** [Bernal & Vigerske, 2019, Bonami & Cornuéjols, 2009]

Usage and early implementation details for MindtPy can be found in the PSE 2018 paper Bernal et al., (ref, preprint). This solver implementation has been developed by David Bernal and Zedong Peng as part of research efforts at the Bernal Research Group and the Grossmann Research Group at Purdue University and Carnegie Mellon University.

MINLP Formulation

The general formulation of the mixed integer nonlinear programming (MINLP) models is as follows.

$$\begin{aligned}
&\min_{\mathbf{x},\mathbf{y}} && f(\mathbf{x},\mathbf{y}) \\
&\text{s.t.} && g_j(\mathbf{x},\mathbf{y}) \leq 0 \quad \forall j = 1, \ldots l, \\
& && \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} \leq \mathbf{b}, \\
& && \mathbf{x} \in \mathbb{R}^n, \ \mathbf{y} \in \mathbb{Z}^m.
\end{aligned} \tag{MINLP}$$

where

- $\mathbf{x} \in \mathbb{R}^n$ are continuous variables,
- $\mathbf{y} \in \mathbb{Z}^m$ are discrete variables,
- $f, g_1, \ldots, g_l$ are non-linear smooth functions,
- $\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} \leq \mathbf{b}$‘ are linear constraints.

Solve Convex MINLPs

Usage of MindtPy to solve a convex MINLP Pyomo model involves:

```
>>> pyo.SolverFactory('mindtpy').solve(model)
```

An example which includes the modeling approach may be found below.

```
Required imports
>>> import pyomo.environ as pyo

Create a simple model
>>> model = pyo.ConcreteModel()

>>> model.x = pyo.Var(bounds=(1.0,10.0),initialize=5.0)
>>> model.y = pyo.Var(within=pyo.Binary)

>>> model.c1 = pyo.Constraint(expr=(model.x-4.0)**2 - model.x <= 50.0*(1-model.y))
>>> model.c2 = pyo.Constraint(expr=model.x*pyo.log(model.x)+5.0 <= 50.0*(model.y))

>>> model.objective = pyo.Objective(expr=model.x, sense=pyo.minimize)

Solve the model using MindtPy
>>> pyo.SolverFactory('mindtpy').solve(model, mip_solver='glpk', nlp_solver='ipopt')
```

The solution may then be displayed by using the commands

```
>>> model.objective.display()
>>> model.display()
>>> model.pprint()
```

Note

When troubleshooting, it can often be helpful to turn on verbose output using the `tee` flag.

```
>>> pyo.SolverFactory('mindtpy').solve(model, mip_solver='glpk', nlp_solver='ipopt',␣
↪tee=True)
```

MindtPy also supports setting options for mip solvers and nlp solvers.

```
>>> pyo.SolverFactory('mindtpy').solve(model,
                                    strategy='OA',
                                    time_limit=3600,
                                    mip_solver='gams',
                                    mip_solver_args=dict(solver='cplex', warmstart=True),
                                    nlp_solver='ipopt',
                                    tee=True)
```

There are three initialization strategies in MindtPy: `rNLP`, `initial_binary`, `max_binary`. In OA and GOA strategies, the default initialization strategy is `rNLP`. In ECP strategy, the default initialization strategy is `max_binary`.

LP/NLP Based Branch-and-Bound

MindtPy also supports single-tree implementation of Outer-Approximation (OA) algorithm, which is known as LP/NLP based branch-and-bound algorithm originally described in [Quesada & Grossmann, 1992]. The LP/NLP based branch-and-bound algorithm in MindtPy is implemented based on the LazyConstraintCallback function in commercial solvers.

Note

In Pyomo, *persistent solvers* are necessary to set or register callback functions. The single tree implementation currently only works with CPLEX and GUROBI, more exactly `cplex_persistent` and `gurobi_persistent`. To use the LazyConstraintCallback function of CPLEX from Pyomo, the CPLEX Python API is required. This means both IBM ILOG CPLEX Optimization Studio and the CPLEX-Python modules should be installed on your computer. To use the cbLazy function of GUROBI from pyomo, gurobipy is required.

A usage example for LP/NLP based branch-and-bound algorithm is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='OA',
...                                    mip_solver='cplex_persistent',  # or 'gurobi_
↪persistent'
...                                    nlp_solver='ipopt',
...                                    single_tree=True)
>>> model.objective.display()
```

Regularized Outer-Approximation

As a new implementation in MindtPy, we provide a flexible regularization technique implementation. In this technique, an extra mixed-integer problem is solved in each decomposition iteration or incumbent solution of the single-tree solution methods. The extra mixed-integer program is constructed to provide a point where the NLP problem is solved closer to the feasible region described by the non-linear constraint. This approach has been proposed in [Kronqvist et al., 2020], and it has shown to be efficient for highly non-linear convex MINLP problems. In [Kronqvist et al., 2020], two different regularization approaches are proposed, using a squared Euclidean norm which was proved to make the procedure equivalent to adding a trust-region constraint to Outer-approximation, and a second-order approximation of the Lagrangian of the problem, which showed better performance. We implement these methods, using PyomoNLP as the interface to compute the second-order approximation of the Lagrangian, and extend them to consider linear norm objectives and first-order approximations of the Lagrangian. Finally, we implemented an approximated second-order expansion of the Lagrangian, drawing inspiration from the Sequential Quadratic Programming (SQP) literature. The details of this implementation are included in [Bernal et al., 2021].

A usage example for regularized OA is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='OA',
...                                    mip_solver='cplex',
...                                    nlp_solver='ipopt',
...                                    add_regularization='level_L1'
...                                    # alternative regularizations
...                                    # 'level_L1', 'level_L2', 'level_L_infinity',
...                                    # 'grad_lag', 'hess_lag', 'hess_only_lag', 'sqp_lag'
...                                    )
>>> model.objective.display()
```

Solution Pool Implementation

MindtPy supports solution pool of the MILP solver, CPLEX and GUROBI. With the help of the solution, MindtPy can explore several integer combinations in one iteration.

A usage example for OA with solution pool is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='OA',
...                                    mip_solver='cplex_persistent',
...                                    nlp_solver='ipopt',
...                                    solution_pool=True,
...                                    num_solution_iteration=10, # default=5
...                                    tee=True
...                                    )
>>> model.objective.display()
```

Feasibility Pump

For some MINLP problems, the Outer Approximation method might have difficulty in finding a feasible solution. MindtPy provides the Feasibility Pump implementation to find feasible solutions for convex MINLPs quickly. The main idea of the Feasibility Pump is to decompose the original mixed-integer problem into two parts: integer feasibility and constraint feasibility. For convex MINLPs, a MIP is solved to obtain a solution, which satisfies the integrality constraints on y, but may violate some of the nonlinear constraints; next, by solving an NLP, a solution is computed that satisfies the nonlinear constraints but might again violate the integrality constraints on y. By minimizing the distance between these two types of solutions iteratively, a constraint and integer feasible solution can be expected. In MindtPy, the Feasibility Pump can be used both as an initialization strategy and a decomposition strategy. For details of this implementation are included in [Bernal et al., 2017].

A usage example for Feasibility Pump as the initialization strategy is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='OA',
...                                    init_strategy='FP',
...                                    mip_solver='cplex',
...                                    nlp_solver='ipopt',
...                                    tee=True
...                                    )
>>> model.objective.display()
```

A usage example for Feasibility Pump as the decomposition strategy is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='FP',
...                                    mip_solver='cplex',
...                                    nlp_solver='ipopt',
...                                    tee=True
...                                    )
>>> model.objective.display()
```

Solve Nonconvex MINLPs

Equality Relaxation

Under certain assumptions concerning the convexity of the nonlinear functions, an equality constraint can be relaxed to be an inequality constraint. This property can be used in the MIP master problem to accumulate linear approximations(OA cuts). The sense of the equivalent inequality constraint is based on the sign of the dual values of the equality constraint. Therefore, the sense of the OA cuts for equality constraint should be determined according to both the objective sense and the sign of the dual values. In MindtPy, the dual value of the equality constraint is calculated as follows.

constraint	status at x_1	dual values
$g(x) \le b$	$g(x_1) \le b$	0
$g(x) \le b$	$g(x_1) > b$	$g(x1) - b$
$g(x) \ge b$	$g(x_1) \ge b$	0
$g(x) \ge b$	$g(x_1) < b$	$b - g(x1)$

Augmented Penalty

Augmented Penalty refers to the introduction of (non-negative) slack variables on the right hand sides of the just described inequality constraints and the modification of the objective function when assumptions concerning convexity do not hold. (From DICOPT)

Global Outer-Approximation

Apart from the decomposition methods for convex MINLP problems [Kronqvist et al., 2019], MindtPy provides an implementation of Global Outer Approximation (GOA) as described in [Kesavan & Allgor, 2004], to provide optimality guaranteed for nonconvex MINLP problems. Here, the validity of the Mixed-integer Linear Programming relaxation of the original problem is guaranteed via the usage of Generalized McCormick envelopes, computed using the *interface to the MC++ package*. The NLP subproblems, in this case, need to be solved to global optimality, which can be achieved through global NLP solvers such as BARON or SCIP.

Convergence

MindtPy provides two ways to guarantee the finite convergence of the algorithm.

- **No-good cuts**. No-good cuts(integer cuts) are added to the MILP master problem in each iteration.
- **Tabu list**. Tabu list is only supported if the `mip_solver` is `cplex_persistent` (`gurobi_persistent` pending). In each iteration, the explored integer combinations will be added to the *tabu_list*. When solving the next MILP problem, the MIP solver will reject the previously explored solutions in the branch and bound process through IncumbentCallback.

Bound Calculation

Since no-good cuts or tabu list is applied in the Global Outer-Approximation (GOA) method, the MILP master problem cannot provide a valid bound for the original problem. After the GOA method has converged, MindtPy will remove the no-good cuts or the tabu integer combinations added when and after the optimal solution has been found. Solving this problem will give us a valid bound for the original problem.

The GOA method also has a single-tree implementation with `cplex_persistent` and `gurobi_persistent`. Notice that this method is more computationally expensive than the other strategies implemented for convex MINLP like OA and ECP, which can be used as heuristics for nonconvex MINLP problems.

A usage example for GOA is as follows:

```
>>> pyo.SolverFactory('mindtpy').solve(model,
...                                    strategy='GOA',
...                                    mip_solver='cplex',
...                                    nlp_solver='baron')
>>> model.objective.display()
```

MindtPy Implementation and Optional Arguments

Warning

MindtPy optional arguments should be considered beta code and are subject to change.

`class pyomo.contrib.mindtpy.MindtPy.MindtPySolver`

Decomposition solver for Mixed-Integer Nonlinear Programming (MINLP) problems.

The MindtPy (Mixed-Integer Nonlinear Decomposition Toolbox in Pyomo) solver applies a variety of decomposition-based approaches to solve Mixed-Integer Nonlinear Programming (MINLP) problems. These approaches include:

- Outer approximation (OA)
- Global outer approximation (GOA)
- Regularized outer approximation (ROA)
- LP/NLP based branch-and-bound (LP/NLP)
- Global LP/NLP based branch-and-bound (GLP/NLP)
- Regularized LP/NLP based branch-and-bound (RLP/NLP)
- Feasibility pump (FP)

available(*exception_flag=True*)

Check if solver is available.

solve(*model*, ***kwds*)

Solve the model.

Parameters

model (Block) – a Pyomo model or block to be solved

Keyword Arguments

- **iteration_limit** (*NonNegativeInt, default=50*) – Number of maximum iterations in the decomposition methods.
- **stalling_limit** (*PositiveInt, default=15*) – Stalling limit for primal bound progress in the decomposition methods.
- **time_limit** (*PositiveInt, default=600*) – Seconds allowed until terminated. Note that the time limit cancurrently only be enforced between subsolver invocations. You mayneed to set subsolver time limits as well.
- **strategy** (In*['OA', 'ECP', 'GOA', 'FP'], default='OA'*) – MINLP Decomposition strategy to be applied to the method. Currently available Outer Approximation (OA), Extended Cutting Plane (ECP), Global Outer Approximation (GOA) and Feasibility Pump (FP).
- **add_regularization** (In*['level_L1', 'level_L2', 'level_L_infinity', 'grad_lag', 'hess_lag', 'hess_only_lag', 'sqp_lag'], optional*) – Solving a regularization problem before solve the fixed subproblemthe objective function of the regularization problem.
- **call_after_main_solve** (*default=<pyomo.contrib.gdpopt.util._DoNothing object at 0x75d7ff2caa50>*) – Callback hook after a solution of the main problem.

- **call_before_subproblem_solve** (*default=<pyomo.contrib.gdpopt.util._DoNothing object at 0x75d7ff0147d0>*) – Callback hook before a solution of the nonlinear subproblem.
- **call_after_subproblem_solve** (*default=<pyomo.contrib.gdpopt.util._DoNothing object at 0x75d7ff014910>*) – Callback hook after a solution of the nonlinear subproblem.
- **call_after_subproblem_feasible** (*default=<pyomo.contrib.gdpopt.util._DoNothing object at 0x75d7ff153a80>*) – Callback hook after a feasible solution of the nonlinear subproblem.
- **tee** (*bool, default=False*) – Stream output to terminal.
- **logger** (*a_logger, default='pyomo.contrib.mindtpy'*) – The logger object or name to use for reporting.
- **logging_level** (*NonNegativeInt, default=20*) – The logging level for MindtPy.CRITICAL = 50, ERROR = 40, WARNING = 30, INFO = 20, DEBUG = 10, NOTSET = 0
- **integer_to_binary** (*bool, default=False*) – Convert integer variables to binaries (for no-good cuts).
- **add_no_good_cuts** (*bool, default=False*) – Add no-good cuts (no-good cuts) to binary variables to disallow same integer solution again.Note that integer_to_binary flag needs to be used to apply it to actual integers and not just binaries.
- **use_tabu_list** (*bool, default=False*) – Use tabu list and incumbent callback to disallow same integer solution again.
- **single_tree** (*bool, default=False*) – Use single tree implementation in solving the MIP main problem.
- **solution_pool** (*bool, default=False*) – Use solution pool in solving the MIP main problem.
- **num_solution_iteration** (*PositiveInt, default=5*) – The number of MIP solutions (from the solution pool) used to generate the fixed NLP subproblem in each iteration.
- **cycling_check** (*bool, default=True*) – Check if OA algorithm is stalled in a cycle and terminate.
- **feasibility_norm** (In*['L1', 'L2', 'L_infinity'], default='L_infinity'*) – Different forms of objective function in feasibility subproblem.
- **differentiate_mode** (In*['reverse_symbolic', 'sympy'], default='reverse_symbolic'*) – Differentiate mode to calculate jacobian.
- **use_mcpp** (*bool, default=False*) – Use package MC++ to set a bound for variable 'objective_value', which is introduced when the original problem's objective function is nonlinear.
- **calculate_dual_at_solution** (*bool, default=False*) – Calculate duals of the NLP subproblem.
- **use_fbbt** (*bool, default=False*) – Use fbbt to tighten the feasible region of the problem.
- **use_dual_bound** (*bool, default=True*) – Add dual bound constraint to enforce the objective satisfies best- found dual bound.

- **partition_obj_nonlinear_terms** (*bool*, *default=True*) – Partition objective with the sum of nonlinear terms using epigraph reformulation.
- **quadratic_strategy** (In*[0, 1, 2]*, *default=0*) – How to treat the quadratic terms in MINLP.0 : treat as nonlinear terms1 : only use quadratic terms in objective function directly in main problem2 : use quadratic terms in objective function and constraints in main problem
- **move_objective** (*bool*, *default=False*) – Whether to replace the objective function to constraint using epigraph constraint.
- **add_cuts_at_incumbent** (*bool*, *default=False*) – Whether to add lazy cuts to the main problem at the incumbent solution found in the branch & bound tree
- **nlp_solver** (In*['ipopt', 'appsi_ipopt', 'gams', 'baron', 'cyipopt']*, *default='ipopt'*) – Which NLP subsolver is going to be used for solving the nonlinearsubproblems.
- **nlp_solver_args** (*dict*, *optional*) – Which NLP subsolver options to be passed to the solver while solving the nonlinear subproblems.
- **mip_solver** (In*['gurobi', 'cplex', 'cbc', 'glpk', 'gams', 'gurobi_persistent', 'cplex_persistent', 'appsi_cplex', 'appsi_gurobi', 'appsi_highs']*, *default='glpk'*) – Which MIP subsolver is going to be used for solving the mixed-integer main problems.
- **mip_solver_args** (*dict*, *optional*) – Which MIP subsolver options to be passed to the solver while solving the mixed-integer main problems.
- **mip_solver_mipgap** (*PositiveFloat*, *default=0.0001*) – Mipgap passed to MIP solver.
- **threads** (*NonNegativeInt*, *default=0*) – Threads used by MIP solver and NLP solver.
- **regularization_mip_threads** (*NonNegativeInt*, *default=0*) – Threads used by MIP solver to solve regularization main problem.
- **solver_tee** (*bool*, *default=False*) – Stream the output of MIP solver and NLP solver to terminal.
- **mip_solver_tee** (*bool*, *default=False*) – Stream the output of MIP solver to terminal.
- **nlp_solver_tee** (*bool*, *default=False*) – Stream the output of nlp solver to terminal.
- **mip_regularization_solver** (In*['gurobi', 'cplex', 'cbc', 'glpk', 'gams', 'gurobi_persistent', 'cplex_persistent', 'appsi_cplex', 'appsi_gurobi', 'appsi_highs']*, *optional*) – Which MIP subsolver is going to be used for solving the regularization problem.
- **absolute_bound_tolerance** (*PositiveFloat*, *default=0.0001*) – Absolute tolerance for bound feasibility checks.
- **relative_bound_tolerance** (*PositiveFloat*, *default=0.001*) – Relative tolerance for bound feasibility checks. $|PrimalBound - DualBound|/(1e-10+|PrimalBound|) <= relativetolerance$
- **small_dual_tolerance** (*default=1e-08*) – When generating cuts, small duals multiplied by expressions can cause problems. Exclude all duals smaller in absolute value than the following.
- **integer_tolerance** (*default=1e-05*) – Tolerance on integral values.

- **constraint_tolerance** (*default=1e-06*) – Tolerance on constraint satisfaction.
- **variable_tolerance** (*default=1e-08*) – Tolerance on variable bounds.
- **zero_tolerance** (*default=1e-08*) – Tolerance on variable equal to zero.
- **fp_cutoffdecr** (*PositiveFloat, default=0.1*) – Additional relative decrement of cutoff value for the original objective function.
- **fp_iteration_limit** (*PositiveInt, default=20*) – Number of maximum iterations in the feasibility pump methods.
- **fp_projcuts** (*bool, default=True*) – Whether to add cut derived from regularization of MIP solution onto NLP feasible set.
- **fp_transfercuts** (*bool, default=True*) – Whether to transfer cuts from the Feasibility Pump MIP to main MIP in selected strategy (all except from the round in which the FP MIP became infeasible).
- **fp_projzerotol** (*PositiveFloat, default=0.0001*) – Tolerance on when to consider optimal value of regularization problem as zero, which may trigger the solution of a Sub-NLP.
- **fp_mipgap** (*PositiveFloat, default=0.01*) – Optimality tolerance (relative gap) to use for solving MIP regularization problem.
- **fp_discrete_only** (*bool, default=True*) – Only calculate the distance among discrete variables in regularization problems.
- **fp_main_norm** (In*['L1', 'L2', 'L_infinity'], default='L1'*) – Different forms of objective function MIP regularization problem.
- **fp_norm_constraint** (*bool, default=True*) – Whether to add the norm constraint to FP-NLP
- **fp_norm_constraint_coef** (*PositiveFloat, default=1*) – The coefficient in the norm constraint, correspond to the Beta in the paper.
- **obj_bound** (*PositiveFloat, default=1000000000000000.0*) – Bound applied to the linearization of the objective function if main MIP is unbounded.
- **continuous_var_bound** (*PositiveFloat, default=10000000000.0*) – Default bound added to unbounded continuous variables in nonlinear constraint if single tree is activated.
- **integer_var_bound** (*PositiveFloat, default=1000000000.0*) – Default bound added to unbounded integral variables in nonlinear constraint if single tree is activated.
- **initial_bound_coef** (*PositiveFloat, default=0.1*) – The coefficient used to approximate the initial primal/dual bound.
- **level_coef** (*PositiveFloat, default=0.5*) – The coefficient in the regularization main problemrepresents how much the linear approximation of the MINLP problem is trusted.
- **solution_limit** (*PositiveInt, default=10*) – The solution limit for the regularization problem since it does not need to be solved to optimality.
- **sqp_lag_scaling_coef** (In*['fixed', 'variable_dependent'], default='fixed'*) – The coefficient used to scale the L2 norm in sqp_lag.

version()

Return a 3-tuple describing the solver version.

Get Help

Ways to get help: https://github.com/Pyomo/pyomo#getting-help

Report a Bug

If you find a bug in MindtPy, we will be grateful if you could

- submit an issue in Pyomo repository
- directly contact David Bernal <dbernaln@purdue.edu> and Zedong Peng <zdpeng95@gmail.com>.

3.3.5 MC++ Interface

The Pyomo-MC++ interface allows for bounding of factorable functions using the MC++ library developed by the OMEGA research group at Imperial College London. Documentation for MC++ may be found on the MC++ website.

Default Installation

Pyomo now supports automated downloading and compilation of MC++. To install MC++ and other third party compiled extensions, run:

```
pyomo download-extensions
pyomo build-extensions
```

To get and install just MC++, run the following commands in the `pyomo/contrib/mcpp` directory:

```
python getMCPP.py
python build.py
```

This should install MC++ to the pyomo plugins directory, by default located at `$HOME/.pyomo/`.

Manual Installation

Support for MC++ has only been validated by Pyomo developers using Linux and OSX. Installation instructions for the MC++ library may be found on the MC++ website.

We assume that you have installed MC++ into a directory of your choice. We will denote this directory by `$MCPP_PATH`. For example, you should see that the file `$MCPP_PATH/INSTALL` exists.

Navigate to the `pyomo/contrib/mcpp` directory in your pyomo installation. This directory should contain a file named `mcppInterface.cpp`. You will need to compile this file using the following command:

```
g++ -I $MCPP_PATH/src/3rdparty/fadbad++ -I $MCPP_PATH/src/mc -I /usr/include/python3.7 -
↪fPIC -O2 -c mcppInterface.cpp
```

This links the MC++ required library FADBAD++, MC++ itself, and Python to compile the Pyomo-MC++ interface. If successful, you will now have a file named `mcppInterface.o` in your working directory. If you are not using Python 3.7, you will need to link to the appropriate Python version. You now need to create a shared object file with the following command:

```
g++ -shared mcppInterface.o -o mcppInterface.so
```

You may then test your installation by running the test file:

```
python test_mcpp.py
```

3.3.6 Multistart Solver

The multistart solver is used in cases where the objective function is known to be non-convex but the global optimum is still desired. It works by running a non-linear solver of your choice multiple times at different starting points, and returns the best of the solutions.

Using Multistart Solver

To use the multistart solver, define your Pyomo model as usual:

```
Required import
>>> import pyomo.environ as pyo

Create a simple model
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)

Invoke the multistart solver
>>> pyo.SolverFactory('multistart').solve(m)
```

Multistart wrapper implementation and optional arguments

class pyomo.contrib.multistart.multi.**MultiStart**

Solver wrapper that initializes at multiple starting points.

TODO: also return appropriate duals

For theoretical underpinning, see https://www.semanticscholar.org/paper/How-many-random-restarts-are-enough-Dick-Wong/55b248b398a03dc1ac9a65437f88b835554329e0

Keyword arguments below are specified for the `solve` function.

Keyword Arguments

- **strategy** (In(*dict_keys(['rand', 'midpoint_guess_and_bound', 'rand_guess_and_bound', 'rand_distributed', 'midpoint'])), default='rand'*) – Specify the restart strategy.
  - ”rand”: random choice between variable bounds
  - ”midpoint_guess_and_bound”: midpoint between current value and farthest bound
  - ”rand_guess_and_bound”: random choice between current value and farthest bound
  - ”rand_distributed”: random choice among evenly distributed values
  - ”midpoint”: exact midpoint between the bounds. If using this option, multiple iterations are useless.
- **solver** (*default='ipopt'*) – solver to use, defaults to ipopt
- **solver_args** (*default={}*) – Dictionary of keyword arguments to pass to the solver.
- **iterations** (*default=10*) – Specify the number of iterations, defaults to 10. If -1 is specified, the high confidence stopping rule will be used
- **stopping_mass** (*default=0.5*) – Maximum allowable estimated missing mass of optima for the high confidence stopping rule, only used with the random strategy. The lower the parameter, the stricter the rule. Value bounded in (0, 1].

- **stopping_delta** (*default=0.5*) – 1 minus the confidence level required for the stopping rule for the high confidence stopping rule, only used with the random strategy. The lower the parameter, the stricter the rule. Value bounded in (0, 1].
- **suppress_unbounded_warning** (*bool, default=False*) – True to suppress warning for skipping unbounded variables.
- **HCS_max_iterations** (*default=1000*) – Maximum number of iterations before interrupting the high confidence stopping rule.
- **HCS_tolerance** (*default=0*) – Tolerance on HCS objective value equality. Defaults to Python float equality precision.

available(*exception_flag=True*)

Check if solver is available.

TODO: For now, it is always available. However, sub-solvers may not always be available, and so this should reflect that possibility.

3.3.7 Trust Region Framework Method Solver

The Trust Region Framework (TRF) method solver allows users to solve hybrid glass box/black box optimization problems in which parts of the system are modeled with open, equation-based models and parts of the system are black boxes. This method utilizes surrogate models that substitute high-fidelity models with low-fidelity basis functions, thus avoiding the direct implementation of the large, computationally expensive high-fidelity models. This is done iteratively, resulting in fewer calls to the computationally expensive functions.

This module implements the method from Yoshio & Biegler [Yoshio & Biegler, 2021] and represents a rewrite of the original 2018 implementation of the algorithm from Eason & Biegler [Eason & Biegler, 2018].

In the context of this updated module, black box functions are implemented as Pyomo External Functions.

This work was conducted as part of the Institute for the Design of Advanced Energy Systems (IDAES) with support through the Simulation-Based Engineering, Crosscutting Research Program within the U.S. Department of Energy's Office of Fossil Energy and Carbon Management.

Methodology Overview

The formulation of the original hybrid problem is:

$$\begin{array}{ll} \min & f\left(z,w,d\left(w\right)\right) \\ \text{s.t.} & h\left(z,w,d\left(w\right)\right)=0 \\ & g\left(z,w,d\left(w\right)\right)\leq 0 \end{array}$$

where:

- $w \in \mathbb{R}^m$ are the inputs to the external functions
- $z \in \mathbb{R}^n$ are the remaining decision variables (i.e., degrees of freedom)
- $d(w) : \mathbb{R}^m \to \mathbb{R}^p$ are the outputs of the external functions as a function of w
- f, h, g, d are all assumed to be twice continuously differentiable

This formulation is reworked to separate all external function information as follows to enable the usage of the trust region method:

$$\begin{array}{ll} \min_x & f\left(x\right) \\ \text{s.t.} & h\left(x\right)=0 \\ & g\left(x\right)\leq 0 \\ & y=d\left(w\right) \end{array}$$

where:

- $y \in \mathbb{R}^p$ are the outputs of the external functions
- $x^T = [w^T, y^T, z^T]$ is a set of all inputs and outputs

Using this formulation and a user-supplied low-fidelity/ideal model basis function $b(w)$, the algorithm iteratively solves subproblems using the surrogate model:

$$r_k(w) = b(w) + (d(w_k) - b(w_k)) + (\nabla d(w_k) - \nabla b(w_k))^T (w - w_k)$$

This acts similarly to Newton's method in that small, incremental steps are taken towards an optimal solution. At each iteration, the current solution of the subproblem is compared to the previous solution to ensure that the iteration has moved in a direction towards an optimal solution. If not true, the step is rejected. If true, the step is accepted and the surrogate model is updated for the next iteration.

Globalization Strategies

The TRF solver supports two globalization strategies to control step acceptance and trust region updates: the **filter** method (default) and the **funnel** method. The strategy is selected via the `globalization_strategy` keyword argument (`'filter'` for filter, `'funnel'` for funnel).

Filter Method (default)

The filter method, used in the original Yoshio & Biegler (2021) implementation, maintains a filter set of (feasibility, objective) pairs. A new iterate is accepted if it is not dominated by any entry in the filter. At each iteration, steps are classified as either f-type (objective-improving) or theta-type (feasibility-improving), and the trust region radius is updated accordingly.

Funnel Method

The funnel globalization strategy is an alternative to the filter method, introduced by [Hameed et al., 2026]. Instead of a discrete filter set, the funnel maintains a dynamic upper bound on the feasibility measure that shrinks as the algorithm converges. For full details of the funnel algorithm, step classification, and acceptance conditions, please refer to [Hameed et al., 2026].

When using TRF, please consider citing the above papers.

TRF Inputs

The required inputs to the TRF *solve* method are the following:

- The optimization model
- List of degree of freedom variables within the model

The optional input to the TRF *solve* method is the following:

- The external function surrogate model rule ("basis function")

TRF Solver Interface

Note

The keyword arguments can be updated at solver instantiation or later when the `solve` method is called.

class `pyomo.contrib.trustregion.TRF.TrustRegionSolver(**kwds)`

The Trust Region Solver is a 'solver' based on the 2016/2018/2020 AiChE papers by Eason (2016/2018), Yoshio (2020), and Biegler.

solve(*model*, *degrees_of_freedom_variables*, *ext_fcn_surrogate_map_rule=None*, ***kwds*)

This method calls the TRF algorithm.

Parameters

- **model** (ConcreteModel) – The model to be solved using the Trust Region Framework.
- **degrees_of_freedom_variables** (*List[*Var*]*) – User-supplied input. The user must provide a list of vars which are the degrees of freedom or decision variables within the model.
- **ext_fcn_surrogate_map_rule** (Function, *optional*) – In the 2020 Yoshio/Biegler paper, this is referred to as the basis function *b(w)*. This is the low-fidelity model with which to solve the original process model problem and which is integrated into the surrogate model. The default is 0 (i.e., no basis function rule.)

Keyword Arguments

- **solver** (*default='ipopt'*) – Solver to use. Default = ipopt.
- **keepfiles** (*Bool, default=False*) – Optional. Whether or not to write files of sub-problems for use in debugging. Default = False.
- **tee** (*Bool, default=False*) – Optional. Sets the tee for sub-solver(s) utilized. Default = False.
- **verbose** (*Bool, default=False*) – Optional. When True, print each iteration's relevant information to the console as well as to the log. Default = False.
- **trust_radius** (*PositiveFloat, default=1.0*) – Initial trust region radius delta_0. Default = 1.0.
- **minimum_radius** (*PositiveFloat, default=1e-06*) – Minimum allowed trust region radius delta_min. Default = 1e-6.
- **maximum_radius** (*PositiveFloat, default=100.0*) – Maximum allowed trust region radius. If trust region radius reaches maximum allowed, solver will exit. Default = 100 * trust_radius.
- **maximum_iterations** (*PositiveInt, default=50*) – Maximum allowed number of iterations. Default = 50.
- **feasibility_termination** (*PositiveFloat, default=1e-05*) – Feasibility measure termination tolerance epsilon_theta. Default = 1e-5.
- **step_size_termination** (*PositiveFloat, default=1e-05*) – Step size termination tolerance epsilon_s. Matches the feasibility termination tolerance by default.
- **minimum_feasibility** (*PositiveFloat, default=0.0001*) – Minimum feasibility measure theta_min. Default = 1e-4.
- **switch_condition_kappa_theta** (In*(0..1), default=0.1*) – Switching condition parameter kappa_theta. Contained in open set (0, 1). Default = 0.1.
- **switch_condition_gamma_s** (*PositiveFloat, default=2.0*) – Switching condition parameter gamma_s. Must satisfy: gamma_s > 1/(1+mu) where mu is contained in set (0, 1]. Default = 2.0.
- **radius_update_param_gamma_c** (In*(0..1), default=0.5*) – Lower trust region update parameter gamma_c. Default = 0.5.
- **radius_update_param_gamma_e** (In*[1..inf], default=2.5*) – Upper trust region update parameter gamma_e. Default = 2.5.

- **ratio_test_param_eta_1** (In*(0..1)*, *default=0.05*) – Lower ratio test parameter `eta_1`. Must satisfy: `0 < eta_1 <= eta_2 < 1`. Default = 0.05.
- **ratio_test_param_eta_2** (In*(0..1)*, *default=0.2*) – Lower ratio test parameter `eta_2`. Must satisfy: `0 < eta_1 <= eta_2 < 1`. Default = 0.2.
- **globalization_strategy** (In*['filter', 'funnel']*, *default='filter'*) – Globalization strategy selection. `'filter'` = Filter method (default), `'funnel'` = Funnel method. Default = `'filter'`.
- **maximum_feasibility** (*PositiveFloat*, *default=50.0*) – Maximum allowable feasibility measure `theta_max`. Parameter for use in filter method.Default = 50.0.
- **param_filter_gamma_theta** (In*(0..1)*, *default=0.01*) – Fixed filter parameter `gamma_theta` within (0, 1). Default = 0.01
- **param_filter_gamma_f** (In*(0..1)*, *default=0.01*) – Fixed filter parameter `gamma_f` within (0, 1). Default = 0.01
- **funnel_param_phi_min** (*PositiveFloat*, *default=1e-08*) – Hard floor on funnel width `phi_min`. Must satisfy: `phi_min > 0`. Default = 1e-8.
- **funnel_param_kappa_f** (*PositiveFloat*, *default=0.25*) – Funnel shrink factor `kappa_f` applied after a theta-type step. Must satisfy: `0 < kappa_f < 1`. Default = 0.25.
- **funnel_param_kappa_r** (*PositiveFloat*, *default=1.05*) – Funnel relaxation factor `kappa_r` for theta-type step. Must satisfy: `kappa_r > 1`. Default = 1.05.
- **funnel_param_eta** (*PositiveFloat*, *default=0.0001*) – Armijo coefficient `eta` for f-type step sufficient decrease condition. Must satisfy: `0 < eta < 1`. Default = 1e-4.
- **funnel_param_alpha** (*PositiveFloat*, *default=0.5*) – Curvature exponent `alpha` in funnel boundary condition `phi^alpha`. Must satisfy: `0 < alpha < 1`. Default = 0.5.
- **funnel_param_beta** (*PositiveFloat*, *default=0.8*) – Theta-type shrink factor `beta`. Must satisfy: `0 < beta < 1`. Default = 0.8.
- **funnel_param_mu_s** (*PositiveFloat*, *default=0.01*) – Switching parameter `mu_s`. Must satisfy: `mu_s > 0` (small value, e.g. 1e-2). Default = 0.01.

TRF Usage Example

Two examples can be found in the examples subdirectory. One of them is implemented below.

Step 0: Import Pyomo

```
>>> # === Required imports ===
>>> import pyomo.environ as pyo
```

Step 1: Define the external function and its gradient

```
>>> # === Define a 'black box' function and its gradient ===
>>> def ext_fcn(a, b):
...     return pyo.sin(a - b)
>>> def grad_ext_fcn(args, fixed):
...     a, b = args[:2]
...     return [ pyo.cos(a - b), -pyo.cos(a - b) ]
```

Step 2: Create the model

```
>>> # === Construct the Pyomo model object ===
>>> def create_model():
...     m = pyo.ConcreteModel()
...     m.name = 'Example 1: Eason'
...     m.z = pyo.Var(range(3), domain=pyo.Reals, initialize=2.)
...     m.x = pyo.Var(range(2), initialize=2.)
...     m.x[1] = 1.0
...
...     m.ext_fcn = pyo.ExternalFunction(ext_fcn, grad_ext_fcn)
...
...     m.obj = pyo.Objective(
...         expr=(m.z[0]-1.0)**2 + (m.z[0]-m.z[1])**2 + (m.z[2]-1.0)**2 \
...            + (m.x[0]-1.0)**4 + (m.x[1]-1.0)**6
...     )
...
...     m.c1 = pyo.Constraint(
...         expr=m.x[0] * m.z[0]**2 + m.ext_fcn(m.x[0], m.x[1]) == 2*pyo.sqrt(2.0)
...         )
...     m.c2 = pyo.Constraint(expr=m.z[2]**4 * m.z[1]**2 + m.z[1] == 8+pyo.sqrt(2.0))
...     return m
>>> model = create_model()
```

Step 3: Solve with TRF (Filter, default)

Note

Reminder from earlier that the `solve` method requires the user pass the model and a list of variables which represent the degrees of freedom in the model. The user may also pass a low-fidelity/ideal model (or "basis function") to this method to improve convergence.

```
>>> # === Instantiate the TRF solver object ===
>>> trf_solver = pyo.SolverFactory('trustregion')
>>> # === Solve with TRF using the default filter globalization strategy ===
>>> result = trf_solver.solve(model, [model.z[0], model.z[1], model.z[2]])
EXIT: Optimal solution found.
...
```

Step 3 (alternative): Solve with TRF (Funnel)

To use the funnel globalization strategy instead of the filter, set `globalization_strategy='funnel'`. The funnel-specific parameters can also be customized as needed:

```
>>> # === Instantiate the TRF solver object with funnel strategy ===
>>> trf_solver = pyo.SolverFactory('trustregion', globalization_strategy='funnel')
>>> # === Solve with TRF using the funnel globalization strategy ===
>>> result = trf_solver.solve(model, [model.z[0], model.z[1], model.z[2]])
EXIT: Optimal solution found.
...
```

The funnel parameters can also be customized at solve time:

```
>>> result = trf_solver.solve(
...     model,
...     [model.z[0], model.z[1], model.z[2]],
...     globalization_strategy='funnel',
...     funnel_param_kappa_f=0.3,
...     funnel_param_eta=1e-4,
... )
EXIT: Optimal solution found.
...
```

The `solve` method returns a clone of the original model which has been run through TRF algorithm, thus leaving the original model intact.

Warning

TRF is still under a beta release. Please provide feedback and/or report any problems by opening an issue on the Pyomo GitHub page.

3.3.8 PyNumero

PyNumero is a package for developing parallel algorithms for nonlinear programs (NLPs). This documentation provides a brief introduction to PyNumero. For more details, see the `API documentation`).

PyNumero Installation

PyNumero is a module within Pyomo. Therefore, Pyomo must be installed to use PyNumero. PyNumero also has some extensions that need built. There are many ways to build the PyNumero extensions. Common use cases are listed below. However, more information can always be found at https://github.com/Pyomo/pyomo/blob/main/pyomo/contrib/pynumero/build.py and https://github.com/Pyomo/pyomo/blob/main/pyomo/contrib/pynumero/src/CMakeLists.txt.

Note that you will need a C++ compiler and CMake installed to build the PyNumero libraries.

Method 1

One way to build PyNumero extensions is with the pyomo *download-extensions* and *build-extensions* subcommands. Note that this approach will build PyNumero without support for the HSL linear solvers.

```
pyomo download-extensions
pyomo build-extensions
```

Method 2

If you want PyNumero support for the HSL solvers and you have an IPOPT compilation for your machine, you can build PyNumero using the build script

```
python -m pyomo.contrib.pynumero.build -DBUILD_ASL=ON -DBUILD_MA27=ON -DIPOPT_DIR=<path/
↪to/ipopt/build/>
```

Method 3

You can build the PyNumero libraries from source using *cmake*. This generally works best when building from a source distribution of Pyomo. Assuming that you are starting in the root of the Pyomo source distribution, you can follow the normal CMake build process

```
mkdir build
cd build
ccmake ../pyomo/contrib/pynumero/src
make
make install
```

10 Minutes to PyNumero

NLP Interfaces

Below are examples of using PyNumero's interfaces to ASL for function and derivative evaluation. More information can be found in the *API documentation*.

Relevant imports

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.pynumero.interfaces.pyomo_nlp import PyomoNLP
>>> import numpy as np
```

Create a Pyomo model

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(bounds=(-5, None))
>>> m.y = pyo.Var(initialize=2.5)
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c1 = pyo.Constraint(expr=m.y == (m.x - 1)**2)
>>> m.c2 = pyo.Constraint(expr=m.y >= pyo.exp(m.x))
```

Create a *pyomo.contrib.pynumero.interfaces.pyomo_nlp.PyomoNLP* instance

```
>>> nlp = PyomoNLP(m)
```

Get values of primals and duals

```
>>> nlp.get_primals()
array([0. , 2.5])
>>> nlp.get_duals()
array([0., 0.])
```

Get variable and constraint bounds

```
>>> nlp.primals_lb()
array([ -5., -inf])
>>> nlp.primals_ub()
array([inf, inf])
>>> nlp.constraints_lb()
array([  0., -inf])
>>> nlp.constraints_ub()
array([0., 0.])
```

Objective and constraint evaluations

```
>>> nlp.evaluate_objective()
6.25
>>> nlp.evaluate_constraints()
array([ 1.5, -1.5])
```

Derivative evaluations

```
>>> nlp.evaluate_grad_objective()
array([0., 5.])
>>> nlp.evaluate_jacobian()
<2x2 sparse matrix of type '<class 'numpy.float64'>'
    with 4 stored elements in COOrdinate format>
>>> nlp.evaluate_jacobian().toarray()
array([[ 2.,  1.],
       [ 1., -1.]])
>>> nlp.evaluate_hessian_lag().toarray()
array([[2., 0.],
       [0., 2.]])
```

Set values of primals and duals

```
>>> nlp.set_primals(np.array([0, 1]))
>>> nlp.evaluate_constraints()
array([0., 0.])
>>> nlp.set_duals(np.array([-2/3, 4/3]))
>>> nlp.evaluate_grad_objective() + nlp.evaluate_jacobian().transpose() * nlp.get_duals()
array([0., 0.])
```

Equality and inequality constraints separately

```
>>> nlp.evaluate_eq_constraints()
array([0.])
>>> nlp.evaluate_jacobian_eq().toarray()
array([[2., 1.]])
>>> nlp.evaluate_ineq_constraints()
array([0.])
>>> nlp.evaluate_jacobian_ineq().toarray()
array([[ 1., -1.]])
>>> nlp.get_duals_eq()
array([-0.66666667])
>>> nlp.get_duals_ineq()
array([1.33333333])
```

Linear Solver Interfaces

PyNumero's interfaces to linear solvers are very thin wrappers, and, hence, are rather low-level. It is relatively easy to wrap these again for specific applications. For example, see the linear solver interfaces in https://github.com/Pyomo/pyomo/tree/main/pyomo/contrib/interior_point/linalg, which wrap PyNumero's linear solver interfaces.

The motivation to keep PyNumero's interfaces as such thin wrappers is that different linear solvers serve different purposes. For example, HSL's MA27 can factorize symmetric indefinite matrices, while MUMPS can factorize unsymmetric, symmetric positive definite, or general symmetric matrices. PyNumero seeks to be independent of the application, giving more flexibility to algorithm developers.

Interface to MA27

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> from scipy.sparse import tril
>>> from pyomo.contrib.pynumero.linalg.ma27_interface import MA27
>>> row = np.array([0, 1, 0, 1, 0, 1, 2, 3, 3, 4, 4, 4])
>>> col = np.array([0, 1, 3, 3, 4, 4, 4, 0, 1, 0, 1, 2])
>>> data = np.array([1.67025575, 2, -1.64872127,  1, -1, -1, -1, -1.64872127, 1, -1, -1,␣
↪-1])
>>> A = coo_matrix((data, (row, col)), shape=(5,5))
>>> A.toarray()
array([[ 1.67025575,  0.        ,  0.        , -1.64872127, -1.        ],
       [ 0.        ,  2.        ,  0.        ,  1.        , -1.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , -1.        ],
       [-1.64872127,  1.        ,  0.        ,  0.        ,  0.        ],
       [-1.        , -1.        , -1.        ,  0.        ,  0.        ]])
>>> rhs = np.array([-0.67025575, -1.2,  0.1,  1.14872127,  1.25])
>>> solver = MA27()
>>> solver.set_cntl(1, 1e-6)  # set the pivot tolerance
>>> status = solver.do_symbolic_factorization(A)
>>> status = solver.do_numeric_factorization(A)
>>> x, status = solver.do_back_solve(rhs)
>>> np.max(np.abs(A*x - rhs)) <= 1e-15
np.True_
```

Interface to MUMPS

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> from scipy.sparse import tril
>>> from pyomo.contrib.pynumero.linalg.mumps_interface import␣
↪MumpsCentralizedAssembledLinearSolver
>>> row = np.array([0, 1, 0, 1, 0, 1, 2, 3, 3, 4, 4, 4])
>>> col = np.array([0, 1, 3, 3, 4, 4, 4, 0, 1, 0, 1, 2])
>>> data = np.array([1.67025575, 2, -1.64872127,  1, -1, -1, -1, -1.64872127, 1, -1, -1,␣
↪-1])
>>> A = coo_matrix((data, (row, col)), shape=(5,5))
>>> A.toarray()
array([[ 1.67025575,  0.        ,  0.        , -1.64872127, -1.        ],
       [ 0.        ,  2.        ,  0.        ,  1.        , -1.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , -1.        ],
       [-1.64872127,  1.        ,  0.        ,  0.        ,  0.        ],
       [-1.        , -1.        , -1.        ,  0.        ,  0.        ]])
>>> rhs = np.array([-0.67025575, -1.2,  0.1,  1.14872127,  1.25])
>>> solver = MumpsCentralizedAssembledLinearSolver(sym=2, par=1, comm=None)  # symmetric␣
↪matrix; solve in serial
>>> status = solver.do_symbolic_factorization(A)
>>> status = solver.do_numeric_factorization(A)
>>> x, status = solver.do_back_solve(rhs)
>>> np.max(np.abs(A*x - rhs)) <= 1e-15
np.True_
```

Of course, SciPy solvers can also be used. See SciPy documentation for details.

Block Vectors and Matrices

Block vectors and matrices (`BlockVector` and `BlockMatrix`) provide a mechanism to perform linear algebra operations with very structured matrices and vectors.

When a BlockVector or BlockMatrix is constructed, the number of blocks must be specified.

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> from pyomo.contrib.pynumero.sparse import BlockVector, BlockMatrix
>>> v = BlockVector(3)
>>> m = BlockMatrix(3, 3)
```

Setting blocks:

```
>>> v.set_block(0, np.array([-0.67025575, -1.2]))
>>> v.set_block(1, np.array([0.1, 1.14872127]))
>>> v.set_block(2, np.array([1.25]))
>>> v.flatten()
array([-0.67025575, -1.2       ,  0.1       ,  1.14872127,  1.25      ])
```

The *flatten* method converts the BlockVector into a NumPy array.

```
>>> m.set_block(0, 0, coo_matrix(np.array([[1.67025575, 0], [0, 2]])))
>>> m.set_block(0, 1, coo_matrix(np.array([[0, -1.64872127], [0, 1]])))
>>> m.set_block(0, 2, coo_matrix(np.array([[-1.0], [-1]])))
>>> m.set_block(1, 0, coo_matrix(np.array([[0, -1.64872127], [0, 1]])).transpose())
>>> m.set_block(1, 2, coo_matrix(np.array([[-1.0], [0]])))
>>> m.set_block(2, 0, coo_matrix(np.array([[-1.0], [-1]])).transpose())
>>> m.set_block(2, 1, coo_matrix(np.array([[-1.0], [0]])).transpose())
>>> m.tocoo().toarray()
array([[ 1.67025575,  0.        ,  0.        , -1.64872127, -1.        ],
       [ 0.        ,  2.        ,  0.        ,  1.        , -1.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , -1.        ],
       [-1.64872127,  1.        ,  0.        ,  0.        ,  0.        ],
       [-1.        , -1.        , -1.        ,  0.        ,  0.        ]])
```

The *tocoo* method converts the *BlockMatrix* to a SciPy sparse *coo_matrix*.

Once the dimensions of a block have been set, they cannot be changed:

```
>>> v.set_block(0, np.ones(3))
Traceback (most recent call last):
  ...
ValueError: Incompatible dimensions for block 0; got 3; expected 2
```

Properties:

```
>>> v.shape
(5,)
>>> v.size
5
>>> v.nblocks
```

(continues on next page)

(continued from previous page)

```
3
>>> v.bshape
(3,)
>>> m.shape
(5, 5)
>>> m.bshape
(3, 3)
>>> m.nnz
12
```

Much of the *BlockVector* API matches that of NumPy arrays:

```
>>> v.sum()
0.62846552
>>> v.max()
1.25
>>> np.abs(v).flatten()
array([0.67025575, 1.2       , 0.1       , 1.14872127, 1.25      ])
>>> (2*v).flatten()
array([-1.3405115 , -2.4       ,  0.2       ,  2.29744254,  2.5       ])
>>> (v + v).flatten()
array([-1.3405115 , -2.4       ,  0.2       ,  2.29744254,  2.5       ])
>>> v.dot(v)
4.781303326558476
```

Similarly, *BlockMatrix* behaves very similarly to SciPy sparse matrices:

```
>>> (2*m).tocoo().toarray()
array([[ 3.3405115 ,  0.        ,  0.        , -3.29744254, -2.        ],
       [ 0.        ,  4.        ,  0.        ,  2.        , -2.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , -2.        ],
       [-3.29744254,  2.        ,  0.        ,  0.        ,  0.        ],
       [-2.        , -2.        , -2.        ,  0.        ,  0.        ]])
>>> (m - m).tocoo().toarray()
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
>>> m * v
BlockVector(3,)
>>> (m * v).flatten()
array([-4.26341971, -2.50127873, -1.25      , -0.09493509,  1.77025575])
```

Accessing blocks

```
>>> v.get_block(1)
array([0.1       , 1.14872127])
>>> m.get_block(1, 0).toarray()
array([[ 0.        ,  0.        ],
       [-1.64872127,  1.        ]])
```

Empty blocks in a *BlockMatrix* return *None*:

```
>>> print(m.get_block(1, 1))
None
```

The dimensions of a blocks in a *BlockMatrix* can be set without setting a block:

```
>>> m2 = BlockMatrix(2, 2)
>>> m2.set_row_size(0, 5)
>>> m2.set_block(0, 0, m.get_block(0, 0))
Traceback (most recent call last):
  ...
ValueError: Incompatible row dimensions for row 0; got 2; expected 5.0
```

Note that operations on *BlockVector* and *BlockMatrix* cannot be performed until the dimensions are fully specified:

```
>>> v2 = BlockVector(3)
>>> v + v2
Traceback (most recent call last):
  ...
NotFullyDefinedBlockVectorError: Operation not allowed with None blocks.
>>> m2 = BlockMatrix(3, 3)
>>> m2 * 2
Traceback (most recent call last):
  ...
NotFullyDefinedBlockMatrixError: Operation not allowed with None rows. Specify at least␣
↪one block in every row
```

The *has_none* property can be used to see if a *BlockVector* is fully specified. If *has_none* returns *True*, then there are *None* blocks, and the *BlockVector* is not fully specified.

```
>>> v.has_none
False
>>> v2.has_none
True
```

For *BlockMatrix*, use the *has_undefined_row_sizes()* and *has_undefined_col_sizes()* methods:

```
>>> m.has_undefined_row_sizes()
False
>>> m.has_undefined_col_sizes()
False
>>> m2.has_undefined_row_sizes()
True
>>> m2.has_undefined_col_sizes()
True
```

To efficiently iterate over non-empty blocks in a *BlockMatrix*, use the *get_block_mask()* method, which returns a 2-D array indicating where the non-empty blocks are:

```
>>> m.get_block_mask(copy=False)
array([[ True,  True,  True],
       [ True, False,  True],
       [ True,  True, False]])
>>> for i, j in zip(*np.nonzero(m.get_block_mask(copy=False))):
...     assert m.get_block(i, j) is not None
```

Copying data:

```
>>> v2 = v.copy()
>>> v2.flatten()
array([-0.67025575, -1.2       ,  0.1       ,  1.14872127,  1.25      ])
>>> v2 = v.copy_structure()
>>> v2.block_sizes()
array([2, 2, 1])
>>> v2.copyfrom(v)
>>> v2.flatten()
array([-0.67025575, -1.2       ,  0.1       ,  1.14872127,  1.25      ])
>>> m2 = m.copy()
>>> (m - m2).tocoo().toarray()
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
>>> m2 = m.copy_structure()
>>> m2.has_undefined_row_sizes()
False
>>> m2.has_undefined_col_sizes()
False
>>> m2.copyfrom(m)
>>> (m - m2).tocoo().toarray()
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Nested blocks:

```
>>> v2 = BlockVector(2)
>>> v2.set_block(0, v)
>>> v2.set_block(1, np.ones(2))
>>> v2.block_sizes()
array([5, 2])
>>> v2.flatten()
array([-0.67025575, -1.2       ,  0.1       ,  1.14872127,  1.25      ,
        1.        ,  1.        ])
>>> v3 = v2.copy_structure()
>>> v3.fill(1)
>>> (v2 + v3).flatten()
array([ 0.32974425, -0.2       ,  1.1       ,  2.14872127,  2.25      ,
        2.        ,  2.        ])
>>> np.abs(v2).flatten()
array([0.67025575, 1.2       , 0.1       , 1.14872127, 1.25      ,
       1.        , 1.        ])
>>> v2.get_block(0)
BlockVector(3,)
```

Nested *BlockMatrix* applications work similarly.

For more information, see the `API documentation`.

MPI-Based Block Vectors and Matrices

PyNumero's MPI-based block vectors and matrices (`MPIBlockVector` and `MPIBlockMatrix`) behave very similarly to *BlockVector* and *BlockMatrix*. The primary difference is in construction. With *MPIBlockVector* and *MPIBlockMatrix*, each block is owned by either a single process/rank or all processes/ranks.

Consider the following example (in a file called "parallel_vector_ops.py").

```
# ____________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ____________________________________________________________________________________

import numpy as np
from pyomo.common.dependencies import mpi4py
from pyomo.contrib.pynumero.sparse.mpi_block_vector import MPIBlockVector


def main():
    comm = mpi4py.MPI.COMM_WORLD
    rank = comm.Get_rank()

    owners = [2, 0, 1, -1]
    x = MPIBlockVector(4, rank_owner=owners, mpi_comm=comm)
    x.set_block(owners.index(rank), np.ones(3) * (rank + 1))
    x.set_block(3, np.array([1, 2, 3]))

    y = MPIBlockVector(4, rank_owner=owners, mpi_comm=comm)
    y.set_block(owners.index(rank), np.ones(3) * (rank + 1))
    y.set_block(3, np.array([1, 2, 3]))

    z1: MPIBlockVector = x + y  # add x and y
    z2 = x.dot(y)  # dot product
    z3 = np.abs(x).max()  # infinity norm

    z1_local = z1.make_local_copy()
    if rank == 0:
        print(z1_local.flatten())
        print(z2)
        print(z3)

    return z1_local, z2, z3


if __name__ == '__main__':
    main()
```

This example can be run with

```
mpirun -np 3 python -m mpi4py parallel_vector_ops.py
```

The output is

```
[6. 6. 6. 2. 2. 2. 4. 4. 4. 2. 4. 6.]
56.0
3
```

Note that the *make_local_copy()* method is not efficient and should only be used for debugging.

The -1 in *owners* means that the block at that index (index 3 in this example) is owned by all processes. The non-negative integer values indicate that the block at that index is owned by the process with rank equal to the value. In this example, rank 0 owns block 1, rank 1 owns block 2, and rank 2 owns block 0. Block 3 is owned by all ranks. Note that blocks should only be set if the process/rank owns that block.

The operations performed with *MPIBlockVector* are identical to the same operations performed with *BlockVector* (or even NumPy arrays), except that the operations are now performed in parallel.

MPIBlockMatrix construction is very similar. Consider the following example in a file called “parallel_matvec.py”.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

import numpy as np
from pyomo.common.dependencies import mpi4py
from pyomo.contrib.pynumero.sparse.mpi_block_vector import MPIBlockVector
from pyomo.contrib.pynumero.sparse.mpi_block_matrix import MPIBlockMatrix
from scipy.sparse import random


def main():
    comm = mpi4py.MPI.COMM_WORLD
    rank = comm.Get_rank()

    owners = [0, 1, 2, -1]
    x = MPIBlockVector(4, rank_owner=owners, mpi_comm=comm)

    owners = np.array([[0, -1, -1, 0], [-1, 1, -1, 1], [-1, -1, 2, 2]])
    a = MPIBlockMatrix(3, 4, rank_ownership=owners, mpi_comm=comm)

    np.random.seed(0)
    x.set_block(3, np.random.uniform(-10, 10, size=10))

    np.random.seed(rank)
    x.set_block(rank, np.random.uniform(-10, 10, size=10))
    a.set_block(rank, rank, random(10, 10, density=0.1))
    a.set_block(rank, 3, random(10, 10, density=0.1))

    b = a * x  # parallel matrix-vector dot product

    # check the answer
```

(continues on next page)

(continued from previous page)

```
    local_x = x.make_local_copy().flatten()
    local_a = a.to_local_array()
    local_b = b.make_local_copy().flatten()

    err = np.abs(local_a.dot(local_x) - local_b).max()

    if rank == 0:
        print('error: ', err)

    return err


if __name__ == '__main__':
    main()
```

Which can be run with

```
mpirun -np 3 python -m mpi4py parallel_matvec.py
```

The output is

```
error:  4.440892098500626e-16
```

The most difficult part of using *MPIBlockVector* and *MPIBlockMatrix* is determining the best structure and rank ownership to maximize parallel efficiency.

Other examples may be found at https://github.com/Pyomo/pyomo/tree/main/pyomo/contrib/pynumero/examples.

Backward Compatibility

While PyNumero is a third-party contribution to Pyomo, we intend to maintain the stability of its core functionality. The core functionality of PyNumero consists of:

1. The `NLP` API and `PyomoNLP` implementation of this API
2. HSL and MUMPS linear solver interfaces
3. `BlockVector` and `BlockMatrix` classes
4. CyIpopt and SciPy solver interfaces

Other parts of PyNumero, such as `ExternalGreyBoxBlock` and `ImplicitFunctionSolver`, are experimental and subject to change without notice.

PyNumero API

pyomo.contrib.pynumero

Developers

The development team includes:

- Jose Santiago Rodriguez
- Michael Bynum
- Carl Laird
- Bethany Nicholson

- Robby Parker
- John Siirola

Packages built on PyNumero

- https://github.com/Pyomo/pyomo/tree/main/pyomo/contrib/interior_point
- https://github.com/parapint/parapint

Papers utilizing PyNumero

- Rodriguez, J. S., Laird, C. D., & Zavala, V. M. (2020). Scalable preconditioning of block-structured linear algebra systems using ADMM. Computers & Chemical Engineering, 133, 106478.

3.3.9 z3 SMT Sat Solver Interface

The z3 Satisfiability Solver interface can convert pyomo variables and expressions for use with the z3 Satisfiability Solver

Installation

z3 is required for use of the Sat Solver can be installed via the command

```
pip install z3-solver
```

Using z3 Sat Solver

To use the sat solver define your pyomo model as usual:

```
Required import
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.satsolver.satsolver import SMTSatSolver

Create a simple model
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)

Invoke the sat solver using optional argument model to automatically process
pyomo model
>>> is_feasible = SMTSatSolver(model = m).check()
```

3.4 Analysis in Pyomo

3.4.1 Generating Alternative (Near-)Optimal Solutions

Optimization solvers are generally designed to return a feasible solution to the user. However, there are many applications where a user needs more context than this result. For example,

- alternative solutions can support an assessment of trade-offs between competing objectives;
- if the optimization formulation may be inaccurate or untrustworthy, then comparisons amongst alternative solutions provide additional insights into the reliability of these model predictions; or

- the user may have unexpressed objectives or constraints, which only are realized in later stages of model analysis.

The *alternative-solutions library* provides a variety of functions that can be used to generate optimal or near-optimal solutions for a pyomo model. Conceptually, these functions are like pyomo solvers. They can be configured with solver names and options, and they return a list of solutions for the pyomo model. However, these functions are independent of pyomo's solver interface because they return a custom solution object.

The following functions are defined in the alternative-solutions library:

- `enumerate_binary_solutions`
  - Finds alternative optimal solutions for a binary problem using no-good cuts.
- `enumerate_linear_solutions`
  - Finds alternative optimal solutions for a (mixed-integer) linear program.
- `enumerate_linear_solutions_soln_pool`
  - Finds alternative optimal solutions for a (mixed-binary) linear program using Gurobi's solution pool feature.
- `gurobi_generate_solutions`
  - Finds alternative optimal solutions for discrete variables using Gurobi's built-in solution pool capability.
- `obbt_analysis_bounds_and_solutions`
  - Calculates the bounds on each variable by solving a series of min and max optimization problems where each variable is used as the objective function. This can be applied to any class of problem supported by the selected solver.

Basic Usage Example

Many of the functions in the alternative-solutions library have similar options, so we simply illustrate the `enumerate_binary_solutions` function. We define a simple knapsack example whose alternative solutions have integer objective values ranging from 0 to 90.

```
>>> import pyomo.environ as pyo

>>> values = [10, 40, 30, 50]
>>> weights = [5, 4, 6, 3]
>>> capacity = 10

>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(range(4), within=pyo.Binary)
>>> m.o = pyo.Objective(expr=sum(values[i] * m.x[i] for i in range(4)), sense=pyo.
↪maximize)
>>> m.c = pyo.Constraint(expr=sum(weights[i] * m.x[i] for i in range(4)) <= capacity)
```

We can execute the `enumerate_binary_solutions` function to generate a list of `Solution` objects that represent alternative optimal solutions:

```
>>> import pyomo.contrib.alternative_solutions as aos
>>> solns = aos.enumerate_binary_solutions(m, num_solutions=100, solver="glpk")
>>> assert len(solns) == 10
```

Each `Solution` object contains information about the objective and variables, and it includes various methods to access this information. For example:

```
>>> print(solns[0])
{
    "fixed_variables": [],
    "objective": "o",
    "objective_value": 90.0,
    "solution": {
        "x[0]": 0,
        "x[1]": 1,
        "x[2]": 0,
        "x[3]": 1
    }
}
```

Gap Usage Example

When we only want some of the solutions based off a tolerance away from optimal, this can be done using the abs_opt_gap parameter. This is shown in the following simple knapsack examples where the weights and values are the same.

```
>>> import pyomo.environ as pyo
>>> import pyomo.contrib.alternative_solutions as aos

>>> values = [10,9,2,1,1]
>>> weights = [10,9,2,1,1]

>>> K = len(values)
>>> capacity = 12

>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(range(K), within=pyo.Binary)
>>> m.o = pyo.Objective(expr=sum(values[i] * m.x[i] for i in range(K)), sense=pyo.
↪maximize)
>>> m.c = pyo.Constraint(expr=sum(weights[i] * m.x[i] for i in range(K)) <= capacity)

>>> solns = aos.enumerate_binary_solutions(m, num_solutions=10, solver="glpk", abs_opt_
↪gap = 0.0)
>>> assert(len(solns) == 4)
```

In this example, we only get the four Solution objects that have an objective_value of 12. Note that while we wanted only those four solutions with no optimality gap, using a gap of half the smallest value (in this case .5) will return the same solutions and avoids any machine precision issues.

```
>>> import pyomo.environ as pyo
>>> import pyomo.contrib.alternative_solutions as aos

>>> values = [10,9,2,1,1]
>>> weights = [10,9,2,1,1]

>>> K = len(values)
>>> capacity = 12

>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(range(K), within=pyo.Binary)
```

(continues on next page)

(continued from previous page)

```
>>> m.o = pyo.Objective(expr=sum(values[i] * m.x[i] for i in range(K)), sense=pyo.
↪maximize)
>>> m.c = pyo.Constraint(expr=sum(weights[i] * m.x[i] for i in range(K)) <= capacity)

>>> solns = aos.enumerate_binary_solutions(m, num_solutions=10, solver="glpk", abs_opt_
↪gap = 0.5)
>>> assert(len(solns) == 4)
>>> for soln in sorted(solns, key=lambda s: str(s.get_variable_name_values())):
...     print(soln)
{
    "fixed_variables": [],
    "objective": "o",
    "objective_value": 12.0,
    "solution": {
        "x[0]": 0,
        "x[1]": 1,
        "x[2]": 1,
        "x[3]": 0,
        "x[4]": 1
    }
}
{
    "fixed_variables": [],
    "objective": "o",
    "objective_value": 12.0,
    "solution": {
        "x[0]": 0,
        "x[1]": 1,
        "x[2]": 1,
        "x[3]": 1,
        "x[4]": 0
    }
}
{
    "fixed_variables": [],
    "objective": "o",
    "objective_value": 12.0,
    "solution": {
        "x[0]": 1,
        "x[1]": 0,
        "x[2]": 0,
        "x[3]": 1,
        "x[4]": 1
    }
}
{
    "fixed_variables": [],
    "objective": "o",
    "objective_value": 12.0,
    "solution": {
        "x[0]": 1,
        "x[1]": 0,
```

(continues on next page)

(continued from previous page)

```
        "x[2]": 1,
        "x[3]": 0,
        "x[4]": 0
    }
}
```

Interface Documentation

pyomo.contrib.alternative_solutions.**enumerate_binary_solutions**(*model*, ** (Keyword-only parameters separator (PEP 3102))*, *num_solutions=10*, *variables=None*, *rel_opt_gap=None*, *abs_opt_gap=None*, *search_mode='optimal'*, *solver='gurobi'*, *solver_options={}*, *tee=False*, *seed=None*)

Finds alternative optimal solutions for a binary problem using no-good cuts.

This function implements a no-good cuts technique inheriting from Balas's work on Canonical Cuts [BJ72].

Parameters

- **model** (ConcreteModel) – A concrete Pyomo model
- **num_solutions** (*int*) – The maximum number of solutions to generate.
- **variables** (*None or a collection of Pyomo _GeneralVarData variables*) – The variables for which bounds will be generated. None indicates that all variables will be included. Alternatively, a collection of _GeneralVarData variables can be provided.
- **rel_opt_gap** (*float or None*) – The relative optimality gap for the original objective for which variable bounds will be found. None indicates that a relative gap constraint will not be added to the model.
- **abs_opt_gap** (*float or None*) – The absolute optimality gap for the original objective for which variable bounds will be found. None indicates that an absolute gap constraint will not be added to the model.
- **search_mode** (*'optimal', 'random', or 'hamming'*) – Indicates the mode that is used to generate alternative solutions. The optimal mode finds the next best solution. The random mode finds an alternative solution in the direction of a random ray. The hamming mode iteratively finds solution that maximize the hamming distance from previously discovered solutions.
- **solver** (*string*) – The solver to be used.
- **solver_options** (*dict*) – Solver option-value pairs to be passed to the solver.
- **tee** (*boolean*) – Boolean indicating that the solver output should be displayed.
- **seed** (*int*) – Optional integer seed for the numpy random number generator

Returns

A list of Solution objects. [Solution]

Return type

solutions

`pyomo.contrib.alternative_solutions.enumerate_linear_solutions`(*model*, *, *num_solutions=10*, *rel_opt_gap=None*, *abs_opt_gap=None*, *zero_threshold=1e-05*, *search_mode='optimal'*, *solver='gurobi'*, *solver_options={}*, *tee=False*, *seed=None*)

Finds alternative optimal solutions a (mixed-integer) linear program.

This function implements the technique described here:

> S. Lee, C. Phalakornkule, M.M. Domach, and I.E. Grossmann, "Recursive MILP model for finding all the alternative optima in LP models for metabolic networks", Computers and Chemical Engineering, 24 (2000) 711-716.

Parameters

- **model** (`ConcreteModel`) – A concrete Pyomo model
- **num_solutions** (*int*) – The maximum number of solutions to generate.
- **rel_opt_gap** (*float or None*) – The relative optimality gap for the original objective for which variable bounds will be found. None indicates that a relative gap constraint will not be added to the model.
- **abs_opt_gap** (*float or None*) – The absolute optimality gap for the original objective for which variable bounds will be found. None indicates that an absolute gap constraint will not be added to the model.
- **zero_threshold** (*float*) – The threshold for which a continuous variables' value is considered to be equal to zero.
- **search_mode** (*'optimal', 'random', or 'norm'*) – Indicates the mode that is used to generate alternative solutions. The optimal mode finds the next best solution. The random mode finds an alternative solution in the direction of a random ray. The norm mode iteratively finds solution that maximize the L2 distance from previously discovered solutions.
- **solver** (*string*) – The solver to be used.
- **solver_options** (*dict*) – Solver option-value pairs to be passed to the solver.
- **tee** (*boolean*) – Boolean indicating that the solver output should be displayed.
- **seed** (*int*) – Optional integer seed for the numpy random number generator

Returns

A list of Solution objects. [Solution]

Return type

solutions

`pyomo.contrib.alternative_solutions.lp_enum_solnpool.enumerate_linear_solutions_soln_pool`(*model*, *num_solutions=1*, *rel_opt_gap=Non*, *abs_opt_gap=No*, *zero_threshold=1*
05, *solver_options={*, *tee=False*)

Finds alternative optimal solutions for a (mixed-binary) linear program using Gurobi's solution pool feature.

Parameters

- **model** (ConcreteModel) – A concrete Pyomo model
- **num_solutions** (*int*) – The maximum number of solutions to generate.
- **variables** (*None or a collection of Pyomo _GeneralVarData variables*) – The variables for which bounds will be generated. None indicates that all variables will be included. Alternatively, a collection of _GeneralVarData variables can be provided.
- **rel_opt_gap** (*float or None*) – The relative optimality gap for the original objective for which variable bounds will be found. None indicates that a relative gap constraint will not be added to the model.
- **abs_opt_gap** (*float or None*) – The absolute optimality gap for the original objective for which variable bounds will be found. None indicates that an absolute gap constraint will not be added to the model.
- **zero_threshold** (*float*) – The threshold for which a continuous variables' value is considered to be equal to zero.
- **solver_options** (*dict*) – Solver option-value pairs to be passed to the solver.
- **tee** (*boolean*) – Boolean indicating that the solver output should be displayed.

Returns
A list of Solution objects. [Solution]

Return type
solutions

pyomo.contrib.alternative_solutions.**gurobi_generate_solutions**(*model*, *, *num_solutions=10*, *rel_opt_gap=None*, *abs_opt_gap=None*, *solver_options={}*, *tee=False*)

Finds alternative optimal solutions for discrete variables using Gurobi's built-in Solution Pool capability.

This method defaults to the optimality-enforced discovery method with PoolSearchMode = 2. There are two other options, standard single optimal solution (PoolSearchMode = 0) and best-effort discovery with no guarantees (PoolSearchMode = 1). Please consult the Gurobi documentation on PoolSearchMode for details on impact on Gurobi results. Changes to this mode can be made by included PoolSearchMode set to the intended value in solver_options.

Parameters

- **model** (ConcreteModel) – A concrete Pyomo model.
- **num_solutions** (*int*) – The maximum number of solutions to generate. This parameter maps to the PoolSolutions parameter in Gurobi.
- **rel_opt_gap** (*non-negative float or None*) – The relative optimality gap for allowable alternative solutions. None implies that there is no limit on the relative optimality gap (i.e. that any feasible solution can be considered by Gurobi). This parameter maps to the PoolGap parameter in Gurobi.
- **abs_opt_gap** (*non-negative float or None*) – The absolute optimality gap for allowable alternative solutions. None implies that there is no limit on the absolute optimality gap (i.e. that any feasible solution can be considered by Gurobi). This parameter maps to the PoolGapAbs parameter in Gurobi.
- **solver_options** (*dict*) – Solver option-value pairs to be passed to the Gurobi solver.
- **tee** (*boolean*) – Boolean indicating that the solver output should be displayed.

Returns
A list of Solution objects. [Solution]

Return type
solutions

`pyomo.contrib.alternative_solutions.`**`obbt_analysis_bounds_and_solutions`**(*model*, *, *variables=None*, *rel_opt_gap=None*, *abs_opt_gap=None*, *refine_discrete_bounds=False*, *warmstart=True*, *solver='gurobi'*, *solver_options={}*, *tee=False*)

Calculates the bounds on each variable by solving a series of min and max optimization problems where each variable is used as the objective function This can be applied to any class of problem supported by the selected solver.

Parameters

- **model** (`ConcreteModel`) – A concrete Pyomo model.
- **variables** (*None or a collection of Pyomo _GeneralVarData variables*) – The variables for which bounds will be generated. None indicates that all variables will be included. Alternatively, a collection of _GeneralVarData variables can be provided.
- **rel_opt_gap** (*float or None*) – The relative optimality gap for the original objective for which variable bounds will be found. None indicates that a relative gap constraint will not be added to the model.
- **abs_opt_gap** (*float or None*) – The absolute optimality gap for the original objective for which variable bounds will be found. None indicates that an absolute gap constraint will not be added to the model.
- **refine_discrete_bounds** (*boolean*) – Boolean indicating that new constraints should be added to the model at each iteration to tighten the bounds for discrete variables.
- **warmstart** (*boolean*) – Boolean indicating that the solver should be warmstarted from the best previously discovered solution.
- **solver** (*string*) – The solver to be used.
- **solver_options** (*dict*) – Solver option-value pairs to be passed to the solver.
- **tee** (*boolean*) – Boolean indicating that the solver output should be displayed.

Returns

- *variable_ranges* – A Pyomo ComponentMap containing the bounds for each variable. {variable: (lower_bound, upper_bound)}. An exception is raised when the solver encountered an issue.
- *solutions* – [Solution]

class `pyomo.contrib.alternative_solutions.`**`Solution`**(*model*, *variable_list*, *include_fixed=True*, *objective=None*)

A class to store solutions from a Pyomo model.

`variables`

A map between Pyomo variables and their values for a solution.

Type
ComponentMap

`fixed_vars`
The set of Pyomo variables that are fixed in a solution.

Type
ComponentSet

`objective`
A map between Pyomo objectives and their values for a solution.

Type
ComponentMap

`pprint():`
Prints a solution.

`get_variable_name_values(self, ignore_fixed_vars=False):`
Get a dictionary of variable name-variable value pairs.

`get_fixed_variable_names(self):`
Get a list of fixed-variable names.

`get_objective_name_values(self):`
Get a dictionary of objective name-objective value pairs.

3.4.2 Community Detection for Pyomo models

This package separates model components (variables, constraints, and objectives) into different communities distinguished by the degree of connectivity between community members.

Description of Package and `detect_communities` function

The community detection package allows users to obtain a community map of a Pyomo model - a Python dictionary-like object that maps sequential integer values to communities within the Pyomo model. The package takes in a model, organizes the model components into a graph of nodes and edges, then uses Louvain community detection (Blondel et al, 2008) to determine the communities that exist within the model.

In graph theory, a community is defined as a subset of nodes that have a greater degree of connectivity within themselves than they do with the rest of the nodes in the graph. In the context of Pyomo models, a community represents a subproblem within the overall optimization problem. Identifying these subproblems and then solving them independently can save computational work compared with trying to solve the entire model at once. Thus, it can be very useful to know the communities that exist in a model.

The manner in which the graph of nodes and edges is constructed from the model directly affects the community detection. Thus, this package provides the user with a lot of control over the construction of the graph. The function we use for this community detection is shown below:

`pyomo.contrib.community_detection.detection.`**`detect_communities`**(*model*, *type_of_community_map='constraint'*, *with_objective=True*, *weighted_graph=True*, *random_seed=None*, *use_only_active_components=True*)

Detects communities in a Pyomo optimization model

This function takes in a Pyomo optimization model and organizes the variables and constraints into a graph of nodes and edges. Then, by using Louvain community detection on the graph, a dictionary (community_map) is created, which maps (arbitrary) community keys to the detected communities within the model.

Parameters

- **model** (Block) – a Pyomo model or block to be used for community detection
- **type_of_community_map** (*str, optional*) – a string that specifies the type of community map to be returned, the default is 'constraint'. 'constraint' returns a dictionary (community_map) with communities based on constraint nodes, 'variable' returns a dictionary (community_map) with communities based on variable nodes, 'bipartite' returns a dictionary (community_map) with communities based on a bipartite graph (both constraint and variable nodes)
- **with_objective** (*bool, optional*) – a Boolean argument that specifies whether or not the objective function is included in the model graph (and thus in 'community_map'); the default is True
- **weighted_graph** (*bool, optional*) – a Boolean argument that specifies whether community_map is created based on a weighted model graph or an unweighted model graph; the default is True (type_of_community_map='bipartite' creates an unweighted model graph regardless of this parameter)
- **random_seed** (*int, optional*) – an integer that is used as the random seed for the (heuristic) Louvain community detection
- **use_only_active_components** (*bool, optional*) – a Boolean argument that specifies whether inactive constraints/objectives are included in the community map

Returns

The CommunityMap object acts as a Python dictionary, mapping integer keys to tuples containing two lists (which contain the components in the given community) - a constraint list and variable list. Furthermore, the CommunityMap object stores relevant information about the given community map (dict), such as the model used to create it, its networkX representation, etc.

Return type

CommunityMap object (dict-like object)

As stated above, the characteristics of the NetworkX graph of the Pyomo model are very important to the community detection. The main graph features the user can specify are the type of community map, whether the graph is weighted or unweighted, and whether the objective function(s) is included in the graph generation. Below, the significance and reasoning behind including each of these options are explained in greater depth.

Type of Community Map (*type_of_community_map*)

In this package's main function (`detect_communities`), the user can select `'bipartite'`, `'constraint'`, or `'variable'` as an input for the 'type_of_community_map' argument, and these result in a community map based on a bipartite graph, a constraint node graph, or a variable node graph (respectively).

If the user sets `type_of_community_map='constraint'`, then each entry in the community map (which is a dictionary) contains a list of all the constraints in the community as well as all the variables contained in those constraints. For the model graph, a node is created for every active constraint in the model, an edge between two constraint nodes is created only if those two constraint equations share a variable, and the weight of each edge is equal to the number of variables the two constraint equations have in common.

If the user sets `type_of_community_map='variable'`, then each entry in the community map (which is a dictionary) contains a list of all the variables in the community as well as all the constraints that contain those variables. For the model graph, a node is created for every variable in the model, an edge between two variable nodes is created only if those two variables occur in the same constraint equation, and the weight of each edge is equal to the number of constraint equations in which the two variables occur together.

If the user sets `type_of_community_map='bipartite'`, then each entry in the community map (which is a dictionary) is simply all of the nodes in the community but split into a list of constraints and a list of variables. For the model graph, a node is created for every variable and every constraint in the model. An edge is created between a constraint node and a variable node only if the constraint equation contains the variable. (Edges are not drawn between nodes of the same type in a bipartite graph.) And as for the edge weights, the edges in the bipartite graph are unweighted regardless of what the user specifies for the `weighted_graph` parameter. (This is because for our purposes, the number of times a variable appears in a constraint is not particularly useful.)

Weighted Graph/Unweighted Graph (*weighted_graph*)

The Louvain community detection algorithm takes edge weights into account, so depending on whether the graph is weighted or unweighted, the communities that are found will vary. This can be valuable depending on how the user intends to use the community detection information. For example, if a user plans on feeding that information into an algorithm, the algorithm may be better suited to the communities detected in a weighted graph (or vice versa).

With/Without Objective in the Graph (*with_objective*)

This argument determines whether the objective function(s) will be included when creating the graphical representation of the model and thus whether the objective function(s) will be included in the community map. Some models have an objective function that contains so many of the model variables that it obscures potential communities within a model. Thus, it can be useful to call `detect_communities(model, with_objective=False)` on such a model to see whether isolating the other components of the model provides any new insights.

External Packages

- NetworkX
- Python-Louvain

The community detection package relies on two external packages, the NetworkX package and the Louvain community detection package. Both of these packages can be installed at the following URLs (respectively):

https://pypi.org/project/networkx/

https://pypi.org/project/python-louvain/

The pip install and conda install commands are included below as well:

```
pip install networkx
pip install python-louvain

conda install -c anaconda networkx
conda install -c conda-forge python-louvain
```

Usage Examples

Let's start off by taking a look at how we can use `detect_communities` to create a CommunityMap object. We'll first use a model from Allman et al, 2019 :

```
Required Imports
>>> from pyomo.contrib.community_detection.detection import detect_communities,␣
↪CommunityMap, generate_model_graph
>>> from pyomo.contrib.mindtpy.tests.eight_process_problem import EightProcessFlowsheet
>>> from pyomo.core import ConcreteModel, Var, Constraint
>>> import networkx as nx

Let's define a model for our use
>>> def decode_model_1():
```

(continues on next page)

(continued from previous page)

```
...     model = m = ConcreteModel()
...     m.x1 = Var(initialize=-3)
...     m.x2 = Var(initialize=-1)
...     m.x3 = Var(initialize=-3)
...     m.x4 = Var(initialize=-1)
...     m.c1 = Constraint(expr=m.x1 + m.x2 <= 0)
...     m.c2 = Constraint(expr=m.x1 - 3 * m.x2 <= 0)
...     m.c3 = Constraint(expr=m.x2 + m.x3 + 4 * m.x4 ** 2 == 0)
...     m.c4 = Constraint(expr=m.x3 + m.x4 <= 0)
...     m.c5 = Constraint(expr=m.x3 ** 2 + m.x4 ** 2 - 10 == 0)
...     return model
>>> model = m = decode_model_1()
>>> seed = 5  # To be used as a random seed value for the heuristic Louvain community␣
↪detection

Let's create an instance of the CommunityMap class (which is what gets returned by the
function detect_communities):
>>> community_map_object = detect_communities(model, type_of_community_map='bipartite',␣
↪random_seed=seed)
```

This community map object has many attributes that contain the relevant information about the community map itself (such as the parameters used to create it, the networkX representation, and other useful information).

An important point to note is that the community_map attribute of the CommunityMap class is the actual dictionary that maps integers to the communities within the model. It is expected that the user will be most interested in the actual dictionary itself, so dict-like usage is permitted.

If a user wishes to modify the actual dictionary (the community_map attribute of the CommunityMap object), creating a deep copy is highly recommended (or else any destructive modifications could have unintended consequences): `new_community_map = copy.deepcopy(community_map_object.community_map)`

Let's take a closer look at the actual community map object generated by *detect_communities*:

```
>>> print(community_map_object)
{0: (['c1', 'c2'], ['x1', 'x2']), 1: (['c3', 'c4', 'c5'], ['x3', 'x4'])}
```

Printing a community map object is made to be user-friendly (by showing the community map with components replaced by their strings). However, if the default Pyomo representation of components is desired, then the community_map attribute or the *repr()* function can be used:

```
>>> print(community_map_object.community_map)
{0: ([<pyomo.core.base.constraint.ScalarConstraint object at ...>, <pyomo.core.base.
↪constraint.ScalarConstraint object at ...>], [<pyomo.core.base.var.ScalarVar object at␣
↪...>, <pyomo.core.base.var.ScalarVar object at ...>]), 1: ([<pyomo.core.base.
↪constraint.ScalarConstraint object at ...>, <pyomo.core.base.constraint.
↪ScalarConstraint object at ...>, <pyomo.core.base.constraint.ScalarConstraint object␣
↪at ...>], [<pyomo.core.base.var.ScalarVar object at ...>, <pyomo.core.base.var.
↪ScalarVar object at ...>])}
>>> print(repr(community_map_object))
{0: ([<pyomo.core.base.constraint.ScalarConstraint object at ...>, <pyomo.core.base.
↪constraint.ScalarConstraint object at ...>], [<pyomo.core.base.var.ScalarVar object at␣
↪...>, <pyomo.core.base.var.ScalarVar object at ...>]), 1: ([<pyomo.core.base.
↪constraint.ScalarConstraint object at ...>, <pyomo.core.base.constraint.
↪ScalarConstraint object at ...>, <pyomo.core.base.constraint.ScalarConstraint object␣
```

(continues on next page)

(continued from previous page)

```
↪at ...>], [<pyomo.core.base.var.ScalarVar object at ...>, <pyomo.core.base.var.
↪ScalarVar object at ...>])}
```

generate_structured_model **method of CommunityMap objects**

It may be useful to create a new model based on the communities found in the model - we can use the `generate_structured_model` method of the CommunityMap class to do this. Calling this method on a CommunityMap object returns a new model made up of blocks that correspond to each of the communities found in the original model. Let's take a look at the example below:

```
Use the CommunityMap object made from the first code example
>>> structured_model = community_map_object.generate_structured_model()
>>> structured_model.pprint()
2 Set Declarations
    b_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    2 : {0, 1}
    equality_constraint_list_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    1 :    {1,}

1 Var Declarations
    x2 : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  None :  None :  None : False :  True :  Reals

1 Constraint Declarations
    equality_constraint_list : Equality Constraints for the different forms of a␣
↪given variable
        Size=1, Index=equality_constraint_list_index, Active=True
        Key : Lower : Body         : Upper : Active
          1 :   0.0 : b[0].x2 - x2 :   0.0 :   True

1 Block Declarations
    b : Size=2, Index=b_index, Active=True
        b[0] : Active=True
            2 Var Declarations
                x1 : Size=1, Index=None
                    Key  : Lower : Value : Upper : Fixed : Stale : Domain
                    None :  None :  None :  None : False :  True :  Reals
                x2 : Size=1, Index=None
                    Key  : Lower : Value : Upper : Fixed : Stale : Domain
                    None :  None :  None :  None : False :  True :  Reals

            2 Constraint Declarations
                c1 : Size=1, Index=None, Active=True
                    Key  : Lower : Body              : Upper : Active
                    None :  -Inf : b[0].x1 + b[0].x2 :   0.0 :   True
                c2 : Size=1, Index=None, Active=True
                    Key  : Lower : Body                : Upper : Active
                    None :  -Inf : b[0].x1 - 3*b[0].x2 :   0.0 :   True

            4 Declarations: x1 x2 c1 c2
```

(continues on next page)

(continued from previous page)

```
        b[1] : Active=True
            2 Var Declarations
                x3 : Size=1, Index=None
                    Key  : Lower : Value : Upper : Fixed : Stale : Domain
                    None :  None :  None :  None : False :  True :  Reals
                x4 : Size=1, Index=None
                    Key  : Lower : Value : Upper : Fixed : Stale : Domain
                    None :  None :  None :  None : False :  True :  Reals

            3 Constraint Declarations
                c3 : Size=1, Index=None, Active=True
                    Key  : Lower : Body                         : Upper : Active
                    None :   0.0 : x2 + b[1].x3 + 4*b[1].x4**2 :   0.0 :   True
                c4 : Size=1, Index=None, Active=True
                    Key  : Lower : Body               : Upper : Active
                    None :  -Inf : b[1].x3 + b[1].x4 :   0.0 :   True
                c5 : Size=1, Index=None, Active=True
                    Key  : Lower : Body                          : Upper : Active
                    None :   0.0 : b[1].x3**2 + b[1].x4**2 - 10 :   0.0 :   True

            5 Declarations: x3 x4 c3 c4 c5

5 Declarations: b_index b x2 equality_constraint_list_index equality_constraint_list
```

We see that there is an equality constraint list (*equality_constraint_list*) that has been created. This is due to the fact that the `detect_communities` function can return a community map that has Pyomo components (variables, constraints, or objectives) in more than one community, and thus, an equality_constraint_list is created to ensure that the new model still corresponds to the original model. This is explained in more detail below.

Consider the case where community detection is done on a constraint node graph - this would result in communities that are made up of the corresponding constraints as well as all the variables that occur in the given constraints. Thus, it is possible for certain Pyomo components to be in multiple communities (and a similar argument exists for community detection done on a variable node graph). As a result, our structured model (the model returned by the `generate_structured_model` method) may need to have several "copies" of a certain component. For example, a variable *original_model.x1* that exists in the original model may have corresponding forms *structured_model.b[0].x1*, *structured_model.b[0].x1*, *structured_model.x1*. In order for these components to meaningfully correspond to their counterparts in the original model, they must be bounded by equality constraints. Thus, we use an *equality_constraint_list* to bind different forms of a component from the original model.

The last point to make about this method is that variables will be created outside of blocks if (1) an objective is not inside a block (for example if the community detection is done *with_objective=False*) or if (2) an objective/constraint contains a variable that is not in the same block as the given objective/constraint.

***visualize_model_graph* method of CommunityMap objects**

If we want a visualization of the communities within the Pyomo model, we can use `visualize_model_graph` to do so. Let's take a look at how this can be done in the following example:

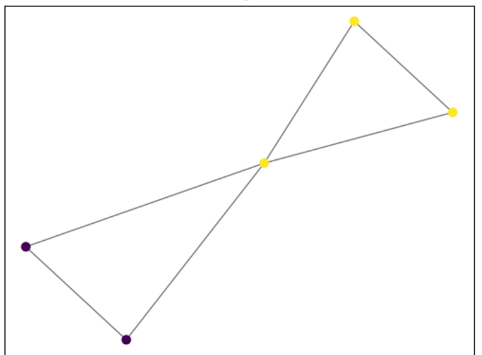
```
Create a CommunityMap object (so we can demonstrate the visualize_model_graph␣
↪method)
>>> community_map_object = cmo = detect_communities(model, type_of_community_map=
↪'bipartite', random_seed=seed)

Generate a matplotlib figure (left_figure) - a constraint graph of the community map
```

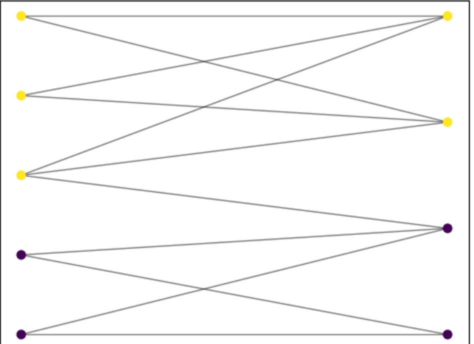
(continues on next page)

(continued from previous page)

```
>>> left_figure, _ = cmo.visualize_model_graph(type_of_graph='constraint')
>>> left_figure.show()

Now, we will generate the figure on the right (a bipartite graph of the community␣
↪map)
>>> right_figure, _ = cmo.visualize_model_graph(type_of_graph='bipartite')
>>> right_figure.show()
```

An example of the two separate graphs created for these two function calls is shown below:





These graph drawings very clearly demonstrate the communities within this model. The constraint graph (which is colored using the bipartite community map) shows a very simple illustration - one node for each constraint, with only one edge connecting the two communities (which represents the variable *m.x2* common to *m.c2* and *m.c3* in separate communities) The bipartite graph is slightly more complicated and we can see again how there is only one edge between the two communities and more edges within each community. This is an ideal situation for breaking a model into separate communities since there is little connectivity between the communities. Also, note that we can choose different graph types (such as a variable node graph, constraint node graph, or bipartite graph) for a given community map.

Let's try a more complicated model (taken from Duran & Grossmann, 1986) - this example will demonstrate how the same graph can be illustrated using different community maps (in the previous example we illustrated different graphs with a single community map):

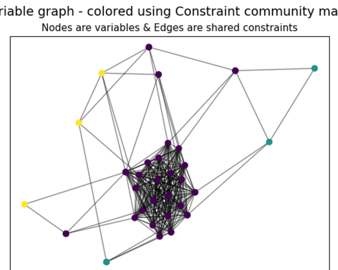
```
Define the model
>>> model = EightProcessFlowsheet()

Now, we follow steps similar to the example above (see above for explanations)
>>> community_map_object = cmo = detect_communities(model, type_of_community_map=
↪'constraint', random_seed=seed)
>>> left_fig, pos = cmo.visualize_model_graph(type_of_graph='variable')
>>> left_fig.show()

As we did before, we will use the returned 'pos' to create a consistent graph layout
>>> community_map_object = cmo = detect_communities(model, type_of_community_map=
↪'bipartite')
>>> middle_fig, _ = cmo.visualize_model_graph(type_of_graph='variable', pos=pos)
>>> middle_fig.show()

>>> community_map_object = cmo = detect_communities(model, type_of_community_map=
```

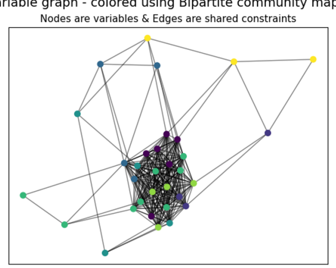
(continues on next page)

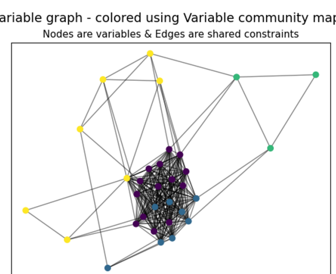
(continued from previous page)

```
↪'variable')
>>> right_fig, _ = cmo.visualize_model_graph(type_of_graph='variable', pos=pos)
>>> right_fig.show()
```

We can see an example for the three separate graphs created by these three function calls below:







The three graphs above are all variable graphs - which means the nodes represent variables in the model, and the edges represent constraint equations. The coloring differs because the three graphs rely on community maps that were created based on a constraint node graph, a bipartite graph, and a variable node graph (from left to right). For example, the community map that was generated from a constraint node graph (`type_of_community_map='constraint'`) resulted in three communities (as seen by the purple, yellow, and blue nodes).

***generate_model_graph* function**

Now, we will take a look at `generate_model_graph` - this function can be used to create a NetworkX graph for a Pyomo model (and is used in *detect_communities*). Here, we will create a NetworkX graph from the model in our first example and then create the edge and adjacency list for the graph.

`generate_model_graph` returns three things:

- a NetworkX graph of the given model
- a dictionary that maps the numbers used to represent the model components to the actual components (because Pyomo components cannot be directly added to a NetworkX graph)
- a dictionary that maps constraints to the variables in them.

For this example, we will only need the NetworkX graph of the model and the number-to-component mapping.

```
Define the model
>>> model = decode_model_1()

See above for the description of the items returned by 'generate_model_graph'
>>> model_graph, number_component_map, constr_var_map = generate_model_graph(model,␣
↪type_of_graph='constraint')

The next two lines create and implement a mapping to change the node values from␣
↪numbers into
strings. The second line uses this mapping to create string_model_graph, which has
the relabeled nodes (strings instead of numbers).

>>> string_map = dict((number, str(comp)) for number, comp in number_component_map.
↪items())
>>> string_model_graph = nx.relabel_nodes(model_graph, string_map)

Now, we print the edge list and the adjacency list:
```

(continues on next page)

(continued from previous page)

```
Edge List:
>>> for line in nx.generate_edgelist(string_model_graph): print(line)
c1 c2 {'weight': 2}
c1 c3 {'weight': 1}
c2 c3 {'weight': 1}
c3 c5 {'weight': 2}
c3 c4 {'weight': 2}
c4 c5 {'weight': 2}

Adjacency List:
>>> print(list(nx.generate_adjlist(string_model_graph)))
['c1 c2 c3', 'c2 c3', 'c3 c5 c4', 'c4 c5', 'c5']
```

It's worth mentioning that in the code above, we do not have to create `string_map` to create an edge list or adjacency list, but for the sake of having an easily understandable output, it is quite helpful. (Without relabeling the nodes, the output below would not have the strings of the components but instead would have integer values.) This code will hopefully make it easier for a user to do the same.

Functions in this Package

Main module for community detection integration with Pyomo models.

This module separates model components (variables, constraints, and objectives) into different communities distinguished by the degree of connectivity between community members.

Original implementation developed by Rahul Joglekar in the Grossmann research group.

class pyomo.contrib.community_detection.detection.**CommunityMap**(*community_map*, *type_of_community_map*, *with_objective*, *weighted_graph*, *random_seed*, *use_only_active_components*, *model*, *graph*, *graph_node_mapping*, *constraint_variable_map*, *graph_partition*)

This class is used to create CommunityMap objects which are returned by the detect_communities function. Instances of this class allow dict-like usage and store relevant information about the given community map, such as the model used to create them, their networkX representation, etc.

The CommunityMap object acts as a Python dictionary, mapping integer keys to tuples containing two lists (which contain the components in the given community) - a constraint list and variable list.

Methods: generate_structured_model visualize_model_graph

generate_structured_model()

Using the community map and the original model used to create this community map, we will create structured_model, which will be based on the original model but will place variables, constraints, and objectives into or outside of various blocks (communities) based on the community map.

Returns

structured_model – a Pyomo model that reflects the nature of the community map

Return type

Block

visualize_model_graph(*type_of_graph='constraint'*, *filename=None*, *pos=None*)

This function draws a graph of the communities for a Pyomo model.

The type_of_graph parameter is used to create either a variable-node graph, constraint-node graph, or bipartite graph of the Pyomo model. Then, the nodes are colored based on the communities they are in - which is based on the community map (self.community_map). A filename can be provided to save the figure, otherwise the figure is illustrated with matplotlib.

Parameters

- **type_of_graph** (`str, optional`) – a string that specifies the types of nodes drawn on the model graph, the default is 'constraint'. 'constraint' draws a graph with constraint nodes, 'variable' draws a graph with variable nodes, 'bipartite' draws a bipartite graph (with both constraint and variable nodes)
- **filename** (`str, optional`) – a string that specifies a path for the model graph illustration to be saved
- **pos** (`dict, optional`) – a dictionary that maps node keys to their positions on the illustration

Returns

- **fig** (*matplotlib figure*) – the figure for the model graph drawing
- **pos** (*dict*) – a dictionary that maps node keys to their positions on the illustration - can be used to create consistent layouts for graphs of a given model

pyomo.contrib.community_detection.detection.**detect_communities**(*model*, *type_of_community_map='constraint'*, *with_objective=True*, *weighted_graph=True*, *random_seed=None*, *use_only_active_components=True*)

Detects communities in a Pyomo optimization model

This function takes in a Pyomo optimization model and organizes the variables and constraints into a graph of nodes and edges. Then, by using Louvain community detection on the graph, a dictionary (community_map) is created, which maps (arbitrary) community keys to the detected communities within the model.

Parameters

- **model** (`Block`) – a Pyomo model or block to be used for community detection
- **type_of_community_map** (`str, optional`) – a string that specifies the type of community map to be returned, the default is 'constraint'. 'constraint' returns a dictionary (community_map) with communities based on constraint nodes, 'variable' returns a dictionary (community_map) with communities based on variable nodes, 'bipartite' returns a dictionary (community_map) with communities based on a bipartite graph (both constraint and variable nodes)
- **with_objective** (`bool, optional`) – a Boolean argument that specifies whether or not the objective function is included in the model graph (and thus in 'community_map'); the default is True
- **weighted_graph** (`bool, optional`) – a Boolean argument that specifies whether community_map is created based on a weighted model graph or an unweighted model graph; the default is True (type_of_community_map='bipartite' creates an unweighted model graph regardless of this parameter)

- **random_seed** (*int, optional*) – an integer that is used as the random seed for the (heuristic) Louvain community detection
- **use_only_active_components** (*bool, optional*) – a Boolean argument that specifies whether inactive constraints/objectives are included in the community map

Returns
The CommunityMap object acts as a Python dictionary, mapping integer keys to tuples containing two lists (which contain the components in the given community) - a constraint list and variable list. Furthermore, the CommunityMap object stores relevant information about the given community map (dict), such as the model used to create it, its networkX representation, etc.

Return type
CommunityMap object (dict-like object)

Model Graph Generator Code

pyomo.contrib.community_detection.community_graph.**generate_model_graph**(*model*, *type_of_graph*, *with_objective=True*, *weighted_graph=True*, *use_only_active_components=True*)

Creates a networkX graph of nodes and edges based on a Pyomo optimization model

This function takes in a Pyomo optimization model, then creates a graphical representation of the model with specific features of the graph determined by the user (see Parameters below).

(This function is designed to be called by detect_communities, but can be used solely for the purpose of creating model graphs as well.)

Parameters

- **model** (Block) – a Pyomo model or block to be used for community detection
- **type_of_graph** (*str*) – a string that specifies the type of graph that is created from the model 'constraint' creates a graph based on constraint nodes, 'variable' creates a graph based on variable nodes, 'bipartite' creates a graph based on constraint and variable nodes (bipartite graph).
- **with_objective** (*bool, optional*) – a Boolean argument that specifies whether or not the objective function is included in the graph; the default is True
- **weighted_graph** (*bool, optional*) – a Boolean argument that specifies whether a weighted or unweighted graph is to be created from the Pyomo model; the default is True (type_of_graph='bipartite' creates an unweighted graph regardless of this parameter)
- **use_only_active_components** (*bool, optional*) – a Boolean argument that specifies whether inactive constraints/objectives are included in the networkX graph

Returns

- **bipartite_model_graph/projected_model_graph** (*nx.Graph*) – a NetworkX graph with nodes and edges based on the given Pyomo optimization model
- **number_component_map** (*dict*) – a dictionary that (deterministically) maps a number to a component in the model
- **constraint_variable_map** (*dict*) – a dictionary that maps a numbered constraint to a list of (numbered) variables that appear in the constraint

3.4.3 Pyomo.DoE

Pyomo.DoE (Pyomo Design of Experiments) is a Python library for model-based design of experiments using science-based models.

Pyomo.DoE was originally created by **Jialu Wang** and **Alexander W. Dowling** at the University of Notre Dame as part of the Carbon Capture Simulation for Industry Impact (CCSI2) project, funded through the U.S. Department Of Energy Office of Fossil Energy with assistance from **John Siirola**, **Bethany Nicholson**, **Miranda Mundt**, and **Hailey Lynch**. Significant improvements and extensions were contributed by **Dan Laky**, and **Shuvashish Mondal** with funding from the Process Optimization & Modeling for Minerals Sustainability (PrOMMiS) project and the University of Notre Dame.

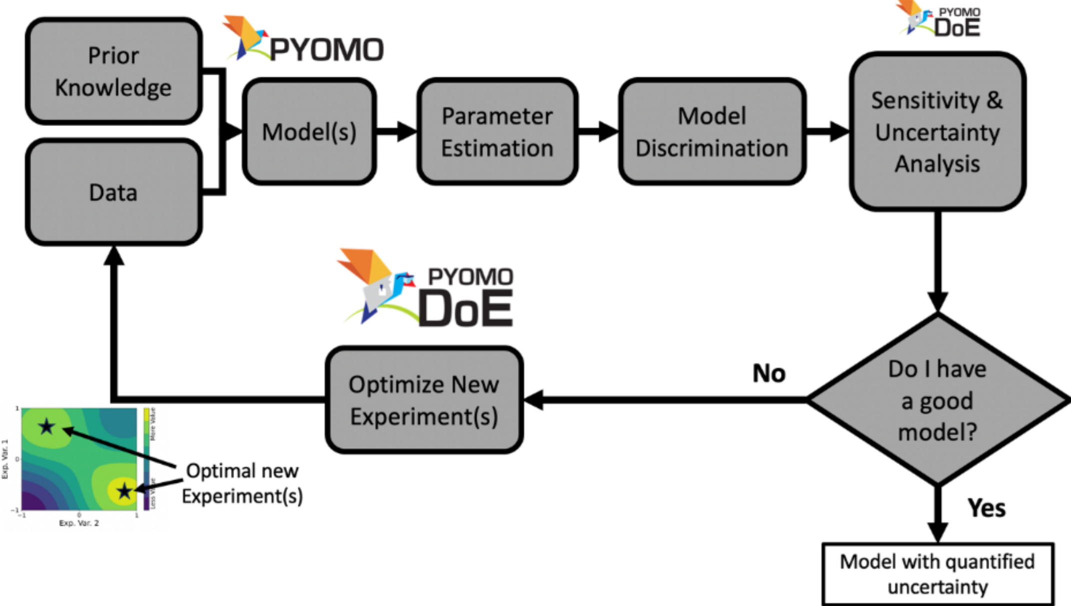
If you use Pyomo.DoE, please cite the [PyomoDOE-paper]:

Wang, Jialu, and Alexander W. Dowling. "Pyomo.DOE: An open-source package for model-based design of experiments in Python", *AIChE Journal*, 68(12), e17813. 2022. DOI 10.1002/aic.17813

Index of Pyomo.DoE documentation

Overview

Model-based Design of Experiments (MBDoE) is a technique to maximize the information gain from experiments by directly using science-based models with physically meaningful parameters. It is one key component in the model calibration and uncertainty quantification workflow shown below:



The parameter estimation, uncertainty analysis, and MBDoE are combined into an iterative framework to select, refine, and calibrate science-based mathematical models with quantified uncertainty. Currently, Pyomo.DoE focuses on increasing parameter precision.

Pyomo.DoE provides the exploratory analysis and MBDoE capabilities to the Pyomo ecosystem. The user provides one Pyomo model, a set of parameter nominal values, the allowable design spaces for design variables, and the postulated observation error model. During exploratory analysis, Pyomo.DoE checks experimental design conditions that will provide the most informative experimental outputs (i.e., data) to estimate the model parameters. Parameter estimation packages such as *Parmest* can perform parameter estimation using the available data to infer values for parameters, and facilitate an uncertainty analysis to approximate the parameter covariance matrix. If the parameter uncertainties are sufficiently small (i.e., at a level acceptable to the user), the workflow terminates and returns the final model with quantified parametric uncertainty. If not, MBDoE recommends optimized experimental conditions to generate new

data that will maximize information gain and eventually reduce parameter uncertainty.

Below is an overview of the type of optimization models Pyomo.DoE can accommodate:

- Pyomo.DoE is suitable for optimization models of **continuous** variables
- Pyomo.DoE can handle **equality constraints** defining state variables
- Pyomo.DoE supports (Partial) Differential-Algebraic Equations (PDAE) models via *Pyomo.DAE*
- Pyomo.DoE also supports models with only algebraic equations

The general form of a DAE problem that can be passed into Pyomo.DoE is shown below:

$$\begin{array}{l} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{y}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \boldsymbol{\theta}) \\ \mathbf{g}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{y}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \boldsymbol{\theta}) = \mathbf{0} \\ \mathbf{y} = \mathbf{h}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \boldsymbol{\theta}) \\ \mathbf{f}^{\mathbf{0}}\left(\dot{\mathbf{x}}\left(t_0\right), \mathbf{x}\left(t_0\right), \mathbf{z}(t_0), \mathbf{y}(t_0), \mathbf{u}\left(t_0\right), \overline{\mathbf{w}}, \boldsymbol{\theta}\right) = \mathbf{0} \\ \mathbf{g}^{\mathbf{0}}\left(\mathbf{x}\left(t_0\right), \mathbf{z}(t_0), \mathbf{y}(t_0), \mathbf{u}\left(t_0\right), \overline{\mathbf{w}}, \boldsymbol{\theta}\right) = \mathbf{0} \\ \mathbf{y}^{\mathbf{0}}\left(t_0\right) = \mathbf{h}\left(\mathbf{x}\left(t_0\right), \mathbf{z}(t_0), \mathbf{u}\left(t_0\right), \overline{\mathbf{w}}, \boldsymbol{\theta}\right) \end{array}$$

where:

- $\boldsymbol{\theta} \in \mathbb{R}^{N_p}$ are unknown model parameters.
- $\mathbf{x} \subseteq \mathcal{X}$ are dynamic state variables which characterize trajectory of the system, $\mathcal{X} \in \mathbb{R}^{N_x \times N_t}$.
- $\mathbf{z} \subseteq \mathcal{Z}$ are algebraic state variables, $\mathcal{Z} \in \mathbb{R}^{N_z \times N_t}$.
- $\mathbf{u} \subseteq \mathcal{U}$ are time-varying decision variables, $\mathcal{U} \in \mathbb{R}^{N_u \times N_t}$.
- $\overline{\mathbf{w}} \in \mathbb{R}^{N_w}$ are time-invariant decision variables.
- $\mathbf{y} \subseteq \mathcal{Y}$ are measured variables (i.e., experimental outputs), $\mathcal{Y} \in \mathbb{R}^{N_r \times N_t}$.
- $\mathbf{f}(\cdot)$ are differential equations.
- $\mathbf{g}(\cdot)$ are algebraic equations.
- $\mathbf{h}(\cdot)$ are measurement functions.
- $\mathbf{t} \in \mathbb{R}^{N_t \times 1}$ is a union of all time sets.

Note

Parameters and design variables should be defined as Pyomo `Var` components when building the model using the `Experiment` class so that users can use both `Parmest` and `Pyomo.DoE` seamlessly.

Based on the above notation, the form of the MBDoE problem addressed in Pyomo.DoE is shown below:

$$
\begin{aligned}
\max_{\boldsymbol{\varphi}} \quad & \Psi(\mathbf{M}(\hat{\boldsymbol{\theta}}, \boldsymbol{\varphi})) \\
\text{s.t.} \quad & \mathbf{M}(\hat{\boldsymbol{\theta}}, \boldsymbol{\varphi}) = \sum_{r}^{N_r} \sum_{r'}^{N_r} \tilde{\sigma}_{(r,r')} \mathbf{Q}_r^{\mathbf{T}} \mathbf{Q}_{r'} + \mathbf{M}_0 \\
& \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{y}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}) \\
& \mathbf{g}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{y}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}) = \mathbf{0} \\
& \mathbf{y} = \mathbf{h}(\mathbf{x}(t), \mathbf{z}(t), \mathbf{u}(t), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}) \\
& \mathbf{f^0}\left(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{z}(t_0), \mathbf{y}(t_0), \mathbf{u}(t_0), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}\right) = \mathbf{0} \\
& \mathbf{g^0}\left(\mathbf{x}(t_0), \mathbf{z}(t_0), \mathbf{y}(t_0), \mathbf{u}(t_0), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}\right) = \mathbf{0} \\
& \mathbf{y^0}(t_0) = \mathbf{h}\left(\mathbf{x}(t_0), \mathbf{z}(t_0), \mathbf{u}(t_0), \overline{\mathbf{w}}, \hat{\boldsymbol{\theta}}\right)
\end{aligned} \tag{3.1}
$$

where:

- $\boldsymbol{\varphi}$ are design variables, which are manipulated to maximize the information content of experiments. It should consist of one or more of $\mathbf{u}(t), \mathbf{y^0}(t_0), \overline{\mathbf{w}}$. With a proper model formulation, the timepoints for control or measurements $\mathbf{t}$ can also be degrees of freedom.
- $\mathbf{M}$ is the Fisher information matrix (FIM), approximated as the inverse of the covariance matrix of parameter estimates $\hat{\boldsymbol{\theta}}$. A large FIM indicates more information contained in the experiment for parameter estimation.
- $\mathbf{Q}$ is the dynamic sensitivity matrix, containing the partial derivatives of $\mathbf{y}$ with respect to $\boldsymbol{\theta}$.
- Ψ is the scalar design criteria to measure the information content in the FIM.
- $\mathbf{M}_0$ is the sum of all the FIMs from previous experiments.

Pyomo.DoE provides five design criteria Ψ to measure the information in the FIM. The covariance matrix of parameter estimates is approximated as the inverse of the FIM, i.e., $\mathbf{V} \approx \mathbf{M}^{-1}$. We can use the FIM or the covariance matrix to define the design criteria.

Table 3.9: Pyomo.DoE design criteria

Design criterion	Computation	Geometrical meaning
A-optimality	$\text{trace}(\mathbf{V}) = \text{trace}(\mathbf{M}^{-1})$	Minimizing this is equivalent to minimizing the enclosing box of the confidence ellipse
Pseudo A-optimality	$\text{trace}(\mathbf{M})$	Maximizing this is equivalent to maximizing the dimensions of the enclosing box of the Fisher Information Matrix
D-optimality	$\det(\mathbf{M}) = 1/\det(\mathbf{V})$	Maximizing this is equivalent to minimizing confidence-ellipsoid volume
E-optimality	$\lambda_{\min}(\mathbf{M}) = 1/\lambda_{\max}(\mathbf{V})$	Maximizing this is equivalent to minimizing the longest axis of the confidence ellipse
Modified E-optimality	$\text{cond}(\mathbf{M}) = \text{cond}(\mathbf{V})$	Minimizing this is equivalent to minimizing the ratio of the longest axis to the shortest axis of the confidence ellipse

Note

A confidence ellipse is a geometric representation of the uncertainty in parameter estimates. It is derived from the covariance matrix $\mathbf{V}$.

In order to solve problems of the above, Pyomo.DoE implements the 2-stage stochastic program. Please see [PyomoDOE-paper] for details.

Quick Start Guide

To use Pyomo.DoE, a user must implement a subclass of the *Parmest* `Experiment` class. The subclass must have a `get_labeled_model` method which returns a Pyomo `ConcreteModel` containing four Pyomo `Suffix` components identifying the parts of the model used in MBDoE analysis. This is in line with the convention used in the parameter estimation tool, *Parmest*. The four Pyomo `Suffix` components are:

- `experiment_inputs` - The experimental design decisions
- `experiment_outputs` - The variables that are being measured
- `measurement_error` - The error associated with the measured value of the experimental outputs. It is passed as a standard deviation or square root of the diagonal elements of the observation (measurement) error covariance matrix. Pyomo.DoE currently assumes that the observation errors are Gaussian and independent both in time and across measurements.
- `unknown_parameters` - Those parameters in the model that are estimated from the experimental outputs

An example of the subclassed `Experiment` object that builds and labels the model is shown in the next few sections.

This guide illustrates the use of Pyomo.DoE using a reaction kinetics example ([PyomoDOE-paper]).

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C \tag{3.2}$$

The Arrhenius equations model the temperature dependence of the reaction rate coefficients k_1 and k_2. Assuming a first-order reaction mechanism gives the reaction rate model shown below. Further, we assume only species A is fed to the reactor.

$$\begin{aligned} k_1 &= A_1 e^{-\frac{E_1}{RT}} \\ k_2 &= A_2 e^{-\frac{E_2}{RT}} \\ \frac{dC_A}{dt} &= -k_1 C_A \\ \frac{dC_B}{dt} &= k_1 C_A - k_2 C_B \\ C_{A0} &= C_A + C_B + C_C \\ C_B(t_0) &= 0 \\ C_C(t_0) &= 0 \end{aligned} \tag{3.3}$$

$C_A(t), C_B(t), C_C(t)$ are the time-varying concentrations of the species A, B, C, respectively. k_1, k_2 are the rate constants for the two chemical reactions using an Arrhenius equation with activation energies E_1, E_2 and pre-exponential factors A_1, A_2. The goal of MBDoE is to optimize the experiment design variables $\boldsymbol{\varphi} = (C_{A0}, T(t))$, where $C_{A0}, T(t)$ are the initial concentration of species A and the time-varying reactor temperature, to maximize the precision of unknown model parameters $\boldsymbol{\theta} = (A_1, E_1, A_2, E_2)$ by measuring $\mathbf{y}(t) = (C_A(t), C_B(t), C_C(t))$. The observation errors are assumed to be independent both in time and across measurements with a constant standard deviation of 1 M for each species.

Step 0: Import Pyomo and the Pyomo.DoE module and create an `Experiment` class

Note

This example uses the data file `result.json`, located in the Pyomo repository at: `pyomo/contrib/doe/examples/result.json`, which contains the nominal parameter values, and measurements for the reaction kinetics experiment.

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar, Simulator
from pyomo.contrib.parmest.experiment import Experiment
```

Subclass the *Parmest* `Experiment` class to define the reaction kinetics experiment and build the Pyomo ConcreteModel.

```
class ReactorExperiment(Experiment):
    def __init__(self, data, nfe, ncp):
        """
        Arguments
        ---------
        data: object containing vital experimental information
        nfe: number of finite elements
        ncp: number of collocation points for the finite elements
        """
        self.data = data
        self.nfe = nfe
        self.ncp = ncp
        self.model = None

        #############################
```

Step 1: Define the Pyomo process model

The process model for the reaction kinetics problem is shown below. Here, we build the model without any data or discretization.

```
    def create_model(self):
        """
        This is an example user model provided to DoE library.
        It is a dynamic problem solved by Pyomo.DAE.

        Return
        ------
        m: a Pyomo.DAE model
        """

        m = self.model = pyo.ConcreteModel()

        # Model parameters
        m.R = pyo.Param(mutable=False, initialize=8.314)

        # Define model variables
        ########################
        # time
        m.t = ContinuousSet(bounds=[0, 1])

        # Concentrations
        m.CA = pyo.Var(m.t, within=pyo.NonNegativeReals)
        m.CB = pyo.Var(m.t, within=pyo.NonNegativeReals)
```

(continues on next page)

(continued from previous page)

```
        m.CC = pyo.Var(m.t, within=pyo.NonNegativeReals)

        # Temperature
        m.T = pyo.Var(m.t, within=pyo.NonNegativeReals)

        # Arrhenius rate law equations
        m.A1 = pyo.Var(within=pyo.NonNegativeReals)
        m.E1 = pyo.Var(within=pyo.NonNegativeReals)
        m.A2 = pyo.Var(within=pyo.NonNegativeReals)
        m.E2 = pyo.Var(within=pyo.NonNegativeReals)

        # Differential variables (Conc.)
        m.dCAdt = DerivativeVar(m.CA, wrt=m.t)
        m.dCBdt = DerivativeVar(m.CB, wrt=m.t)

        ########################
        # End variable def.

        # Equation definition
        ########################

        # Expression for rate constants
        @m.Expression(m.t)
        def k1(m, t):
            return m.A1 * pyo.exp(-m.E1 * 1000 / (m.R * m.T[t]))

        @m.Expression(m.t)
        def k2(m, t):
            return m.A2 * pyo.exp(-m.E2 * 1000 / (m.R * m.T[t]))

        # Concentration odes
        @m.Constraint(m.t)
        def CA_rxn_ode(m, t):
            return m.dCAdt[t] == -m.k1[t] * m.CA[t]

        @m.Constraint(m.t)
        def CB_rxn_ode(m, t):
            return m.dCBdt[t] == m.k1[t] * m.CA[t] - m.k2[t] * m.CB[t]

        # algebraic balance for concentration of C
        # Valid because the reaction system (A --> B --> C) is equimolar
        @m.Constraint(m.t)
        def CC_balance(m, t):
            return m.CA[0] == m.CA[t] + m.CB[t] + m.CC[t]

        ########################
```

Step 2: Finalize the Pyomo process model

Here, we add data to the model and discretize it. This step is required before the model can be labeled.

```
    def finalize_model(self):
        """
        Example finalize model function. There are two main tasks
        here:

            1. Extracting useful information for the model to align
               with the experiment. (Here: CA0, t_final, t_control)
            2. Discretizing the model subject to this information.

        """
        m = self.model

        # Unpacking data before simulation
        control_points = self.data["control_points"]

        # Set initial concentration values for the experiment
        m.CA[0].value = self.data["CA0"]
        m.CB[0].fix(self.data["CB0"])

        # Update model time `t` with time range and control time points
        m.t.update(self.data["t_range"])
        m.t.update(control_points)

        # Fix the unknown parameter values
        m.A1.fix(self.data["A1"])
        m.A2.fix(self.data["A2"])
        m.E1.fix(self.data["E1"])
        m.E2.fix(self.data["E2"])

        # Add upper and lower bounds to the design variable, CA[0]
        m.CA[0].setlb(self.data["CA_bounds"][0])
        m.CA[0].setub(self.data["CA_bounds"][1])

        m.t_control = control_points

        # Discretizing the model
        discr = pyo.TransformationFactory("dae.collocation")
        discr.apply_to(m, nfe=self.nfe, ncp=self.ncp, wrt=m.t)

        # Initializing Temperature in the model
        cv = None
        for t in m.t:
            if t in control_points:
                cv = control_points[t]
                m.T[t].fix()
            m.T[t].setlb(self.data["T_bounds"][0])
            m.T[t].setub(self.data["T_bounds"][1])
            m.T[t] = cv

        # Make a constraint that holds temperature constant between control time points
        @m.Constraint(m.t - control_points)
        def T_control(m, t):
```

(continues on next page)

(continued from previous page)

```
        """
        Piecewise constant temperature between control points
        """
        neighbour_t = max(tc for tc in control_points if tc < t)
        return m.T[t] == m.T[neighbour_t]

    ########################
```

Step 3: Label the information needed for DoE analysis

This step formally labels the Pyomo model with the experimental inputs (design variables), experimental outputs (measurements), measurement errors, and unknown parameters. The labeling of these four important groups is performed using Pyomo `Suffix` components (as discussed earlier) by defining a `label_experiment` method.

```
    def label_experiment(self):
        """
        Example for annotating (labeling) the model with a
        full experiment.
        """
        m = self.model

        # Set measurement labels
        m.experiment_outputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        # Add CA to experiment outputs
        m.experiment_outputs.update((m.CA[t], None) for t in m.t_control)
        # Add CB to experiment outputs
        m.experiment_outputs.update((m.CB[t], None) for t in m.t_control)
        # Add CC to experiment outputs
        m.experiment_outputs.update((m.CC[t], None) for t in m.t_control)

        # Adding error for measurement values (assuming no covariance and constant error␣
↪for all measurements)
        m.measurement_error = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        concentration_error = 1e-2  # Error in concentration measurement
        # Add measurement error for CA
        m.measurement_error.update((m.CA[t], concentration_error) for t in m.t_control)
        # Add measurement error for CB
        m.measurement_error.update((m.CB[t], concentration_error) for t in m.t_control)
        # Add measurement error for CC
        m.measurement_error.update((m.CC[t], concentration_error) for t in m.t_control)

        # Identify design variables (experiment inputs) for the model
        m.experiment_inputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        # Add experimental input label for initial concentration
        m.experiment_inputs[m.CA[m.t.first()]] = None
        # Add experimental input label for Temperature
        m.experiment_inputs.update((m.T[t], None) for t in m.t_control)

        # Add unknown parameter labels
        m.unknown_parameters = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        # Add labels to all unknown parameters with nominal value as the value
```

(continues on next page)

(continued from previous page)

```
        m.unknown_parameters.update((k, pyo.value(k)) for k in [m.A1, m.A2, m.E1, m.E2])

        #########################
```

Step 4: Implement the `get_labeled_model` method

This method utilizes the previous 3 steps and is used by *Pyomo.DoE* to build the model to perform optimal experimental design.

```
    def get_labeled_model(self):
        if self.model is None:
            self.create_model()
            self.finalize_model()
            self.label_experiment()
        return self.model
```

Step 5: Exploratory analysis (Enumeration)

After creating the subclass of the `Experiment` class, exploratory analysis is suggested to systematically enumerate the experimental design space and identify regions that provide high information content about the model parameters, as quantified by the A-, D-, E-, and ME-optimality criteria. Additionally, it helps to initialize the model for the optimal experimental design step.

Pyomo.DoE can perform exploratory sensitivity analysis with the `compute_FIM_full_factorial()` method. The `compute_FIM_full_factorial()` method generates a grid over the design space as specified by the user. Each grid point represents an MBDoE problem solved using the `compute_FIM()` method. In this way, sensitivity of the FIM over the design space can be evaluated. Pyomo.DoE supports plotting the results from the `compute_FIM_full_factorial()` method with the `draw_factorial_figure()` method.

The following code defines the `run_reactor_doe` function. This function encapsulates the workflow for both sensitivity analysis (Step 5) and optimal design (Step 6).

```
from pyomo.common.dependencies import numpy as np, pathlib

from pyomo.contrib.doe.examples.reactor_experiment import ReactorExperiment
from pyomo.contrib.doe import DesignOfExperiments

import pyomo.environ as pyo

import json


# Example for sensitivity analysis on the reactor experiment
# After sensitivity analysis is done, we perform optimal DoE
def run_reactor_doe(
    n_points_for_C0: int = 9,
    n_points_for_T0: int = 9,
    C0_bounds_for_factorial: tuple = (1, 5),
    T0_bounds_for_factorial: tuple = (300, 700),
    compute_FIM_full_factorial=True,
```

(continues on next page)

(continued from previous page)

```
    plot_factorial_results=True,
    figure_file_name="example_reactor_compute_FIM",
    log_scale=False,
    run_optimal_doe=True,
):
    """
    This function demonstrates how to perform sensitivity analysis on the reactor

    Parameters
    ----------
    n_points_for_C0 : int, optional
        number of points to use for the C0 design range, by default 9
    n_points_for_T0 : int, optional
        number of points to use for the T0 design range, by default 9
    compute_FIM_full_factorial : bool, optional
        whether to compute the full factorial design, by default True
    plot_factorial_results : bool, optional
        whether to plot the results of the full factorial design, by default True
    figure_file_name : str, optional
        file name to save the factorial figure, by default "example_reactor_compute_FIM"
    run_optimal_doe : bool, optional
        whether to run the optimal DoE, by default True
    """
    # Read in file
    DATA_DIR = pathlib.Path(__file__).parent
    file_path = DATA_DIR / "result.json"

    with open(file_path) as f:
        data_ex = json.load(f)

    # Put temperature control time points into correct format for reactor experiment
    data_ex["control_points"] = {
        float(k): v for k, v in data_ex["control_points"].items()
    }

    # Create a ReactorExperiment object; data and discretization information are part
    # of the constructor of this object
    experiment = ReactorExperiment(data=data_ex, nfe=10, ncp=3)

    # Use a central difference, with step size 1e-3
    fd_formula = "central"
    step_size = 1e-3

    # Use the determinant objective with scaled sensitivity matrix
    objective_option = "determinant"
    scale_nominal_param_value = True

    # Create the DesignOfExperiments object
    # We will not be passing any prior information in this example
    # and allow the experiment object and the DesignOfExperiments
    # call of ``run_doe`` perform model initialization.
    doe_obj = DesignOfExperiments(
```

(continues on next page)

(continued from previous page)

```
        experiment,
        fd_formula=fd_formula,
        step=step_size,
        objective_option=objective_option,
        scale_constant_value=1,
        scale_nominal_param_value=scale_nominal_param_value,
        prior_FIM=None,
        jac_initial=None,
        fim_initial=None,
        L_diagonal_lower_bound=1e-7,
        solver=None,
        tee=False,
        get_labeled_model_args=None,
        _Cholesky_option=True,
        _only_compute_fim_lower=True,
    )
    if compute_FIM_full_factorial:
        # Make design ranges to compute the full factorial design
        design_ranges = {
            "CA[0]": [*C0_bounds_for_factorial, n_points_for_C0],
            "T[0]": [*T0_bounds_for_factorial, n_points_for_T0],
        }

        # Compute the full factorial design with the sequential FIM calculation
        doe_obj.compute_FIM_full_factorial(
            design_ranges=design_ranges, method="sequential"
        )
    if plot_factorial_results:
        # Plot the results
        doe_obj.draw_factorial_figure(
            sensitivity_design_variables=["CA[0]", "T[0]"],
            fixed_design_variables={
                "T[0.125]": 300,
                "T[0.25]": 300,
                "T[0.375]": 300,
                "T[0.5]": 300,
                "T[0.625]": 300,
                "T[0.75]": 300,
                "T[0.875]": 300,
                "T[1]": 300,
            },
            title_text="Reactor Example",
            xlabel_text="Concentration of A (M)",
            ylabel_text="Initial Temperature (K)",
            figure_file_name=figure_file_name,
            log_scale=log_scale,
        )

    ###########################
    # End sensitivity analysis

    # Begin optimal DoE
```

(continues on next page)

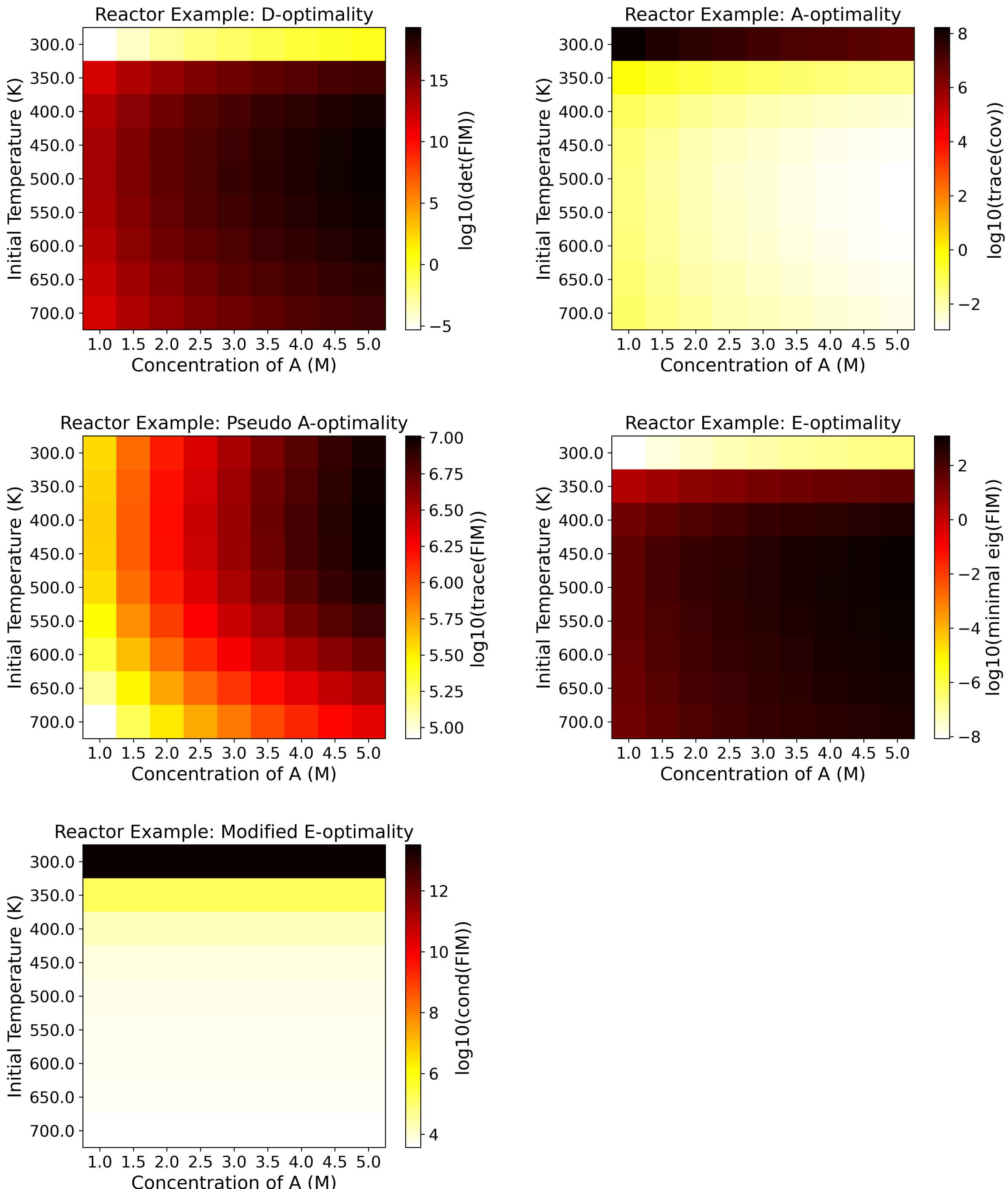
(continued from previous page)

```
    ####################
    if run_optimal_doe:
        doe_obj.run_doe()

        # Print out a results summary
        print("Optimal experiment values: ")
        print(
            "\tInitial concentration: {:.2f}".format(
                doe_obj.results["Experiment Design"][0]
            )
        )
        print(
            ("\tTemperature values: [" + "{:.2f}, " * 8 + "{:.2f}]").format(
                *doe_obj.results["Experiment Design"][1:]
            )
        )
        print("FIM at optimal design:\n {}".format(np.array(doe_obj.results["FIM"])))
        print(
            "Objective value at optimal design: {:.2f}".format(
                pyo.value(doe_obj.model.objective)
            )
        )

        print(doe_obj.results["Experiment Design Names"])

        ###################
        # End optimal DoE

    return doe_obj
```

After defining the function, we will call it to perform the exploratory analysis and the optimal experimental design.

```
run_reactor_doe(figure_file_name=None)
```

A design exploration for the initial concentration and temperature as experimental design variables with 9 values for each, produces the the five figures for five optimality criteria using the `compute_FIM_full_factorial` and `draw_factorial_figure` methods as shown below:



The heatmaps show the variation of a FIM-based objective function (specified by the user) over a grid of the experimental design space. Therefore, the heatmaps are a representation of the experimental information of various design conditions. Horizontal and vertical axes are the two experimental design variables, while the color of each grid shows the experimental information content. For example, the D-optimality (upper left subplot) heatmap figure shows that the most informative region is around $C_{A0} = 5.0$ M, $T = 500.0$ K with a $\log_{10}$ determinant of FIM being around 19, while the least informative region is around $C_{A0} = 1.0$ M, $T = 300.0$ K, with a $\log_{10}$ determinant of FIM being around -5. For D-, Pseudo A-, and E-optimality we want to maximize the objective function, while for A- and Modified E-optimality we want to minimize the objective function.

In this sensitivity analysis plot (heatmap), we only varied the initial concentration and the initial temperature, while the

temperature at other time points is fixed at 300 K.

$$T(t) = \begin{cases} T_0, & t \leq 0.125 \\ 300 \text{ K}, & t > 0.125 \end{cases}$$

If $T_0 = 300$ K, the reaction is conducted under strictly isothermal conditions. Because the temperature is constant, the sensitivities of the species concentrations with respect to the Arrhenius parameters (A_i and E_i) become linearly dependent. This high correlation means the effects of the pre-exponential factor and the activation energy cannot be uniquely distinguished from the measurements. Consequently, the Fisher Information Matrix (FIM) becomes ill-conditioned, resulting in a near-zero determinant and a very large condition number.

To break this correlation and make the parameters identifiable, introducing a time- varying temperature profile (for example, a temperature step or a ramp) is required. As shown in the heatmap, when the initial temperature T_0 differs from the subsequent 300 K baseline, such a temperature change breaks the linear dependence, yielding a well-conditioned FIM and identifiable parameters.

Step 6: Performing an optimal experimental design

In Step 5, we defined the `run_reactor_doe` function. This function constructs the DoE object and performs the exploratory sensitivity analysis. The way the function is defined, it also proceeds immediately to the optimal experimental design step (applying `run_doe` on the `DesignOfExperiments` object). We can initialize the model with the result we obtained from the exploratory analysis (optimal point from the heatmaps) to help the optimal design step to speed up convergence. However, implementation of this initialization is not shown here.

After applying `run_doe` on the `DesignOfExperiments` object, the optimal design is an initial concentration of 5.0 mol/L and an initial temperature of 494 K with all other temperatures being 300 K. The corresponding $\log_{10}$ determinant of the FIM is 19.32.

Experiment Abstraction

Note

Detailed descriptions and example code for experiment abstraction in Pyomo.DoE will be added in a future update.

Objective Options

Note

Detailed descriptions and example code for the objective options in Pyomo.DoE will be added in a future update.

Multiple Experiments

Note

Detailed descriptions and example code for simultaneous design of multiple experiments will be added in a future update.

Parameter Uncertainty

Note

Detailed descriptions and example code for experiment design under parameter uncertainty will be added in a future update.

3.4.4 Infeasibility Diagnostics

There are two closely related tools for infeasibility diagnosis:

- *Infeasible Irreducible System (IIS) Tool*
- *Minimal Intractable System finder (MIS) Tool*

The first simply provides a conduit for solvers that compute an infeasible irreducible system (e.g., Cplex, Gurobi, or Xpress). The second provides similar functionality, but uses the `mis` package contributed to Pyomo.

Infeasible Irreducible System (IIS) Tool

This module contains functions for computing an irreducible infeasible set for a Pyomo MILP or LP using a specified commercial solver, one of CPLEX, Gurobi, or Xpress.

`pyomo.contrib.iis.`**`write_iis`**(*pyomo_model*, *iis_file_name*, *solver=None*, *logger=<Logger pyomo.contrib.iis (INFO)>*)

Write an irreducible infeasible set for a Pyomo MILP or LP using the specified commercial solver.

Parameters

- **`pyomo_model`** – A Pyomo Block or ConcreteModel
- **`iis_file_name`** (*`str`*) – A file name to write the IIS to, e.g., infeasible_model.ilp
- **`solver`** (*`str`*) – Specify the solver to use, one of "cplex", "gurobi", or "xpress". If None, the tool will use the first solver available.
- **`logger`** (*`logging.Logger`*) – A logger for messages. Uses pyomo.contrib.iis logger by default.

Returns

iis_file_name – The file containing the IIS.

Return type

str

Minimal Intractable System finder (MIS) Tool

The file `mis.py` finds sets of actions that each, independently, would result in feasibility. The zero-tolerance is whatever the solver uses, so users may want to post-process output if it is going to be used for analysis. It also computes a minimal intractable system (which is not guaranteed to be unique). It was written by Ben Knueven as part of the watertap project (https://github.com/watertap-org/watertap) and is therefore governed by a license shown at the top of `mis.py`.

The algorithms come from John Chinneck's slides, see: http://www.sce.carleton.ca/faculty/chinneck/docs/CPAIOR07InfeasibilityTutorial.pdf

Solver

At the time of this writing, you need to use IPopt even for LPs.

Quick Start

The file `trivial_mis.py` is a tiny example listed at the bottom of this help file, which references a Pyomo model with the Python variable *m* and has these lines:

```
from pyomo.contrib.mis import compute_infeasibility_explanation
ipopt = pyo.SolverFactory("ipopt")
compute_infeasibility_explanation(m, solver=ipopt)
```

Note

This is done instead of solving the problem.

Note

IDAES users can pass `get_solver()` imported from `ideas.core.solvers` as the solver.

Interpreting the Output

Assuming the dependencies are installed, running `trivial_mis.py` (shown below) will produce a lot of warnings from IPopt and then meaningful output (using a logger).

Repair Options

This output for the trivial example shows three independent ways that the model could be rendered feasible:

```
Model Trivial Quad may be infeasible. A feasible solution was found with only the␣
↪following variable bounds relaxed:
     ub of var x[1] by 4.464126126706818e-05
     lb of var x[2] by 0.9999553410114216
Another feasible solution was found with only the following variable bounds relaxed:
     lb of var x[1] by 0.7071067726864677
     ub of var x[2] by 0.41421355687130673
     ub of var y by 0.7071067651855212
Another feasible solution was found with only the following inequality constraints,␣
↪equality constraints, and/or variable bounds relaxed:
     constraint: c by 0.9999999861866736
```

Minimal Intractable System (MIS)

This output shows a minimal intractable system:

```
Computed Minimal Intractable System (MIS)!
Constraints / bounds in MIS:
     lb of var x[2]
     lb of var x[1]
     constraint: c
```

Constraints / bounds in guards for stability

This part of the report is for nonlinear programs (NLPs).

When we're trying to reduce the constraint set, for an NLP there may be constraints that when missing cause the solver to fail in some catastrophic fashion. In this implementation this is interpreted as failing to get a *results* object back from the call to *solve*. In these cases we keep the constraint in the problem but it's in the set of "guard" constraints – we can't really be sure they're a source of infeasibility or not, just that "bad things" happen when they're not included.

Perhaps ideally we would put a constraint in the "guard" set if IPopt failed to converge, and only put it in the MIS if IPopt converged to a point of local infeasibility. However, right now the code generally makes the assumption that if IPopt fails to converge the subproblem is infeasible, though obviously that is far from the truth. Hence for difficult NLPs even the "Phase 1" may "fail" – in that when finished the subproblem containing just the constraints in the elastic filter may be feasible – because IPopt failed to converge and we assumed that meant the subproblem was not feasible.

Dealing with NLPs is far from clean, but that doesn't mean the tool can't return useful results even when its assumptions are not satisfied.

trivial_mis.py

```
import pyomo.environ as pyo
m = pyo.ConcreteModel("Trivial Quad")
m.x = pyo.Var([1,2], bounds=(0,1))
m.y = pyo.Var(bounds=(0, 1))
m.c = pyo.Constraint(expr=m.x[1] * m.x[2] == -1)
m.d = pyo.Constraint(expr=m.x[1] + m.y >= 1)

from pyomo.contrib.mis import compute_infeasibility_explanation
ipopt = pyo.SolverFactory("ipopt")
compute_infeasibility_explanation(m, solver=ipopt)
```

3.4.5 Incidence Analysis

Tools for constructing and analyzing the incidence graph of variables and constraints.

This documentation contains the following resources:

Overview

What is Incidence Analysis?

A Pyomo extension for constructing the bipartite incidence graph of variables and constraints, and an interface to useful algorithms for analyzing or decomposing this graph.

Why is Incidence Analysis useful?

It can identify the source of certain types of singularities in a system of variables and constraints. These singularities often violate assumptions made while modeling a physical system or assumptions required for an optimization solver to guarantee convergence. In particular, interior point methods used for nonlinear local optimization require the Jacobian of equality constraints (and active inequalities) to be full row rank, and this package implements the Dulmage-Mendelsohn partition, which can be used to determine if this Jacobian is structurally rank-deficient.

Who develops and maintains Incidence Analysis?

This extension was developed by Robert Parker while a PhD student in Professor Biegler's lab at Carnegie Mellon University, with guidance from Bethany Nicholson and John Siirola at Sandia.

How can I cite Incidence Analysis?

If you use Incidence Analysis in your research, we would appreciate you citing the following paper:

```
@article{parker2023dulmage,
title = {Applications of the {Dulmage-Mendelsohn} decomposition for debugging nonlinear␣
↪optimization problems},
journal = {Computers \& Chemical Engineering},
volume = {178},
pages = {108383},
year = {2023},
issn = {0098-1354},
doi = {https://doi.org/10.1016/j.compchemeng.2023.108383},
url = {https://www.sciencedirect.com/science/article/pii/S0098135423002533},
author = {Robert B. Parker and Bethany L. Nicholson and John D. Siirola and Lorenz T.␣
↪Biegler},
}
```

Incidence Analysis Tutorial

This tutorial walks through examples of the most common use cases for Incidence Analysis:

Debugging a structural singularity with the Dulmage-Mendelsohn partition

We start with some imports and by creating a Pyomo model we would like to debug. Usually the model is much larger and more complicated than this. This particular system appeared when debugging a dynamic 1-D partial differential-algebraic equation (PDAE) model representing a chemical looping combustion reactor.

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface

>>> m = pyo.ConcreteModel()
>>> m.components = pyo.Set(initialize=[1, 2, 3])
>>> m.x = pyo.Var(m.components, initialize=1.0/3.0)
>>> m.flow_comp = pyo.Var(m.components, initialize=10.0)
>>> m.flow = pyo.Var(initialize=30.0)
>>> m.density = pyo.Var(initialize=1.0)
>>> m.sum_eqn = pyo.Constraint(
...     expr=sum(m.x[j] for j in m.components) - 1 == 0
... )
>>> m.holdup_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.density - 1 == 0 for j in m.components
... })
>>> m.density_eqn = pyo.Constraint(
...     expr=1/m.density - sum(1/m.x[j] for j in m.components) == 0
... )
>>> m.flow_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.flow - m.flow_comp[j] == 0 for j in m.components
... })
```

To check this model for structural singularity, we apply the Dulmage-Mendelsohn partition. `var_dm_partition` and `con_dm_partition` are named tuples with fields for each of the four subsets defined by the partition: `unmatched`, `overconstrained`, `square`, and `underconstrained`.

```
>>> igraph = IncidenceGraphInterface(m)
>>> # Make sure we have a square system
>>> print(len(igraph.variables))
8
>>> print(len(igraph.constraints))
8
>>> var_dm_partition, con_dm_partition = igraph.dulmage_mendelsohn()
```

If any variables or constraints are unmatched, the (Jacobian of the) model is structurally singular.

```
>>> # Note that the unmatched variables/constraints are not mathematically
>>> # unique and could change with implementation!
>>> for var in var_dm_partition.unmatched:
...     print(var.name)
flow_comp[1]
>>> for con in con_dm_partition.unmatched:
...     print(con.name)
density_eqn
```

This model has one unmatched constraint and one unmatched variable, so it is structurally singular. However, the unmatched variable and constraint are not unique. For example, `flow_comp[2]` could have been unmatched instead of `flow_comp[1]`. The exact variables and constraints that are unmatched depends on both the order in which variables are identified in Pyomo expressions and the implementation of the matching algorithm. For a given implementation, however, these variables and constraints should be deterministic.

Unique subsets of variables and constraints that are useful when debugging a structural singularity are the underconstrained and overconstrained subsystems. The variables in the underconstrained subsystem are contained in the `unmatched` and `underconstrained` fields of the `var_dm_partition` named tuple, while the constraints are contained in the `underconstrained` field of the `con_dm_partition` named tuple. The variables in the overconstrained subsystem are contained in the `overconstrained` field of the `var_dm_partition` named tuple, while the constraints are contained in the `overconstrained` and `unmatched` fields of the `con_dm_partition` named tuple.

We now construct the underconstrained and overconstrained subsystems:

```
>>> uc_var = var_dm_partition.unmatched + var_dm_partition.underconstrained
>>> uc_con = con_dm_partition.underconstrained
>>> oc_var = var_dm_partition.overconstrained
>>> oc_con = con_dm_partition.overconstrained + con_dm_partition.unmatched
```

And display the variables and constraints contained in each:

```
>>> # Note that while these variables/constraints are uniquely determined,
>>> # their order is not!

>>> # Overconstrained subsystem
>>> for var in oc_var:
>>>     print(var.name)
x[1]
density
x[2]
x[3]
```

(continues on next page)

(continued from previous page)

```
>>> for con in oc_con:
>>>     print(con.name)
sum_eqn
holdup_eqn[1]
holdup_eqn[2]
holdup_eqn[3]
density_eqn

>>> # Underconstrained subsystem
>>> for var in uc_var:
>>>     print(var.name)
flow_comp[1]
flow
flow_comp[2]
flow_comp[3]
>>> for con in uc_con:
>>>     print(con.name)
flow_eqn[1]
flow_eqn[2]
flow_eqn[3]
```

At this point we must use our intuition about the system being modeled to identify "what is causing" the singularity. Looking at the under and over- constrained systems, it appears that we are missing an equation to calculate `flow`, the total flow rate, and that `density` is over-specified as it is computed by both the bulk density equation and one of the component density equations.

With this knowledge, we can eventually figure out (a) that we need an equation to calculate `flow` from density and (b) that our "bulk density equation" is actually a *skeletal* density equation. Admittedly, this is difficult to figure out without the full context behind this particular system.

The following code constructs a new version of the model and verifies that it is structurally nonsingular:

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface

>>> m = pyo.ConcreteModel()
>>> m.components = pyo.Set(initialize=[1, 2, 3])
>>> m.x = pyo.Var(m.components, initialize=1.0/3.0)
>>> m.flow_comp = pyo.Var(m.components, initialize=10.0)
>>> m.flow = pyo.Var(initialize=30.0)
>>> m.dens_bulk = pyo.Var(initialize=1.0)
>>> m.dens_skel = pyo.Var(initialize=1.0)
>>> m.porosity = pyo.Var(initialize=0.25)
>>> m.velocity = pyo.Param(initialize=1.0)
>>> m.sum_eqn = pyo.Constraint(
...     expr=sum(m.x[j] for j in m.components) - 1 == 0
... )
>>> m.holdup_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.dens_bulk - 1 == 0 for j in m.components
... })
>>> m.dens_skel_eqn = pyo.Constraint(
...     expr=1/m.dens_skel - sum(1/m.x[j] for j in m.components) == 0
... )
```

(continues on next page)

(continued from previous page)

```
>>> m.dens_bulk_eqn = pyo.Constraint(
...     expr=m.dens_bulk == (1 - m.porosity)*m.dens_skel
... )
>>> m.flow_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.flow - m.flow_comp[j] == 0 for j in m.components
... })
>>> m.flow_dens_eqn = pyo.Constraint(
...     expr=m.flow == m.velocity*m.dens_bulk
... )

>>> igraph = IncidenceGraphInterface(m, include_inequality=False)
>>> print(len(igraph.variables))
10
>>> print(len(igraph.constraints))
10
>>> var_dm_partition, con_dm_partition = igraph.dulmage_mendelsohn()

>>> # There are now no unmatched variables or equations
>>> print(len(var_dm_partition.unmatched))
0
>>> print(len(con_dm_partition.unmatched))
0
```

Debugging a numeric singularity using block triangularization

We start with some imports. To debug a *numeric* singularity, we will need PyomoNLP from *PyNumero* to get the constraint Jacobian, and will need NumPy to compute condition numbers.

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.pynumero.interfaces.pyomo_nlp import PyomoNLP
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface
>>> import numpy as np
```

We now build the model we would like to debug. Compared to the model in *Debugging a structural singularity with the Dulmage-Mendelsohn partition*, we have converted the sum equation to use a sum over component flow rates rather than a sum over mass fractions.

```
>>> m = pyo.ConcreteModel()
>>> m.components = pyo.Set(initialize=[1, 2, 3])
>>> m.x = pyo.Var(m.components, initialize=1.0/3.0)
>>> m.flow_comp = pyo.Var(m.components, initialize=10.0)
>>> m.flow = pyo.Var(initialize=30.0)
>>> m.density = pyo.Var(initialize=1.0)
>>> # This equation is new!
>>> m.sum_flow_eqn = pyo.Constraint(
...     expr=sum(m.flow_comp[j] for j in m.components) == m.flow
... )
>>> m.holdup_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.density - 1 == 0 for j in m.components
... })
>>> m.density_eqn = pyo.Constraint(
...     expr=1/m.density - sum(1/m.x[j] for j in m.components) == 0
```

(continues on next page)

(continued from previous page)

```
... )
>>> m.flow_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.flow - m.flow_comp[j] == 0 for j in m.components
... })
```

We now construct the incidence graph and check unmatched variables and constraints to validate structural nonsingularity.

```
>>> igraph = IncidenceGraphInterface(m, include_inequality=False)
>>> var_dmp, con_dmp = igraph.dulmage_mendelsohn()
>>> print(len(var_dmp.unmatched))
0
>>> print(len(con_dmp.unmatched))
0
```

Our system is structurally nonsingular. Now we check whether we are numerically nonsingular (well-conditioned) by checking the condition number. Admittedly, deciding if a matrix is "singular" by looking at its condition number is somewhat of an art. We might define "numerically singular" as having a condition number greater than the inverse of machine precision (approximately `1e16`), but poorly conditioned matrices can cause problems even if they don't meet this definition. Here we use `1e10` as a somewhat arbitrary condition number threshold to indicate a problem in our system.

```
>>> # PyomoNLP requires exactly one objective function
>>> m._obj = pyo.Objective(expr=0.0)
>>> nlp = PyomoNLP(m)
>>> cond_threshold = 1e10
>>> cond = np.linalg.cond(nlp.evaluate_jacobian_eq().toarray())
>>> print(cond > cond_threshold)
True
```

The system is poorly conditioned. Now we can check diagonal blocks of a block triangularization to determine which blocks are causing the poor conditioning.

```
>>> var_blocks, con_blocks = igraph.block_triangularize()
>>> for i, (vblock, cblock) in enumerate(zip(var_blocks, con_blocks)):
...     submatrix = nlp.extract_submatrix_jacobian(vblock, cblock)
...     cond = np.linalg.cond(submatrix.toarray())
...     print(f"block {i}: {cond}")
...     if cond > cond_threshold:
...         for var in vblock:
...             print(f"  {var.name}")
...         for con in cblock:
...             print(f"  {con.name}")
block 0: 24.492504515710433
block 1: 1.2480741394486336e+17
  flow
  flow_comp[1]
  flow_comp[2]
  flow_comp[3]
  sum_flow_eqn
  flow_eqn[1]
  flow_eqn[2]
  flow_eqn[3]
```

We see that the second block is causing the singularity, and that this block contains the sum equation that we modified for this example. This suggests that converting this equation to sum over flow rates rather than mass fractions just converted a structural singularity to a numeric singularity, and didn't really solve our problem. To see a fix that *does* resolve the singularity, see *Debugging a structural singularity with the Dulmage-Mendelsohn partition*.

Solving a square system with a block triangular decomposition

We start with imports. The key function from Incidence Analysis we will use is `solve_strongly_connected_components`.

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import (
...     solve_strongly_connected_components
... )
```

Now we construct the model we would like to solve. This is a model with the same structure as the "fixed model" in *Debugging a structural singularity with the Dulmage-Mendelsohn partition*.

```
>>> m = pyo.ConcreteModel()
>>> m.components = pyo.Set(initialize=[1, 2, 3])
>>> m.x = pyo.Var(m.components, initialize=1.0/3.0)
>>> m.flow_comp = pyo.Var(m.components, initialize=10.0)
>>> m.flow = pyo.Var(initialize=30.0)
>>> m.dens_bulk = pyo.Var(initialize=1.0)
>>> m.dens_skel = pyo.Var(initialize=1.0)
>>> m.porosity = pyo.Var(initialize=0.25)
>>> m.velocity = pyo.Param(initialize=1.0)
>>> m.holdup = pyo.Param(
...     m.components, initialize={j: 1.0+j/10.0 for j in m.components}
... )
>>> m.sum_eqn = pyo.Constraint(
...     expr=sum(m.x[j] for j in m.components) - 1 == 0
... )
>>> m.holdup_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.dens_bulk - m.holdup[j] == 0 for j in m.components
... })
>>> m.dens_skel_eqn = pyo.Constraint(
...     expr=1/m.dens_skel - sum(1e-3/m.x[j] for j in m.components) == 0
... )
>>> m.dens_bulk_eqn = pyo.Constraint(
...     expr=m.dens_bulk == (1 - m.porosity)*m.dens_skel
... )
>>> m.flow_eqn = pyo.Constraint(m.components, expr={
...     j: m.x[j]*m.flow - m.flow_comp[j] == 0 for j in m.components
... })
>>> m.flow_dens_eqn = pyo.Constraint(
...     expr=m.flow == m.velocity*m.dens_bulk
... )
```

Solving via a block triangular decomposition is useful in cases where the full model does not converge when considered simultaneously by a Newton solver. In this case, we specify a solver to use for the diagonal blocks and call `solve_strongly_connected_components`.

```
>>> # Suppose a solve like this does not converge
>>> # pyo.SolverFactory("scipy.fsolve").solve(m)

>>> # We solve via block-triangular decomposition
>>> solver = pyo.SolverFactory("scipy.fsolve")
>>> res_list = solve_strongly_connected_components(m, solver=solver)
```

We can now display the variable values at the solution:

```
for var in m.component_objects(pyo.Var):
    var.pprint()
```

API Reference

Incident Variables

Functionality for identifying variables that participate in expressions

`pyomo.contrib.incidence_analysis.incidence.`**`get_incident_variables`**(*expr*, ***kwds*)

Get variables that participate in an expression

The exact variables returned depends on the method used to determine incidence. For example, `method=IncidenceMethod.identify_variables` will return all variables participating in the expression, while `method=IncidenceMethod.standard_repn` will return only the variables identified by `generate_standard_repn` which ignores variables that only appear multiplied by a constant factor of zero.

Keyword arguments must be valid options for `IncidenceConfig`.

Parameters

expr (`NumericExpression`) – Expression to search for variables

Returns

List containing the variables that participate in the expression

Return type

list of *VarData*

Example

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import get_incident_variables
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var([1, 2, 3])
>>> expr = m.x[1] + 2*m.x[2] + 3*m.x[3]**2
>>> print([v.name for v in get_incident_variables(expr)])
['x[1]', 'x[2]', 'x[3]']
>>> print([v.name for v in get_incident_variables(expr, linear_only=True)])
['x[1]', 'x[2]']
```

Incidence Options

Configuration options for incidence graph generation

enum `pyomo.contrib.incidence_analysis.config.`**`IncidenceMethod`**(*value*)

Methods for identifying variables that participate in expressions

identify_variables = 0
Use pyomo.core.expr.visitor.identify_variables

standard_repn = 1
Use pyomo.repn.standard_repn.generate_standard_repn

standard_repn_compute_values = 2
Use pyomo.repn.standard_repn.generate_standard_repn with compute_values=True

ampl_repn = 3
Use pyomo.repn.ampl.AMPLRepnVisitor

enum pyomo.contrib.incidence_analysis.config.**IncidenceOrder**(*value*)

pyomo.contrib.incidence_analysis.config.**get_config_from_kwds**(***kwds*)
Get an instance of IncidenceConfig from provided keyword arguments.

If the method argument is IncidenceMethod.ampl_repn and no AMPLRepnVisitor has been provided, a new AMPLRepnVisitor is constructed. This function should generally be used by callers such as IncidenceGraphInterface to ensure that a visitor is created then re-used when calling get_incident_variables in a loop.

pyomo.contrib.incidence_analysis.config.**IncidenceConfig = <pyomo.common.config.ConfigDict object>**
Options for incidence graph generation

- include_fixed – Flag indicating whether fixed variables should be included in the incidence graph
- linear_only – Flag indicating whether only variables that participate linearly should be included.
- method – Method used to identify incident variables. Must be a value of the IncidenceMethod enum.
- _ampl_repn_visitor – Expression visitor used to generate AMPLRepn of each constraint. Must be an instance of AMPLRepnVisitor. *This option is constructed automatically when needed and should not be set by users!*

Pyomo Interfaces

Utility functions and a utility class for interfacing Pyomo components with useful graph algorithms.

class pyomo.contrib.incidence_analysis.interface.**IncidenceGraphInterface**(*model=None*, *active=True*, *include_inequality=True*, ***kwds*)
An interface for applying graph algorithms to Pyomo variables and constraints

Parameters

- **model** (Pyomo BlockData or PyNumero PyomoNLP, default None) – An object from which an incidence graph will be constructed.
- **active** (Bool, default True) – Whether only active constraints should be included in the incidence graph. Cannot be set to False if the model is provided as a PyomoNLP.
- **include_fixed** (Bool, default False) – Whether to include fixed variables in the incidence graph. Cannot be set to False if model is a PyomoNLP.
- **include_inequality** (Bool, default True) – Whether to include inequality constraints (those whose expressions are not instances of EqualityExpression) in the incidence graph. If a PyomoNLP is provided, setting to False uses the evaluate_jacobian_eq method instead of evaluate_jacobian rather than checking constraint expression types.

add_edge(*variable*, *constraint*)

Adds an edge between variable and constraint in the incidence graph

Parameters

- **variable** (VarData) – A variable in the graph
- **constraint** (ConstraintData) – A constraint in the graph

block_triangularize(*variables=None*, *constraints=None*)

Compute an ordered partition of the provided variables and constraints such that their incidence matrix is block lower triangular

Subsets in the partition correspond to the strongly connected components of the bipartite incidence graph, projected with respect to a perfect matching.

Returns

- **var_partition** (*list of lists*) – Partition of variables. The inner lists hold unindexed variables.
- **con_partition** (*list of lists*) – Partition of constraints. The inner lists hold unindexed constraints.

Example

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var([1, 2])
>>> m.eq1 = pyo.Constraint(expr=m.x[1]**2 == 7)
>>> m.eq2 = pyo.Constraint(expr=m.x[1]*m.x[2] == 3)
>>> igraph = IncidenceGraphInterface(m)
>>> vblocks, cblocks = igraph.block_triangularize()
>>> print([[v.name for v in vb] for vb in vblocks])
[['x[1]'], ['x[2]']]
>>> print([[c.name for c in cb] for cb in cblocks])
[['eq1'], ['eq2']]
```

Note

Breaking change in Pyomo 6.5.0

The pre-6.5.0 `block_triangularize` method returned maps from each variable or constraint to the index of its block in a block lower triangularization as the original intent of this function was to identify when variables do or don't share a diagonal block in this partition. Since then, the dominant use case of `block_triangularize` has been to partition variables and constraints into these blocks and inspect or solve each block individually. A natural return type for this functionality is the ordered partition of variables and constraints, as lists of lists. This functionality was previously available via the `get_diagonal_blocks` method, which was confusing as it did not capture that the partition was the diagonal of a block *triangularization* (as opposed to diagonalization). The pre-6.5.0 functionality of `block_triangularize` is still available via the `map_nodes_to_block_triangular_indices` method.

dulmage_mendelsohn(*variables=None*, *constraints=None*)

Partition variables and constraints according to the Dulmage- Mendelsohn characterization of the incidence graph

Variables are partitioned into the following subsets:

- **unmatched** - Variables not matched in a particular maximum cardinality matching
- **underconstrained** - Variables that *could possibly be* unmatched in a maximum cardinality matching
- **square** - Variables in the well-constrained subsystem
- **overconstrained** - Variables matched with constraints that can possibly be unmatched

Constraints are partitioned into the following subsets:

- **underconstrained** - Constraints matched with variables that can possibly be unmatched
- **square** - Constraints in the well-constrained subsystem
- **overconstrained** - Constraints that *can possibly be* unmatched with a maximum cardinality matching
- **unmatched** - Constraints that were not matched in a particular maximum cardinality matching

While the Dulmage-Mendelsohn decomposition does not specify an order within any of these subsets, the order returned by this function preserves the maximum matching that is used to compute the decomposition. That is, zipping "corresponding" variable and constraint subsets yields pairs in this maximum matching. For example:

```
>>> igraph = IncidenceGraphInterface(model)
>>> var_dmpartition, con_dmpartition = igraph.dulmage_mendelsohn()
>>> vdmp = var_dmpartition
>>> cdmp = con_dmpartition
>>> matching = list(zip(
...     vdmp.underconstrained + vdmp.square + vdmp.overconstrained,
...     cdmp.underconstrained + cdmp.square + cdmp.overconstrained,
... ))
>>> # matching is a valid maximum matching of variables and constraints!
```

Returns

- **var_partition** (`ColPartition` named tuple) – Partitions variables into square, underconstrained, overconstrained, and unmatched.
- **con_partition** (`RowPartition` named tuple) – Partitions constraints into square, underconstrained, overconstrained, and unmatched.

Example

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var([1, 2])
>>> m.eq1 = pyo.Constraint(expr=m.x[1]**2 == 7)
>>> m.eq2 = pyo.Constraint(expr=m.x[1]*m.x[2] == 3)
>>> m.eq3 = pyo.Constraint(expr=m.x[1] + 2*m.x[2] == 5)
>>> igraph = IncidenceGraphInterface(m)
>>> var_dmp, con_dmp = igraph.dulmage_mendelsohn()
>>> print([v.name for v in var_dmp.overconstrained])
['x[1]', 'x[2]']
>>> print([c.name for c in con_dmp.overconstrained])
['eq1', 'eq2']
```

(continues on next page)

(continued from previous page)

```
>>> print([c.name for c in con_dmp.unmatched])
['eq3']
```

get_adjacent_to(*component*)

Return a list of components adjacent to the provided component in the cached bipartite incidence graph of variables and constraints

Parameters

component (ComponentData) – The variable or constraint data object whose adjacent components are returned

Returns

List of constraint or variable data objects adjacent to the provided component

Return type

list of *ComponentData*

Example

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.incidence_analysis import IncidenceGraphInterface
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var([1, 2])
>>> m.eq1 = pyo.Constraint(expr=m.x[1]**2 == 7)
>>> m.eq2 = pyo.Constraint(expr=m.x[1]*m.x[2] == 3)
>>> m.eq3 = pyo.Constraint(expr=m.x[1] + 2*m.x[2] == 5)
>>> igraph = IncidenceGraphInterface(m)
>>> adj_to_x2 = igraph.get_adjacent_to(m.x[2])
>>> print([c.name for c in adj_to_x2])
['eq2', 'eq3']
```

get_connected_components(*variables=None*, *constraints=None*)

Partition variables and constraints into weakly connected components of the incidence graph

These correspond to diagonal blocks in a block diagonalization of the incidence matrix.

Returns

- **var_blocks** (*list of lists of variables*) – Partition of variables into connected components
- **con_blocks** (*list of lists of constraints*) – Partition of constraints into corresponding connected components

get_diagonal_blocks(*variables=None*, *constraints=None*)

DEPRECATED.

Deprecated since version 6.5.0: `IncidenceGraphInterface.get_diagonal_blocks` is deprecated. Please use `IncidenceGraphInterface.block_triangularize` instead.

get_matrix_coord(*component*)

Return the row or column coordinate of the component in the incidence *matrix* of variables and constraints

Variables will return a column coordinate and constraints will return a row coordinate.

Parameters

component (ComponentData) – Component whose coordinate to locate

Returns
Column or row coordinate of the provided variable or constraint

Return type
`int`

`map_nodes_to_block_triangular_indices`(*variables=None*, *constraints=None*)

Map variables and constraints to indices of their diagonal blocks in a block lower triangular permutation

Returns

- **var_block_map** (`ComponentMap`) – Map from variables to their diagonal blocks in a block triangularization
- **con_block_map** (`ComponentMap`) – Map from constraints to their diagonal blocks in a block triangularization

`maximum_matching`(*variables=None*, *constraints=None*)

Return a maximum cardinality matching of variables and constraints.

The matching maps constraints to their matched variables.

Returns
A map from constraints to their matched variables.

Return type
`ComponentMap`

`plot`(*variables=None*, *constraints=None*, *title=None*, *show=True*)

Plot the bipartite incidence graph of variables and constraints

`remove_nodes`(*variables=None*, *constraints=None*)

Removes the specified variables and constraints (columns and rows) from the cached incidence matrix.

This is a "projection" of the variable and constraint vectors, rather than something like a vertex elimination. For the puropse of this method, there is no need to distinguish between variables and constraints. However, we provide the "constraints" argument so a call signature similar to other methods in this class is still valid.

Parameters

- **variables** (*list*) – VarData objects whose nodes will be removed from the incidence graph
- **constraints** (*list*) – ConData objects whose nodes will be removed from the incidence graph
- **note::** (*..*) – **Deprecation in Pyomo v6.7.2**

  The pre-6.7.2 implementation of `remove_nodes` allowed variables and constraints to remove to be specified in a single list. This made error checking difficult, and indeed, if invalid components were provided, we carried on silently instead of throwing an error or warning. As part of a fix to raise an error if an invalid component (one that is not part of the incidence graph) is provided, we now require variables and constraints to be specified separately.

`subgraph`(*variables*, *constraints*)

Extract a subgraph defined by the provided variables and constraints

Underlying data structures are copied, and constraints are not reinspected for incidence variables (the edges from this incidence graph are used).

Returns
A new incidence graph containing only the specified variables and constraints, and the edges between pairs thereof.

Return type
`IncidenceGraphInterface`

property col_block_map
DEPRECATED.

Deprecated since version 6.5.0: The `col_block_map` attribute is deprecated and will be removed.

property con_index_map
DEPRECATED.

Deprecated since version 6.5.0: `con_index_map` is deprecated. Please use `get_matrix_coord` instead.

property constraints
The constraints participating in the incidence graph

property incidence_matrix
The structural incidence matrix of variables and constraints.

Variables correspond to columns and constraints correspond to rows. All matrix entries have value 1.0.

property n_edges
The number of edges in the incidence graph, or the number of structural nonzeros in the incidence matrix

property row_block_map
DEPRECATED.

Deprecated since version 6.5.0: The `row_block_map` attribute is deprecated and will be removed.

property var_index_map
DEPRECATED.

Deprecated since version 6.5.0: `var_index_map` is deprecated. Please use `get_matrix_coord` instead.

property variables
The variables participating in the incidence graph

pyomo.contrib.incidence_analysis.interface.**extract_bipartite_subgraph**(*graph*, *nodes0*, *nodes1*)
Return the bipartite subgraph of a graph.

Two lists of nodes to project onto must be provided. These will correspond to the "bipartite sets" in the subgraph. If the two sets provided have M and N nodes, the subgraph will have nodes 0 through M+N-1, with the first M corresponding to the first set provided and the last N corresponding to the second set.

Parameters

- **graph** (*NetworkX Graph*) – The graph from which a subgraph is extracted
- **nodes0** (*list*) – A list of nodes in the original graph that will form the first bipartite set of the projected graph (and have `bipartite=0`)
- **nodes1** (*list*) – A list of nodes in the original graph that will form the second bipartite set of the projected graph (and have `bipartite=1`)

Returns
subgraph – Graph containing integer nodes corresponding to positions in the provided lists, with edges where corresponding nodes are adjacent in the original graph.

Return type
networkx.Graph

pyomo.contrib.incidence_analysis.interface.**get_bipartite_incidence_graph**(*variables*, *constraints*, ***kwds*)

Return the bipartite incidence graph of Pyomo variables and constraints.

Each node in the returned graph is an integer. The convention is that, for a graph with N variables and M constraints, nodes 0 through M-1 correspond to constraints and nodes M through M+N-1 correspond to variables. Nodes correspond to variables and constraints in the provided orders. For consistency with NetworkX's "convention", constraint nodes are tagged with `bipartite=0` while variable nodes are tagged with `bipartite=1`, although these attributes are not used.

Parameters

- **variables** (*List of Pyomo VarData objects*) – Variables that will appear in incidence graph
- **constraints** (*List of Pyomo ConstraintData objects*) – Constraints that will appear in incidence graph
- **include_fixed** (*Bool*) – Flag for whether fixed variable should be included in the incidence

Return type
networkx.Graph

pyomo.contrib.incidence_analysis.interface.**get_numeric_incidence_matrix**(*variables*, *constraints*)

Return the "numeric incidence matrix" (Jacobian) of Pyomo variables and constraints.

Each matrix value is the derivative of a constraint body with respect to a variable. Rows correspond to constraints and columns correspond to variables. Entries are included even if the value of the derivative is zero. Only active constraints and unfixed variables that participate in these constraints are included.

Parameters

- **variables** (*List of Pyomo VarData objects*)
- **constraints** (*List of Pyomo ConstraintData objects*)

Returns
COO matrix. Rows are indices into the user-provided list of constraints, columns are indices into the user-provided list of variables.

Return type
scipy.sparse.coo_matrix

pyomo.contrib.incidence_analysis.interface.**get_structural_incidence_matrix**(*variables*, *constraints*, ***kwds*)

Return the incidence matrix of Pyomo constraints and variables

Parameters

- **variables** (*List of Pyomo VarData objects*)
- **constraints** (*List of Pyomo ConstraintData objects*)
- **include_fixed** (*Bool*) – Flag for whether fixed variables should be included in the matrix nonzeros

Returns
COO matrix. Rows are indices into the user-provided list of constraints, columns are indices into the user-provided list of variables. Entries are 1.0.

Return type
scipy.sparse.coo_matrix

Maximum Matching

pyomo.contrib.incidence_analysis.matching.**maximum_matching**(*matrix_or_graph*, *top_nodes=None*)

Return a maximum cardinality matching of the provided matrix or bipartite graph

If a matrix is provided, the matching is returned as a map from row indices to column indices. If a bipartite graph is provided, a list of "top nodes" must be provided as well. These correspond to one of the "bipartite sets". The matching is then returned as a map from "top nodes" to the other set of nodes.

Parameters

- **matrix_or_graph** (*SciPy sparse matrix or NetworkX Graph*) – The matrix or graph whose maximum matching will be computed
- **top_nodes** (*list*) – Integer nodes representing a bipartite set in a graph. Must be provided if and only if a NetworkX Graph is provided.

Returns
max_matching – Dict mapping from integer nodes in the first bipartite set (row indices) to nodes in the second (column indices).

Return type
dict

Weakly Connected Components

pyomo.contrib.incidence_analysis.connected.**get_independent_submatrices**(*matrix*)

Partition a matrix into irreducible block diagonal form

This is equivalent to identifying the connected components of the bipartite incidence graph of rows and columns.

Parameters
matrix (scipy.sparse.coo_matrix) – Matrix to partition into block diagonal form

Returns

- **row_blocks** (*list of lists*) – Partition of row coordinates into diagonal blocks
- **col_blocks** (*list of lists*) – Partition of column coordinates into diagonal blocks

Block Triangularization

pyomo.contrib.incidence_analysis.triangularize.**block_triangularize**(*matrix*, *matching=None*)

Compute ordered partitions of the matrix's rows and columns that permute the matrix to block lower triangular form

Subsets in the partition correspond to diagonal blocks in the block triangularization. The order is topological, with ties broken "lexicographically".

Parameters

- **matrix** (scipy.sparse.coo_matrix) – Matrix whose rows and columns will be permuted
- **matching** (dict) – A perfect matching. Maps rows to columns *and* columns back to rows.

Returns

- **row_partition** (*list of lists*) – A partition of rows. The inner lists hold integer row coordinates.

- **col_partition** (*list of lists*) – A partition of columns. The inner lists hold integer column coordinates.

Note

Breaking change in Pyomo 6.5.0

The pre-6.5.0 `block_triangularize` function returned maps from each row or column to the index of its block in a block lower triangularization as the original intent of this function was to identify when coordinates do or don't share a diagonal block in this partition. Since then, the dominant use case of `block_triangularize` has been to partition variables and constraints into these blocks and inspect or solve each block individually. A natural return type for this functionality is the ordered partition of rows and columns, as lists of lists. This functionality was previously available via the `get_diagonal_blocks` method, which was confusing as it did not capture that the partition was the diagonal of a block *triangularization* (as opposed to diagonalization). The pre-6.5.0 functionality of `block_triangularize` is still available via the `map_coords_to_block_triangular_indices` function.

`pyomo.contrib.incidence_analysis.triangularize.`**`get_blocks_from_maps`**(*row_block_map*, *col_block_map*)

DEPRECATED.

Deprecated since version 6.5.0: `get_blocks_from_maps` is deprecated. This functionality has been incorporated into `block_triangularize`.

`pyomo.contrib.incidence_analysis.triangularize.`**`get_diagonal_blocks`**(*matrix*, *matching=None*)

DEPRECATED.

Deprecated since version 6.5.0: `get_diagonal_blocks` has been deprecated. Please use `block_triangularize` instead.

`pyomo.contrib.incidence_analysis.triangularize.`**`get_scc_of_projection`**(*graph*, *top_nodes*, *matching=None*)

Return the topologically ordered strongly connected components of a bipartite graph, projected with respect to a perfect matching

The provided undirected bipartite graph is projected into a directed graph on the set of "top nodes" by treating "matched edges" as out-edges and "unmatched edges" as in-edges. Then the strongly connected components of the directed graph are computed. These strongly connected components are unique, regardless of the choice of perfect matching. The strongly connected components form a directed acyclic graph, and are returned in a topological order. The order is unique, as ambiguities are resolved "lexicographically".

The "direction" of the projection (where matched edges are out-edges) leads to a block *lower* triangular permutation when the top nodes correspond to *rows* in the bipartite graph of a matrix.

Parameters

- **graph** (*NetworkX Graph*) – A bipartite graph
- **top_nodes** (*list*) – One of the bipartite sets in the graph
- **matching** (*dict*) – Maps each node in `top_nodes` to its matched node

Returns

The outer list is a list of strongly connected components. Each strongly connected component is a list of tuples of matched nodes. The first node is a "top node", and the second is an "other node".

Return type

list of lists

Dulmage-Mendelsohn Partition

`class pyomo.contrib.incidence_analysis.dulmage_mendelsohn.ColPartition(unmatched, underconstrained, overconstrained, square)`

Named tuple containing the subsets of the Dulmage-Mendelsohn partition when applied to matrix columns (variables).

`overconstrained`

Alias for field number 2

`square`

Alias for field number 3

`underconstrained`

Alias for field number 1

`unmatched`

Alias for field number 0

`class pyomo.contrib.incidence_analysis.dulmage_mendelsohn.RowPartition(unmatched, overconstrained, underconstrained, square)`

Named tuple containing the subsets of the Dulmage-Mendelsohn partition when applied to matrix rows (constraints).

`overconstrained`

Alias for field number 1

`square`

Alias for field number 3

`underconstrained`

Alias for field number 2

`unmatched`

Alias for field number 0

`pyomo.contrib.incidence_analysis.dulmage_mendelsohn.dulmage_mendelsohn(matrix_or_graph, top_nodes=None, matching=None)`

Partition a bipartite graph or incidence matrix according to the Dulmage-Mendelsohn characterization

The Dulmage-Mendelsohn partition tells which nodes of the two bipartite sets *can possibly be* unmatched after a maximum cardinality matching. Applied to an incidence matrix, it can be interpreted as partitioning rows and columns into under-constrained, over-constrained, and well-constrained subsystems.

As it is often useful to explicitly check the unmatched rows and columns, `dulmage_mendelsohn` partitions rows into the subsets:

- **underconstrained** - The rows matched with *possibly* unmatched columns (unmatched and underconstrained columns)
- **square** - The well-constrained rows, which are matched with well-constrained columns
- **overconstrained** - The matched rows that *can possibly be* unmatched in some maximum cardinality matching

- **unmatched** - The unmatched rows in a particular maximum cardinality matching

and partitions columns into the subsets:

- **unmatched** - The unmatched columns in a particular maximum cardinality matching
- **underconstrained** - The columns that *can possibly be* unmatched in some maximum cardinality matching
- **square** - The well-constrained columns, which are matched with well-constrained rows
- **overconstrained** - The columns matched with *possibly* unmatched rows (unmatched and overconstrained rows)

While the Dulmage-Mendelsohn decomposition does not specify an order within any of these subsets, the order returned by this function preserves the maximum matching that is used to compute the decomposition. That is, zipping "corresponding" row and column subsets yields pairs in this maximum matching. For example:

```
>>> row_dmpartition, col_dmpartition = dulmage_mendelsohn(matrix)
>>> rdmp = row_dmpartition
>>> cdmp = col_dmpartition
>>> matching = list(zip(
...     rdmp.underconstrained + rdmp.square + rdmp.overconstrained,
...     cdmp.underconstrained + cdmp.square + cdmp.overconstrained,
... ))
>>> # matching is a valid maximum matching of rows and columns of the matrix!
```

Parameters

- **matrix_or_graph** (scipy.sparse.coo_matrix or networkx.Graph) – The incidence matrix or bipartite graph to be partitioned
- **top_nodes** (*list*) – List of nodes in one bipartite set of the graph. Must be provided if a graph is provided.
- **matching** (*dict*) – A maximum cardinality matching in the form of a dict mapping from "top nodes" to their matched nodes *and* from the matched nodes back to the "top nodes".

Returns

- **row_dmp** (*RowPartition*) – The Dulmage-Mendelsohn partition of rows
- **col_dmp** (*ColPartition*) – The Dulmage-Mendelsohn partition of columns

Block Triangular Decomposition Solver

pyomo.contrib.incidence_analysis.scc_solver.**generate_strongly_connected_components**(*constraints*, *variables=None*, *include_fixed=False*, *igraph=None*)

Yield in order BlockData that each contain the variables and constraints of a single diagonal block in a block lower triangularization of the incidence matrix of constraints and variables

These diagonal blocks correspond to strongly connected components of the bipartite incidence graph, projected with respect to a perfect matching into a directed graph.

Parameters

- **constraints** (*List of Pyomo constraint data objects*) – Constraints used to generate strongly connected components.

- **variables** (*List of Pyomo variable data objects*) – Variables that may participate in strongly connected components. If not provided, all variables in the constraints will be used.
- **include_fixed** (*Bool, optional*) – Indicates whether fixed variables will be included when identifying variables in constraints.
- **igraph** (IncidenceGraphInterface, *optional*) – Incidence graph containing (at least) the provided constraints and variables.

Yields

Tuple of `BlockData`, list-of-variables – Blocks containing the variables and constraints of every strongly connected component, in a topological order. The variables are the "input variables" for that block.

`pyomo.contrib.incidence_analysis.scc_solver.`**`solve_strongly_connected_components`**(*block*, *, *solver=None*, *solve_kwds=None*, *use_calc_var=True*, *calc_var_kwds=None*)

Solve a square system of variables and equality constraints by solving strongly connected components individually.

Strongly connected components (of the directed graph of constraints obtained from a perfect matching of variables and constraints) are the diagonal blocks in a block triangularization of the incidence matrix, so solving the strongly connected components in topological order is sufficient to solve the entire block.

One-by-one blocks are solved using Pyomo's calculate_variable_from_constraint function, while higher-dimension blocks are solved using the user-provided solver object.

Parameters

- **block** (*Pyomo Block*) – The Pyomo block whose variables and constraints will be solved
- **solver** (*Pyomo solver object*) – The solver object that will be used to solve strongly connected components of size greater than one constraint. Must implement a solve method.
- **solve_kwds** (*Dictionary*) – Keyword arguments for the solver's solve method
- **use_calc_var** (*Bool*) – Whether to use `calculate_variable_from_constraint` for one-by-one square system solves
- **calc_var_kwds** (*Dictionary*) – Keyword arguments for calculate_variable_from_constraint

Return type

List of results objects returned by each call to solve

If you are wondering what Incidence Analysis is and would like to learn more, please see *Overview*. If you already know what Incidence Analysis is and are here for reference, see *Incidence Analysis Tutorial* or *API Reference* as needed.

3.4.6 MPC

Pyomo MPC contains data structures and utilities for dynamic optimization and rolling horizon applications, e.g. model predictive control.

Overview

What does this package contain?

1. Data structures for values and time series data associated with time-indexed variables (or parameters, or named expressions). Examples are setpoint values associated with a subset of state variables or time series data from a simulation
2. Utilities for loading and extracting this data into and from variables in a model
3. Utilities for constructing components from this data (expressions, constraints, and objectives) that are useful for dynamic optimization

What is the goal of this package?

This package was written to help developers of Pyomo-based dynamic optimization case studies, especially rolling horizon dynamic optimization case studies, write scripts that are small, legible, and maintainable. It does this by providing utilities for mundane data-management and model construction tasks, allowing the developer to focus on their application.

Why is this package useful?

First, it is not normally easy to extract "flattened" time series data, in which all indexing structure other than time-indexing has been flattened to yield a set of one-dimensional arrays, from a Pyomo model. This is an extremely convenient data structure to have for plotting, analysis, initialization, and manipulation of dynamic models. If all variables are indexed by time and only time, this data is relatively easy to obtain. The first issue comes up when dealing with components that are indexed by time in addition to some other set(s). For example:

```
>>> import pyomo.environ as pyo

>>> m = pyo.ConcreteModel()
>>> m.time = pyo.Set(initialize=[0, 1, 2])
>>> m.comp = pyo.Set(initialize=["A", "B"])
>>> m.var = pyo.Var(m.time, m.comp, initialize=1.0)

>>> t0 = m.time.first()
>>> data = {
...     m.var[t0, j].name: [m.var[i, j].value for i in m.time]
...     for j in m.comp
... }
>>> data
{'var[0,A]': [1.0, 1.0, 1.0], 'var[0,B]': [1.0, 1.0, 1.0]}
```

To generate data in this form, we need to (a) know that our variable is indexed by time and `m.comp` and (b) arbitrarily select a time index `t0` to generate a unique key for each time series. This gets more difficult when blocks and time-indexed blocks are used as well. The first difficulty can be alleviated using `flatten_dae_components` from `pyomo.dae.flatten`:

```
>>> import pyomo.environ as pyo
>>> from pyomo.dae.flatten import flatten_dae_components

>>> m = pyo.ConcreteModel()
>>> m.time = pyo.Set(initialize=[0, 1, 2])
>>> m.comp = pyo.Set(initialize=["A", "B"])
>>> m.var = pyo.Var(m.time, m.comp, initialize=1.0)
```

(continues on next page)

(continued from previous page)

```
>>> t0 = m.time.first()
>>> scalar_vars, dae_vars = flatten_dae_components(m, m.time, pyo.Var)
>>> data = {var[t0].name: list(var[:].value) for var in dae_vars}
>>> data
{'var[0,A]': [1.0, 1.0, 1.0], 'var[0,B]': [1.0, 1.0, 1.0]}
```

Addressing the arbitrary `t0` index requires us to ask what key we would like to use to identify each time series in our data structure. The key should uniquely correspond to a component, or "sub-component" that is indexed only by time. A slice, e.g. `m.var[:, "A"]` seems natural. However, Pyomo provides a better data structure that can be constructed from a component, slice, or string, called `ComponentUID`. Being constructable from a string is important as we may want to store or serialize this data in a form that is agnostic of any particular `ConcreteModel` object. We can now generate our data structure as:

```
>>> data = {
...     pyo.ComponentUID(var.referent): list(var[:].value)
...     for var in dae_vars
... }
>>> data
{var[*,A]: [1.0, 1.0, 1.0], var[*,B]: [1.0, 1.0, 1.0]}
```

This is the structure of the underlying dictionary in the `TimeSeriesData` class provided by this package. We can generate this data using this package as:

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.mpc import DynamicModelInterface

>>> m = pyo.ConcreteModel()
>>> m.time = pyo.Set(initialize=[0, 1, 2])
>>> m.comp = pyo.Set(initialize=["A", "B"])
>>> m.var = pyo.Var(m.time, m.comp, initialize=1.0)

>>> # Construct a helper class for interfacing model with data
>>> helper = DynamicModelInterface(m, m.time)

>>> # Generates a TimeSeriesData object
>>> series_data = helper.get_data_at_time()

>>> # Get the underlying dictionary
>>> data = series_data.get_data()
>>> data
{var[*,A]: [1.0, 1.0, 1.0], var[*,B]: [1.0, 1.0, 1.0]}
```

The first value proposition of this package is that `DynamicModelInterface` and `TimeSeriesData` provide wrappers to ease loading and extraction of data via `flatten_dae_components` and `ComponentUID`.

The second difficulty addressed by this package is that of extracting and loading data between (potentially) different models. For instance, in model predictive control, we often want to extract data from a particular time point in a plant model and load it into a controller model as initial conditions. This can be done as follows:

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.mpc import DynamicModelInterface
```

(continues on next page)

(continued from previous page)

```
>>> m1 = pyo.ConcreteModel()
>>> m1.time = pyo.Set(initialize=[0, 1, 2])
>>> m1.comp = pyo.Set(initialize=["A", "B"])
>>> m1.var = pyo.Var(m1.time, m1.comp, initialize=1.0)

>>> m2 = pyo.ConcreteModel()
>>> m2.time = pyo.Set(initialize=[0, 1, 2])
>>> m2.comp = pyo.Set(initialize=["A", "B"])
>>> m2.var = pyo.Var(m2.time, m2.comp, initialize=2.0)

>>> # Construct helper objects
>>> m1_helper = DynamicModelInterface(m1, m1.time)
>>> m2_helper = DynamicModelInterface(m2, m2.time)

>>> # Extract data from final time point of m2
>>> tf = m2.time.last()
>>> tf_data = m2_helper.get_data_at_time(tf)

>>> # Load data into initial time point of m1
>>> t0 = m1.time.first()
>>> m1_helper.load_data(tf_data, time_points=t0)

>>> # Get TimeSeriesData object
>>> series_data = m1_helper.get_data_at_time()
>>> # Get underlying dictionary
>>> series_data.get_data()
{var[*,A]: [2.0, 1.0, 1.0], var[*,B]: [2.0, 1.0, 1.0]}
```

Note

Here we rely on the fact that our variable has the same name in both models.

Finally, this package provides methods for constructing components like tracking cost expressions and piecewise-constant constraints from the provided data structures. For example, the following code constructs a tracking cost expression.

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.mpc import DynamicModelInterface

>>> m = pyo.ConcreteModel()
>>> m.time = pyo.Set(initialize=[0, 1, 2])
>>> m.comp = pyo.Set(initialize=["A", "B"])
>>> m.var = pyo.Var(m.time, m.comp, initialize=1.0)

>>> # Construct helper object
>>> helper = DynamicModelInterface(m, m.time)

>>> # Construct data structure for setpoints
>>> setpoint = {m.var[:, "A"]: 0.5, m.var[:, "B"]: 2.0}
>>> var_set, tr_cost = helper.get_penalty_from_target(setpoint)
>>> m.setpoint_idx = var_set
```

(continues on next page)

(continued from previous page)

```
>>> m.tracking_cost = tr_cost
>>> m.tracking_cost.pprint()
tracking_cost : Size=6, Index=setpoint_idx*time
    Key    : Expression
    (0, 0) : (var[0,A] - 0.5)**2
    (0, 1) : (var[1,A] - 0.5)**2
    (0, 2) : (var[2,A] - 0.5)**2
    (1, 0) : (var[0,B] - 2.0)**2
    (1, 1) : (var[1,B] - 2.0)**2
    (1, 2) : (var[2,B] - 2.0)**2
```

These methods will hopefully allow developers to declutter dynamic optimization scripts and pay more attention to the application of the optimization problem rather than the setup of the optimization problem.

Who develops and maintains this package?

This package was developed by Robert Parker while a PhD student in Larry Biegler's group at CMU, with guidance from Bethany Nicholson and John Siirola.

Examples

Please see `pyomo/contrib/mpc/examples/cstr/run_openloop.py` and `pyomo/contrib/mpc/examples/cstr/run_mpc.py` for examples of some simple use cases.

Frequently asked questions

1. Why not use Pandas DataFrames?

Pandas DataFrames are a natural data structure for storing "columns" of time series data. These columns, or individual time series, could each represent the data for a single variable. This is very similar to the TimeSeriesData class introduced in this package. The reason a new data structure is introduced is primarily that a DataFrame does not provide any utility for converting labels into a consistent format, as TimeSeriesData does by accepting variables, strings, slices, etc. as keys and converting them into the form of a time-indexed ComponentUID. Also, DataFrames do not have convenient analogs for scalar data and time interval data, which this package provides as the ScalarData and IntervalData classes with very similar APIs to TimeSeriesData.

API Reference

Data Structures

automodule:: pyomo.contrib.mpc.data.dynamic_data_base :members: :noindex:

automodule:: pyomo.contrib.mpc.data.scalar_data :members: :noindex:

automodule:: pyomo.contrib.mpc.data.series_data :members: :noindex:

automodule:: pyomo.contrib.mpc.data.interval_data :members: :noindex:

`pyomo.contrib.mpc.data.get_cuid.`**`get_indexed_cuid`**(*var*, *sets=None*, *dereference=None*, *context=None*)

Attempt to convert the provided "var" object into a CUID with wildcards

Parameters

- **var** – Object to process. May be a VarData, IndexedVar (reference or otherwise), ComponentUID, slice, or string.
- **sets** (*Tuple of sets*) – Sets to use if slicing a vardata object

- **dereference** (*None or int*) – Number of times we may access referent attribute to recover a "base component" from a reference.
- **context** (Block) – Block with respect to which slices and CUIDs will be generated

Returns
ComponentUID corresponding to the provided `var` and sets

Return type
`ComponentUID`

Data Conversion

`pyomo.contrib.mpc.data.convert.`**`interval_to_series`**(*data*, *time_points=None*, *tolerance=0.0*, *use_left_endpoints=False*, *prefer_left=True*)

Parameters

- **data** (IntervalData) – Data to convert to a TimeSeriesData object
- **time_points** (*Iterable (optional)*) – Points at which time series will be defined. Values are taken from the interval in which each point lives. The default is to use the right endpoint of each interval.
- **tolerance** (*Float (optional)*) – Tolerance within which time points are considered equal. Default is zero.
- **use_left_endpoints** (*Bool (optional)*) – Whether the left endpoints should be used in the case when time_points is not provided. Default is False, meaning that the right interval endpoints will be used. Should not be set if time points are provided.
- **prefer_left** (*Bool (optional)*) – If time_points is provided, and a time point is equal (within tolerance) to a boundary between two intervals, this flag controls which interval is used.

Return type
TimeSeriesData

`pyomo.contrib.mpc.data.convert.`**`series_to_interval`**(*data*, *use_left_endpoints=False*)

Parameters

- **data** (TimeSeriesData) – Data that will be converted into an IntervalData object
- **use_left_endpoints** (*Bool (optional)*) – Flag indicating whether values on intervals should come from the values at the left or right endpoints of the intervals

Return type
IntervalData

Interfaces

class `pyomo.contrib.mpc.interfaces.model_interface.`**`DynamicModelInterface`**(*model*, *time*, *context=NOTSET*)

A helper class for working with dynamic models, e.g. those where many components are indexed by some ordered set referred to as "time."

This class provides methods for interacting with time-indexed components, for instance, loading and extracting data or shifting values by some time offset. It also provides methods for constructing components useful for dynamic optimization.

copy_values_at_time(*source_time=None*, *target_time=None*)

Copy values of all time-indexed variables from source time point to target time points.

Parameters

- **source_time** (*Float*) – Time point from which to copy values.
- **target_time** (*Float or iterable*) – Time point or points to which to copy values.

get_data_at_time(*time=None*, *include_expr=False*)

Gets data at a single time point or set of time points. Note that the returned type changes depending on whether a scalar or iterable is supplied.

get_penalty_from_target(*target_data*, *time=None*, *variables=None*, *weight_data=None*, *variable_set=None*, *tolerance=None*, *prefer_left=None*)

A method to get a quadratic penalty expression from a provided setpoint data structure

Parameters

- **target_data** (ScalarData, TimeSeriesData, *or* IntervalData) – Holds target values for variables
- **time** (Set *(optional)*) – Points at which to apply the tracking cost. Default will use the model's time set.
- **variables** (*List of Pyomo VarData (optional)*) – Subset of variables supplied in setpoint_data to use in the tracking cost. Default is to use all variables supplied.
- **weight_data** (ScalarData *(optional)*) – Holds the weights to use in the tracking cost for each variable
- **variable_set** (Set *(optional)*) – A set indexing the list of provided variables, if one already exists.
- **tolerance** (*Float (optional)*) – Tolerance for checking inclusion in an interval. Only may be provided if IntervalData is provided for target_data. In this case the default is 0.0.
- **prefer_left** (*Bool (optional)*) – Flag indicating whether the left end point of intervals should be preferred over the right end point. Only may be provided if IntervalData is provided for target_data. In this case the default is False.

Returns

Set indexing the list of variables to be penalized, and Expression indexed by this set and time. This Expression contains the weighted tracking cost for each variable at each point in time.

Return type

Set, *Expression*

get_piecewise_constant_constraints(*variables*, *sample_points*, *use_next=True*, *tolerance=0.0*)

A method to get an indexed constraint ensuring that inputs are piecewise constant.

Parameters

- **variables** (*List of Pyomo Vars*) – Variables to enforce piecewise constant
- **sample_points** (*List of floats*) – Points marking the boundaries of intervals within which variables must be constant
- **use_next** (*Bool (optional)*) – Whether to enforce constancy by setting each variable equal to itself at the next point in time (as opposed to at the previous point in time). Default is True.

- **tolerance** (*Float (optional)*) – Absolute tolerance used to determine whether provided sample points are in the model's time set.

Returns
First entry is a Set indexing the list of provided variables (with integers). Second entry is a constraint indexed by this set and time enforcing the piecewise constant condition via equality constraints.

Return type
Tuple

get_scalar_variable_data()

Get data corresponding to non-time-indexed variables.

Returns
Maps CUIDs of non-time-indexed variables to the value of these variables.

Return type
dict

load_data(*data*, *time_points=None*, *tolerance=0.0*, *prefer_left=None*, *exclude_left_endpoint=None*, *exclude_right_endpoint=None*, *ignore_named_expressions=False*)

Method to load data into the model.

Loads data into indicated variables in the model, possibly at specified time points.

Parameters

- **data** (ScalarData, TimeSeriesData, *or mapping*) – If ScalarData, loads values into indicated variables at all (or specified) time points. If TimeSeriesData, loads lists of values into time points. If mapping, checks whether each variable and value is indexed or iterable and correspondingly loads data into variables.
- **time_points** (*Iterable (optional)*) – Subset of time points into which data should be loaded. Default of None corresponds to loading into all time points.
- **ignore_named_expressions** (*(optional)*) – ignore data for named expressions, otherwise a TypeError will be raise on encountering a named expression.

shift_values_by_time(*dt*)

Shift values in time indexed variables by a specified time offset.

class pyomo.contrib.mpc.interfaces.var_linker.**DynamicVarLinker**(*source_variables*, *target_variables*, *source_time=None*, *target_time=None*)

The purpose of this class is so that we do not have to call find_component or construct ComponentUIDs in a loop when transferring values between two different dynamic models. It also allows us to transfer values between variables that have different names in different models.

Modeling Components

pyomo.contrib.mpc.modeling.constraints.**get_piecewise_constant_constraints**(*inputs*, *time*, *sample_points*, *use_next=True*)

Returns an IndexedConstraint that constrains the provided variables to be constant between the provided sample points

Parameters

- **inputs** (*list of variables*) – Time-indexed variables that will be constrained piecewise constant
- **time** (Set) – Set of points at which provided variables will be constrained
- **sample_points** (*List of floats*) – Points at which "constant constraints" will be omitted; these are points at which the provided variables may vary.
- **use_next** (*Bool (default True)*) – Whether the next time point will be used in the constant constraint at each point in time. Otherwise, the previous time point is used.

Returns
A RangeSet indexing the list of variables provided and a Constraint indexed by the product of this RangeSet and time.

Return type
Set, *IndexedConstraint*

pyomo.contrib.mpc.modeling.cost_expressions.**get_penalty_from_constant_target**(*variables*, *time*, *setpoint_data*, *weight_data=None*, *variable_set=None*)

This function returns a tracking cost IndexedExpression for the given time-indexed variables and associated setpoint data.

Parameters

- **variables** (*list*) – List of time-indexed variables to include in the tracking cost expression
- **time** (*iterable*) – Set of variable indices for which a cost expression will be created
- **setpoint_data** (ScalarData, *dict*, *or* ComponentMap) – Maps variable names to setpoint values
- **weight_data** (ScalarData, *dict*, *or* ComponentMap) – Optional. Maps variable names to tracking cost weights. If not provided, weights of one are used.
- **variable_set** (Set) – Optional. A set of indices into the provided list of variables by which the cost expression will be indexed.

Returns
RangeSet that indexes the list of variables provided and an Expression indexed by the RangeSet and time containing the cost term for each variable at each point in time.

Return type
Set, *Expression*

pyomo.contrib.mpc.modeling.cost_expressions.**get_penalty_from_piecewise_constant_target**(*variables*, *time*, *setpoint_data*, *weight_data=None*, *variable_set=None*, *tolerance=0.0*, *prefer_left=True*)

Returns an IndexedExpression penalizing deviation between the specified variables and piecewise constant target data.

Parameters

- **variables** (*List of Pyomo variables*) – Variables that participate in the cost expressions.
- **time** (*Iterable*) – Index used for the cost expression
- **setpoint_data** (IntervalData) – Holds the piecewise constant values that will be used as setpoints
- **weight_data** (ScalarData *(optional)*) – Weights for variables. Default is all ones.
- **tolerance** (*Float (optional)*) – Tolerance used for determining whether a time point is within an interval. Default is zero.
- **prefer_left** (*Bool (optional)*) – If a time point lies at the boundary of two intervals, whether the value on the left will be chosen. Default is True.

Returns

Pyomo Expression, indexed by time, for the total weighted tracking cost with respect to the provided setpoint.

Return type

Set, *Expression*

pyomo.contrib.mpc.modeling.cost_expressions.**get_penalty_from_target**(*variables*, *time*, *setpoint_data*, *weight_data=None*, *variable_set=None*, *tolerance=None*, *prefer_left=None*)

A function to get a penalty expression for specified variables from a target that is constant, piecewise constant, or time-varying.

This function accepts *ScalarData*, *IntervalData*, or *TimeSeriesData* objects, or compatible mappings/tuples as the target, and builds the appropriate penalty expression for each. Mappings are converted to ScalarData, and tuples (of data dict, time list) are unpacked and converted to IntervalData or TimeSeriesData depending on the contents of the time list.

Parameters

- **variables** (*List*) – List of time-indexed variables to be penalized
- **time** (Set) – Set of time points at which to construct penalty expressions. Also indexes the returned Expression.
- **setpoint_data** (ScalarData, TimeSeriesData, *or* IntervalData) – Data structure representing the possibly time-varying or piecewise constant setpoint
- **weight_data** (ScalarData *(optional)*) – Data structure holding the weights to be applied to each variable
- **variable_set** (Set *(optional)*) – Set indexing the provided variables, if one already exists. Also indexes the returned Expression.
- **tolerance** (*Float (optional)*) – Tolerance for checking inclusion within an interval. Only may be provided if IntervalData is provided as the setpoint.

- **prefer_left** (*Bool (optional)*) – Flag indicating whether left endpoints of intervals should take precedence over right endpoints. Default is False. Only may be provided if IntervalData is provided as the setpoint.

Returns
Set indexing the list of provided variables and an Expression, indexed by this set and the provided time set, containing the penalties for each variable at each point in time.

Return type
Set, *Expression*

pyomo.contrib.mpc.modeling.cost_expressions.**get_penalty_from_time_varying_target**(*variables*, *time*, *setpoint_data*, *weight_data=None*, *variable_set=None*)

Constructs a penalty expression for the specified variables and specified time-varying target data.

Parameters

- **variables** (*List of Pyomo variables*) – Variables that participate in the cost expressions.
- **time** (*Iterable*) – Index used for the cost expression
- **setpoint_data** (TimeSeriesData) – Holds the trajectory values that will be used as a setpoint
- **weight_data** (ScalarData *(optional)*) – Weights for variables. Default is all ones.
- **variable_set** (Set *(optional)*) – Set indexing the list of provided variables, if one exists already.

Returns
Set indexing the list of provided variables and Expression, indexed by the variable set and time, for the total weighted penalty with respect to the provided setpoint.

Return type
Set, *Expression*

pyomo.contrib.mpc.modeling.terminal.**get_penalty_at_time**(*variables*, *t*, *target_data*, *weight_data=None*, *time_set=None*, *variable_set=None*)

Returns an Expression penalizing the deviation of the specified variables at the specified point in time from the specified target

Parameters

- **variables** (*List*) – List of time-indexed variables that will be penalized
- **t** (*Float*) – Time point at which to apply the penalty
- **target_data** (ScalarData) – ScalarData object containing the target for (at least) the variables to be penalized
- **weight_data** (ScalarData *(optional)*) – ScalarData object containing the penalty weights for (at least) the variables to be penalized
- **time_set** (Set *(optional)*) – Time set that indexes the provided variables. This is only used if target or weight data are provided as a ComponentMap with VarData as keys. In this case the Set is necessary to recover the CUIDs used internally as keys

- **variable_set** (Set *(optional)*) – Set indexing the list of variables provided, if such a set already exists

Returns
Set indexing the list of variables provided and an Expression, indexed by this set, containing the weighted penalty expressions

Return type
Set, *Expression*

pyomo.contrib.mpc.modeling.terminal.**get_terminal_penalty**(*variables*, *time_set*, *target_data*, *weight_data=None*, *variable_set=None*)

Returns an Expression penalizing the deviation of the specified variables at the final point in time from the specified target

Parameters

- **variables** (*List*) – List of time-indexed variables that will be penalized
- **time_set** (Set) – Time set that indexes the provided variables. Penalties are applied at the last point in this set.
- **target_data** (ScalarData) – ScalarData object containing the target for (at least) the variables to be penalized
- **weight_data** (ScalarData *(optional)*) – ScalarData object containing the penalty weights for (at least) the variables to be penalized
- **variable_set** (Set *(optional)*) – Set indexing the list of variables provided, if such a set already exists

Returns
Set indexing the list of variables provided and an Expression, indexed by this set, containing the weighted penalty expressions

Return type
Set, *Expression*

Citation

If you use Pyomo MPC in your research, please cite the following paper:

```
@article{parker2023mpc,
title = {Model predictive control simulations with block-hierarchical differential-
↪algebraic process models},
journal = {Journal of Process Control},
volume = {132},
pages = {103113},
year = {2023},
issn = {0959-1524},
doi = {https://doi.org/10.1016/j.jprocont.2023.103113},
url = {https://www.sciencedirect.com/science/article/pii/S0959152423002007},
author = {Robert B. Parker and Bethany L. Nicholson and John D. Siirola and Lorenz T.␣
↪Biegler},
}
```

3.4.7 Parameter Estimation

`parmest` is a Python package built on the Pyomo optimization modeling language ([Pyomo-paper], [PyomoBookIII]) to support parameter estimation using experimental data along with confidence regions and subsequent creation of scenarios for stochastic programming.

Citation for parmest

If you use parmest, please cite [Parmest-paper]

Index of parmest documentation

Overview

The Python package called parmest facilitates model-based parameter estimation along with characterization of uncertainty associated with the estimates. For example, parmest can provide confidence regions around the parameter estimates. Additionally, parameter vectors, each with an attached probability estimate, can be used to build scenarios for design optimization.

Functionality in parmest includes:

- Model-based parameter estimation using experimental data
- Covariance matrix estimation
- Bootstrap resampling for uncertainty quantification
- Confidence regions based on single or multi-variate distributions
- Likelihood ratio test
- Leave-N-out cross validation
- Regularization for objective function improvement
- Multi-start initialization optimization
- Parallel processing

Background

The goal of parameter estimation is to estimate values for a vector, $\boldsymbol{\theta}$, to use in the functional form

$$\boldsymbol{y}_i = \boldsymbol{f}\left(\boldsymbol{x}_i, \boldsymbol{\theta}\right) + \boldsymbol{\varepsilon}_i \quad \forall \; i \in \{1, \ldots, n\}$$

where $\boldsymbol{y}_i \in \mathbb{R}^m$ are observations of the measured or output variables, $\boldsymbol{f}(\cdot)$ is the model function, $\boldsymbol{x}_i \in \mathbb{R}^q$ are the decision or input variables, $\boldsymbol{\theta} \in \mathbb{R}^p$ are the model parameters, $\boldsymbol{\varepsilon}_i \in \mathbb{R}^m$ are measurement errors, and n is the number of experiments.

The following least squares objective can be used to estimate model parameters from data assuming that the measurement errors follow a Gaussian distribution:

$$\min_{\boldsymbol{\theta}} \; g(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$$

where $g(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ can be:

1. Sum of squared errors

   If the measurement errors (which are assumed to follow a Gaussian distribution) are independent and identically distributed, the objective function can be defined as the sum of squared errors

$$g(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta})\right)^{\mathrm{T}} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta})\right)$$

2. Weighted sum of squared errors

   When the measurement errors are correlated and their covariance matrix, $\boldsymbol{\Sigma}_{\boldsymbol{y}}$, is known a priori, the objective function is defined as the weighted sum of squared errors

$$g(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^{n} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta})\right)^{\mathrm{T}} \boldsymbol{\Sigma}_{\boldsymbol{y}}^{-1} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta})\right)$$

Custom objectives can also be defined for parameter estimation.

In the applications of interest to us, the function $g(\cdot)$ is usually defined as an optimization problem with a large number of (perhaps constrained) optimization variables, a subset of which are fixed at values $\boldsymbol{x}$ when the optimization is performed. In other applications, the values of $\boldsymbol{\theta}$ are fixed parameter values, but for the problem formulation above, the values of $\boldsymbol{\theta}$ are the primary optimization variables. Note that in general, the function $g(\cdot)$ will have a large set of parameters that are not included in $\boldsymbol{\theta}$.

Installation Instructions

parmest is included in Pyomo (pyomo/contrib/parmest). To run parmest, you will need Python version 3.x along with various Python package dependencies and the IPOPT software library for non-linear optimization.

Python Package Dependencies

1. Install NumPy and Pandas with your preferred package manager; both NumPy and SciPy are required dependencies of parmest. You may install NumPy and Pandas with, for example, `conda`:

   ```
   conda install numpy pandas
   ```

   or `pip`:

   ```
   pip install numpy pandas
   ```

2. Install Pyomo. parmest is included in the Pyomo software package, at pyomo/contrib/parmest.
3. (*Optional*) Install `matplotlib` and `scipy`:

   ```
   pip install scipy matplotlib
   ```

4. (*Optional*) Install `seaborn`

   ```
   pip install seaborn
   ```

IPOPT

The IPOPT project homepage is https://github.com/coin-or/Ipopt

Testing

The following commands can be used to test parmest:

```
cd pyomo/contrib/parmest/tests
python test_parmest.py
```

Parmest Quick Start Guide

This quick start guide shows how to use parmest to estimate model parameters from experimental data as well as compute their uncertainty. The model and data used in this guide were taken from [RB01].

The mathematical model of interest is:

$$y_i(\theta_1, \theta_2, t_i) = \theta_1 \left(1 - e^{-\theta_2 t_i}\right), \quad \forall \quad i \in 1, \ldots, n$$

Where y is the observation of the measured variable, t is the time, θ_1 is the asymptote, and θ_2 is the rate constant.

The experimental data is given in the table below:

Table 3.10: Data

hour	y
1	8.3
2	10.3
3	19.0
4	16.0
5	15.6
7	19.8

To use parmest to estimate θ_1 and θ_2 from the data, we provide the following detailed steps:

Step 0: Import parmest and Pandas

Before solving the parameter estimation problem, the following code must be executed to import the required packages for parameter estimation in parmest:

```
>>> import pyomo.contrib.parmest.parmest as parmest
>>> import pandas as pd
```

Step 1: Create an Experiment Class

Parmest requires that the user create an `Experiment` class that builds an annotated Pyomo model denoting experiment outputs, unknown parameters, and measurement errors using Pyomo *Suffix* components.

- `m.experiment_outputs` maps the experiment output (or measurement) terms in the model (Pyomo *Param*, *Var*, or *Expression*) to their associated data values (float, int).
- `m.unknown_parameters` maps the model parameters to estimate (Pyomo *Param* or *Var*) to their component unique identifier (Pyomo *ComponentUID*) which is used to identify equivalent parameters across multiple experiments. Within parmest, any parameters that are to be estimated are converted to unfixed variables. Variables that are to be estimated are also unfixed.
- `m.measurement_error` maps the measurement error (float, int) of the experiment output, or measurement (Pyomo *Param*, *Var*, or *Expression*) defined in the model.

The experiment class has one required method:

- `get_labeled_model` which returns the labeled Pyomo model.

An example *`Experiment`* class is shown below.

Listing 3.2: RooneyBieglerExperiment class from the parmest example

```
class RooneyBieglerExperiment(Experiment):
    """Experiment class for Rooney-Biegler parameter estimation and design of␣
↪experiments.

    This class wraps the Rooney-Biegler exponential model for use with Pyomo's
    parmest (parameter estimation) and DoE (Design of Experiments) tools.
    """

    def __init__(self, data, measure_error=None, theta=None):
        """Initialize a Rooney-Biegler experiment instance.

        Parameters
        ----------
        data : pandas.Series or dict
            Experiment data containing 'hour' (time input) and 'y' (response) values.
        measure_error : float, optional
            Standard deviation of measurement error for the response variable.
            Required for DoE and covariance estimation. Default is None.
        theta : dict, optional
            Initial parameter values with 'asymptote' and 'rate_constant' keys.
            Default is None, which uses {'asymptote': 15, 'rate_constant': 0.5}.
        """
        self.data = data
        self.model = None
        self.measure_error = measure_error
        self.theta = theta

    def create_model(self):
        # rooney_biegler_model expects a dataframe
        if hasattr(self.data, 'to_frame'):
            # self.data is a pandas Series
            data_df = self.data.to_frame().transpose()
        else:
            # self.data is a dict
            data_df = pd.DataFrame([self.data])
        self.model = rooney_biegler_model(data_df, theta=self.theta)

    def label_model(self):

        m = self.model

        # Add experiment outputs as a suffix
        # Experiment outputs suffix is required for parmest
        m.experiment_outputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_outputs.update([(m.y, self.data['y'])])

        # Add unknown parameters as a suffix
```

(continues on next page)

(continued from previous page)

```
        # Unknown parameters suffix is required for both Pyomo.DoE and parmest
        m.unknown_parameters = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.unknown_parameters.update(
            (k, pyo.value(k)) for k in [m.asymptote, m.rate_constant]
        )

        # Add measurement error as a suffix
        # Measurement error suffix is required for Pyomo.DoE and
        #  `cov` estimation in parmest
        m.measurement_error = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.measurement_error.update([(m.y, self.measure_error)])

        # Add hour as an experiment input
        # Experiment inputs suffix is required for Pyomo.DoE
        m.experiment_inputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_inputs.update([(m.hour, self.data['hour'])])

    def get_labeled_model(self):
        if self.model is None:
            self.create_model()
            self.label_model()
        return self.model
```

Step 2: Load the Data and Create a List of Experiments

Load the experimental data into Python and create an instance of your *Experiment* class for each set of experimental data. In this example, each measurement of y is treated as a separate experiment.

```
>>> data = pd.DataFrame(data=[[1, 8.3], [2, 10.3], [3, 19.0], [4, 16.0], [5, 15.6], [7,␣
↪19.8]],
...                     columns=["hour", "y"])
>>> exp_list = []
>>> for i in range(data.shape[0]):
...     exp_list.append(RooneyBieglerExperiment(data.loc[i, :]))
```

Step 3: Create the Estimator Object

To use parmest, the user creates an *Estimator* object which includes the following methods:

theta_est	Parameter estimation using all scenarios in the data
cov_est	Covariance matrix calculation using all scenarios in the data
theta_est_bootstrap	Parameter estimation using bootstrap resampling of the data
theta_est_leaveNout	Parameter estimation where N data points are left out of each sample
objective_at_theta	Objective value for each theta, solving extensive form problem with fixed theta values.
confidence_region_test	Confidence region test to determine if theta values are within a rectangular, multivariate normal, or Gaussian kernel density distribution for a range of alpha values

continues on next page

Table 3.11 – continued from previous page

likelihood_ratio_test	Likelihood ratio test to identify theta values within a confidence region using the χ^2 distribution
leaveNout_bootstrap_test	Leave-N-out bootstrap test to compare theta values where N data points are left out to a bootstrap analysis using the remaining data, results indicate if theta is within a confidence region determined by the bootstrap analysis

Additional functions are available in parmest to plot results and fit distributions to theta values.

pairwise_plot	Plot pairwise relationship for theta values, and optionally alpha-level confidence intervals and objective value contours
grouped_boxplot	Plot a grouped boxplot to compare two datasets
grouped_violinplot	Plot a grouped violinplot to compare two datasets
fit_rect_dist	Fit an alpha-level rectangular distribution to theta values
fit_mvn_dist	Fit a multivariate normal distribution to theta values
fit_kde_dist	Fit a Gaussian kernel-density distribution to theta values

A *Estimator* object can be created using the following code. A description of the arguments are listed below.

```
>>> pest = parmest.Estimator(exp_list, obj_function="SSE")
```

Alternatively, the weighted sum of squared errors objective can be used.

```
>>> pest = parmest.Estimator(exp_list, obj_function="SSE_weighted")
```

Optionally, solver options can be supplied, e.g.,

```
>>> solver_options = {"max_iter": 6000}
>>> pest = parmest.Estimator(exp_list, obj_function="SSE", solver_options=solver_options)
```

Objective function

The `obj_function` keyword argument is used to specify the objective function to use for parameter estimation if the user has not implemented their own custom objective function. Parmest includes two built-in objective functions ("SSE" and "SSE_weighted") to compute the sum of squared errors between the `m.experiment_outputs` model values and data values. If the user wants to use an objective that is different from the built-in options, a custom objective function can be specified in the user's model, however, covariance matrix estimation (see covariancesection Section) is not supported for custom objective functions.

When declaring a custom objective function, parmest assumes the model has the structure of a two-stage stochastic programming problem so the objective function should be implemented using Pyomo Expressions for the first stage cost (named "FirstStageCost") and the second stage cost (named "SecondStageCost"). For parameter estimation problems the first stage cost is usually set to zero and the second stage cost is usually defined as the deviation between the model and the observations.

Step 4: Estimate the Parameters

After creating the *Estimator* object with the desired objective function, solve the parameter estimation problem by calling *theta_est*, e.g.,

```
>>> pest = parmest.Estimator(exp_list, obj_function="SSE")
>>> obj_val, theta_val = pest.theta_est()
```

Suggested Initialization Procedure for Parameter Estimation Problems

To check the quality of initial guess values provided for the fitted parameters, we suggest solving a square instance of the problem prior to solving the parameter estimation problem using the following steps:

1. Create `Estimator` object. To initialize the parameter estimation solve from the square problem solution, set optional argument `solver_options = {bound_push: 1e-8}`.

2. Call `objective_at_theta` with optional argument `(initialize_parmest_model=True)`. Different initial guess values for the fitted parameters can be provided using optional argument *theta_values* (**Pandas Dataframe**)

More Examples Beyond this Quick Guide

More detailed examples, such as parameter estimation of reaction kinetics are provided in the *Examples* Section.

Data Reconciliation

The optional argument `return_values` in `theta_est` can be used for data reconciliation or to return model values based on the specified objective.

For data reconciliation, the `m.unknown_parameters` is empty and the objective function is defined to minimize measurement to model error. Note that the model used for data reconciliation may differ from the model used for parameter estimation.

The functions `grouped_boxplot` or `grouped_violinplot` can be used to visually compare the original and reconciled data.

The following example from the reactor design subdirectory returns reconciled values for experiment outputs (*ca*, *cb*, *cc*, and *cd*) and then uses those values in parameter estimation (*k1*, *k2*, and *k3*).

```
# ____________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ____________________________________________________________________________________

import pyomo.environ as pyo
from pyomo.common.dependencies import numpy as np, pandas as pd
import pyomo.contrib.parmest.parmest as parmest
from pyomo.contrib.parmest.examples.reactor_design.reactor_design import (
    reactor_design_model,
    ReactorDesignExperiment,
)

np.random.seed(1234)


class ReactorDesignExperimentDataRec(ReactorDesignExperiment):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, data, data_std, experiment_number):

        super().__init__(data, experiment_number)
        self.data_std = data_std

    def create_model(self):

        self.model = m = reactor_design_model()
        m.caf.fixed = False

        return m

    def label_model(self):

        m = self.model

        # experiment outputs
        m.experiment_outputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_outputs.update(
            [
                (m.ca, self.data_i['ca']),
                (m.cb, self.data_i['cb']),
                (m.cc, self.data_i['cc']),
                (m.cd, self.data_i['cd']),
            ]
        )

        # experiment standard deviations
        m.experiment_outputs_std = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_outputs_std.update(
            [
                (m.ca, self.data_std['ca']),
                (m.cb, self.data_std['cb']),
                (m.cc, self.data_std['cc']),
                (m.cd, self.data_std['cd']),
            ]
        )

        # no unknowns (theta names)
        m.unknown_parameters = pyo.Suffix(direction=pyo.Suffix.LOCAL)

        return m


class ReactorDesignExperimentPostDataRec(ReactorDesignExperiment):

    def __init__(self, data, data_std, experiment_number):

        super().__init__(data, experiment_number)
        self.data_std = data_std

    def label_model(self):
```

(continues on next page)

(continued from previous page)

```
        m = super().label_model()

        # add experiment standard deviations
        m.experiment_outputs_std = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_outputs_std.update(
            [
                (m.ca, self.data_std['ca']),
                (m.cb, self.data_std['cb']),
                (m.cc, self.data_std['cc']),
                (m.cd, self.data_std['cd']),
            ]
        )

        return m


def generate_data():

    ### Generate data based on real sv, caf, ca, cb, cc, and cd
    sv_real = 1.05
    caf_real = 10000
    ca_real = 3458.4
    cb_real = 1060.8
    cc_real = 1683.9
    cd_real = 1898.5

    data = pd.DataFrame()
    ndata = 200
    # Normal distribution, mean = 3400, std = 500
    data["ca"] = 500 * np.random.randn(ndata) + 3400
    # Random distribution between 500 and 1500
    data["cb"] = np.random.rand(ndata) * 1000 + 500
    # Lognormal distribution
    data["cc"] = np.random.lognormal(np.log(1600), 0.25, ndata)
    # Triangular distribution between 1000 and 2000
    data["cd"] = np.random.triangular(1000, 1800, 3000, size=ndata)

    data["sv"] = sv_real
    data["caf"] = caf_real

    return data


def main():

    # Generate data
    data = generate_data()
    data_std = data.std()

    # Create an experiment list
    exp_list = []
```

(continues on next page)

(continued from previous page)

```
    for i in range(data.shape[0]):
        exp_list.append(ReactorDesignExperimentDataRec(data, data_std, i))

    # Define sum of squared error objective function for data rec
    def SSE_with_std(model):
        expr = sum(
            ((y - y_hat) / model.experiment_outputs_std[y]) ** 2
            for y, y_hat in model.experiment_outputs.items()
        )
        return expr

    ### Data reconciliation
    pest = parmest.Estimator(exp_list, obj_function=SSE_with_std)

    obj, theta, data_rec = pest.theta_est(return_values=["ca", "cb", "cc", "cd", "caf"])
    print(obj)
    print(theta)

    parmest.graphics.grouped_boxplot(
        data[["ca", "cb", "cc", "cd"]],
        data_rec[["ca", "cb", "cc", "cd"]],
        group_names=["Data", "Data Rec"],
    )

    ### Parameter estimation using reconciled data
    data_rec["sv"] = data["sv"]

    # make a new list of experiments using reconciled data
    exp_list = []
    for i in range(data_rec.shape[0]):
        exp_list.append(ReactorDesignExperimentPostDataRec(data_rec, data_std, i))

    pest = parmest.Estimator(exp_list, obj_function=SSE_with_std)
    obj, theta = pest.theta_est()
    print(obj)
    print(theta)

    theta_real = {"k1": 5.0 / 6.0, "k2": 5.0 / 3.0, "k3": 1.0 / 6000.0}
    print(theta_real)


if __name__ == "__main__":
    main()
```

The following example returns model values from a Pyomo Expression.

```
>>> import pandas as pd
>>> import pyomo.contrib.parmest.parmest as parmest
>>> from pyomo.contrib.parmest.examples.rooney_biegler.rooney_biegler import␣
↪RooneyBieglerExperiment

>>> # Generate data
```

(continues on next page)

(continued from previous page)

```
>>> data = pd.DataFrame(data=[[1,8.3],[2,10.3],[3,19.0],
...                           [4,16.0],[5,15.6],[7,19.8]],
...                     columns=['hour', 'y'])

>>> # Create an experiment list
>>> exp_list = []
>>> for i in range(data.shape[0]):
...     exp_list.append(RooneyBieglerExperiment(data.loc[i, :]))

>>> # Define objective
>>> def SSE(model):
...     expr = (model.experiment_outputs[model.y]
...             - model.y
...            ) ** 2
...     return expr

>>> pest = parmest.Estimator(exp_list, obj_function=SSE, solver_options=None)
>>> obj, theta, var_values = pest.theta_est(return_values=['response_function'])
>>> #print(var_values)
```

Uncertainty Quantification

The goal of parameter estimation (see *Parmest Quick Start Guide* Section) is to estimate unknown model parameters from experimental data. Uncertainty quantification then aims to characterize how close the parameter estimates are to their true (unknown) values. This parameter uncertainty can be computed using four methods in parmest: covariance matrix, likelihood ratio test, bootstrapping, and leave-N-out.

Covariance Matrix Estimation

The uncertainty in estimated model parameters can be quantified by computing the covariance matrix. The diagonal of this covariance matrix contains the variance of the estimated parameters, which is used to calculate their uncertainty. Assuming Gaussian independent and identically distributed measurement errors, the covariance matrix of the estimated parameters can be computed using the following methods which have been implemented in parmest.

1. Reduced Hessian Method

   When the objective function is the sum of squared errors (SSE) for homogeneous data, defined as $\text{SSE} = \sum_{i=1}^{n} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^{\text{T}} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})\right)$, the covariance matrix is:

   $$\boldsymbol{V}_{\boldsymbol{\theta}} = 2\sigma^2 \left(\frac{\partial^2 \text{SSE}}{\partial \boldsymbol{\theta}^2}\right)^{-1}_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}}$$

   Similarly, when the objective function is the weighted SSE (WSSE) for heterogeneous data, defined as $\text{WSSE} = \frac{1}{2} \sum_{i=1}^{n} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})\right)^{\text{T}} \boldsymbol{\Sigma}_{\boldsymbol{y}}^{-1} \left(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})\right)$, the covariance matrix is:

   $$\boldsymbol{V}_{\boldsymbol{\theta}} = \left(\frac{\partial^2 \text{WSSE}}{\partial \boldsymbol{\theta}^2}\right)^{-1}_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}}$$

   Where $\boldsymbol{V}_{\boldsymbol{\theta}}$ is the covariance matrix of the estimated parameters $\hat{\boldsymbol{\theta}} \in \mathbb{R}^p$, $\boldsymbol{y}_i \in \mathbb{R}^m$ are observations of the measured variables, $\boldsymbol{f}$ is the model function, $\boldsymbol{x}_i \in \mathbb{R}^q$ are the input variables, n is the number of experiments, $\boldsymbol{\Sigma}_{\boldsymbol{y}}$ is the measurement error covariance matrix, and σ^2 is the variance of the measurement error. When the standard deviation of the measurement error is not supplied by the user, parmest approximates σ^2 as: $\hat{\sigma}^2 = \frac{1}{n-p} \sum_{i=1}^{n} \boldsymbol{\varepsilon}_i(\boldsymbol{\theta})^{\text{T}} \boldsymbol{\varepsilon}_i(\boldsymbol{\theta})$, and $\boldsymbol{\varepsilon}_i \in \mathbb{R}^m$ are the residuals between the data and model for experiment i.

In parmest, this method computes the inverse of the Hessian by scaling the objective function (SSE or WSSE) with a constant probability factor, $\frac{1}{n}$.

2. Finite Difference Method

In this method, the covariance matrix, $\boldsymbol{V}_{\boldsymbol{\theta}}$, is computed by differentiating the Hessian, $\frac{\partial^2 \text{SSE}}{\partial \boldsymbol{\theta}^2}$ or $\frac{\partial^2 \text{WSSE}}{\partial \boldsymbol{\theta}^2}$, and applying Gauss-Newton approximation which results in:

$$\boldsymbol{V}_{\boldsymbol{\theta}} = \left(\sum_{i=1}^{n} \boldsymbol{G}_i^{\text{T}} \boldsymbol{\Sigma}_{\boldsymbol{y}}^{-1} \boldsymbol{G}_i \right)^{-1}$$

where

$$\boldsymbol{G}_i = \frac{\partial \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

This method uses central finite difference to compute the Jacobian matrix, $\boldsymbol{G}_i$, for experiment i.

$$\boldsymbol{G}_i[:,\, k] \approx \frac{\boldsymbol{f}(\boldsymbol{x}_i; \theta_k + \Delta\theta_k)|_{\hat{\boldsymbol{\theta}}} - \boldsymbol{f}(\boldsymbol{x}_i; \theta_k - \Delta\theta_k)|_{\hat{\boldsymbol{\theta}}}}{2\Delta\theta_k} \quad \forall \quad \theta_k \in [\theta_1, \cdots, \theta_p]$$

3. Automatic Differentiation Method

Similar to the finite difference method, the covariance matrix is calculated as:

$$\boldsymbol{V}_{\boldsymbol{\theta}} = \left(\sum_{i=1}^{n} \boldsymbol{G}_{\text{kaug},\, i}^{\text{T}} \boldsymbol{\Sigma}_{\boldsymbol{y}}^{-1} \boldsymbol{G}_{\text{kaug},\, i} \right)^{-1}$$

However, this method uses implicit differentiation and the model-optimality or Karush–Kuhn–Tucker (KKT) conditions to compute the Jacobian matrix, $\boldsymbol{G}_{\text{kaug},\, i}$, for experiment i.

$$\boldsymbol{G}_{\text{kaug},\, i} = \frac{\partial \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta})}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{x}_i}{\partial \boldsymbol{\theta}}$$

The covariance matrix calculation is only supported with the built-in objective functions “SSE” or “SSE_weighted”.

In parmest, the covariance matrix can be computed after creating the `Experiment` class, defining the `Estimator` object, and estimating the model parameters using `theta_est` (all these steps were addressed in the *Parmest Quick Start Guide* Section).

To estimate the covariance matrix, with the default method being “finite_difference”, call the `cov_est` function as follows:

```
>>> import pyomo.contrib.parmest.parmest as parmest
>>> pest = parmest.Estimator(exp_list, obj_function="SSE")
>>> obj_val, theta_val = pest.theta_est()
>>> cov = pest.cov_est()
```

Optionally, one of the three methods; “reduced_hessian”, “finite_difference”, and “automatic_differentiation_kaug” can be supplied for the covariance calculation, e.g.,

```
>>> pest = parmest.Estimator(exp_list, obj_function="SSE")
>>> obj_val, theta_val = pest.theta_est()
>>> cov_method = "reduced_hessian"
>>> cov = pest.cov_est(method=cov_method)
```

Bootstrapping

Bootstrapping is a non-intrusive method that uses resampling to approximate the variance of the parameter estimates. By repeatedly fitting the model to resampled datasets, the parameter uncertainty (e.g., variance or covariance) can be computed without relying on strong distributional assumptions. This method is summarized as follows:

1. Step 0: Define the input data

   Given input data: $[y_1, \ldots, y_n]$

2. Step 1: Generate B artificial datasets through sampling

   Sample with replacement from the original data to create B bootstrap datasets:

$$\left\{y_1^{(1)}, \ldots, y_n^{(1)}\right\}, \ldots, \left\{y_1^{(B)}, \ldots, y_n^{(B)}\right\}$$

3. Step 2: Compute the estimator of the parameters

   Fit the model to each bootstrap dataset to obtain parameter estimates:

$$\left\{\hat{\theta}^{(1)}, \ldots, \hat{\theta}^{(B)}\right\}$$

4. Step 3: Compute the approximate variance of the parameter estimates

   The variability across bootstrap estimates approximates the estimator variance:

$$\mathrm{Var}(\hat{\theta}) = \frac{1}{B}\sum_{j=1}^{B}\left(\hat{\theta}^{(j)}\right)^2 - \left(\frac{1}{B}\sum_{j=1}^{B}\hat{\theta}^{(j)}\right)^2$$

Note

The example code for this method will soon be provided.

Likelihood Ratio Test

The likelihood ratio test is a non-intrusive method that compares how well two parameter sets explain the observed data: an unconstrained set and a constrained set defined by the null hypothesis. It is commonly used to assess whether restricting parameters significantly degrades model fit. This method is summarized as follows:

1. Step 1: State the hypothesis

   Null hypothesis: $\theta \in \Theta_0$ vs. Alternative hypothesis: $\theta \notin \Theta_0$

2. Step 2: Define the test statistic

   The test statistic is the ratio of the maximum likelihood under the full parameter space to that under the constrained space:

$$\lambda_n = \frac{\sup_{\theta\in\Theta} L(\theta; y_1, \ldots, y_n)}{\sup_{\theta\in\Theta_0} L(\theta; y_1, \ldots, y_n)}$$

3. Step 3: Define the decision rule

   Reject the null hypothesis if:

$$\lambda_n > c_\alpha$$

Equivalently, using maximum likelihood estimates:

$$\lambda_n = \frac{L(\hat{\theta}; y_1, \ldots, y_n)}{L(\hat{\theta}_0; y_1, \ldots, y_n)} > c_\alpha$$

Note

The example code for this method will soon be provided.

Leave-N-Out

Note

Detailed descriptions and example code for this method will be added in a future update.

Scenario Creation

In addition to model-based parameter estimation, parmest can create scenarios for use in optimization under uncertainty. To do this, one first creates an `Estimator` object, then a `ScenarioCreator` object, which has methods to add `ParmestScen` scenario objects to a `ScenarioSet` object, which can write them to a csv file or output them via an iterator method.

This example is in the semibatch subdirectory of the examples directory in the file `scenario_example.py`. It creates a csv file with scenarios that correspond one-to-one with the experiments used as input data. It also creates a few scenarios using the bootstrap methods and outputs prints the scenarios to the screen, accessing them via the `ScensItator` a `print`

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

import json
from os.path import join, abspath, dirname
import pyomo.contrib.parmest.parmest as parmest
from pyomo.contrib.parmest.examples.semibatch.semibatch import SemiBatchExperiment
import pyomo.contrib.parmest.scenariocreator as sc


def main():

    # Data: list of dictionaries
    data = []
    file_dirname = dirname(abspath(str(__file__)))
    for exp_num in range(10):
        fname = join(file_dirname, 'exp' + str(exp_num + 1) + '.out')
        with open(fname, 'r') as infile:
```

(continues on next page)

(continued from previous page)

```
            d = json.load(infile)
            data.append(d)

    # Create an experiment list
    exp_list = []
    for i in range(len(data)):
        exp_list.append(SemiBatchExperiment(data[i]))

    # View one model
    # exp0_model = exp_list[0].get_labeled_model()
    # exp0_model.pprint()

    pest = parmest.Estimator(exp_list)

    scenmaker = sc.ScenarioCreator(pest, "ipopt")

    # Make one scenario per experiment and write to a csv file
    output_file = "scenarios.csv"
    experimentscens = sc.ScenarioSet("Experiments")
    scenmaker.ScenariosFromExperiments(experimentscens)
    experimentscens.write_csv(output_file)

    # Use the bootstrap to make 3 scenarios and print
    bootscens = sc.ScenarioSet("Bootstrap")
    scenmaker.ScenariosFromBootstrap(bootscens, 3)
    for s in bootscens.ScensIterator():
        print("{}, {}".format(s.name, s.probability))
        for n, v in s.ThetaVals.items():
            print("    {}={}".format(n, v))


if __name__ == "__main__":
    main()
```

Note

This example may produce an error message if your version of Ipopt is not based on a good linear solver.

Graphics

parmest includes the following functions to help visualize results:

- *grouped_boxplot*
- *grouped_violinplot*
- *pairwise_plot*

Grouped boxplots and violinplots are used to compare datasets, generally before and after data reconciliation. Pairwise plots are used to visualize results from parameter estimation and include a histogram of each parameter along the diagonal and a scatter plot for each pair of parameters in the upper and lower sections. The pairwise plot can also include the following optional information:

- A single value for each theta (generally theta* from parameter estimation).

- Confidence intervals for rectangular, multivariate normal, and/or Gaussian kernel density estimate distributions at a specified level (i.e. 0.8). For plots with more than 2 parameters, theta* is used to extract a slice of the confidence region for each pairwise plot.
- Filled contour lines for objective values at a specified level (i.e. 0.8). For plots with more than 2 parameters, theta* is used to extract a slice of the contour lines for each pairwise plot.

The following examples were generated using the reactor design example. Fig. 3.3 uses output from data reconciliation, Fig. 3.4 uses output from the bootstrap analysis, and Fig. 3.5 uses output from the likelihood ratio test.

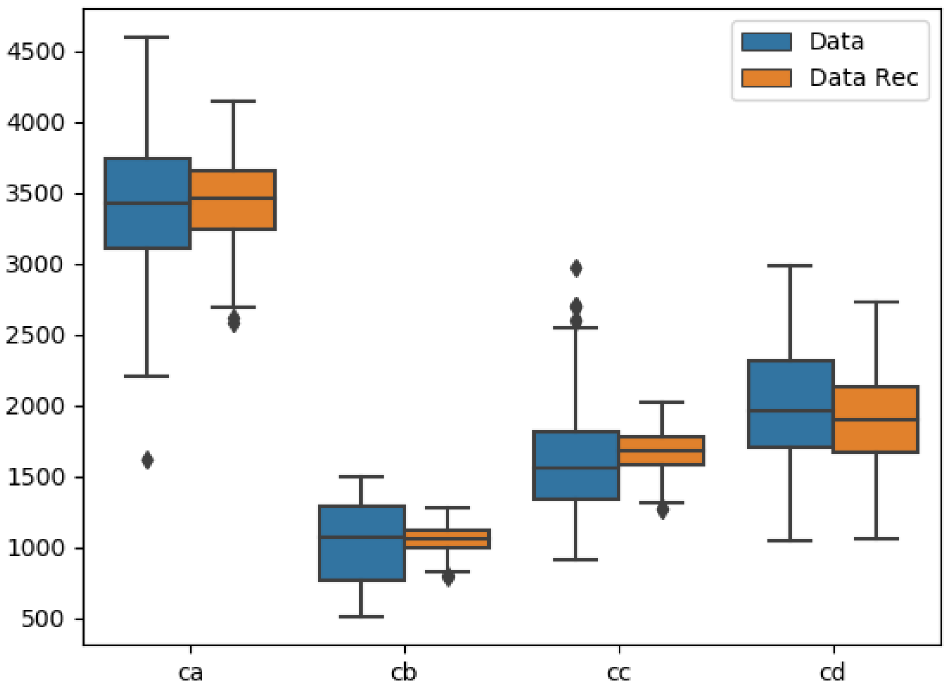


Fig. 3.3: Grouped boxplot showing data before and after data reconciliation.

Examples

Examples can be found in *pyomo/contrib/parmest/examples* and include:

- Reactor design example [PyomoBookIII]
- Semibatch example [AM00]
- Rooney Biegler example [RB01]

Each example includes a Python file that contains the Pyomo model and a Python file to run parameter estimation.

Additional use cases include:

- Data reconciliation (reactor design example)
- Parameter estimation using data with duplicate sensors and time-series data (reactor design example)
- Parameter estimation using mpi4py, the example saves results to a file for later analysis/graphics (semibatch example)

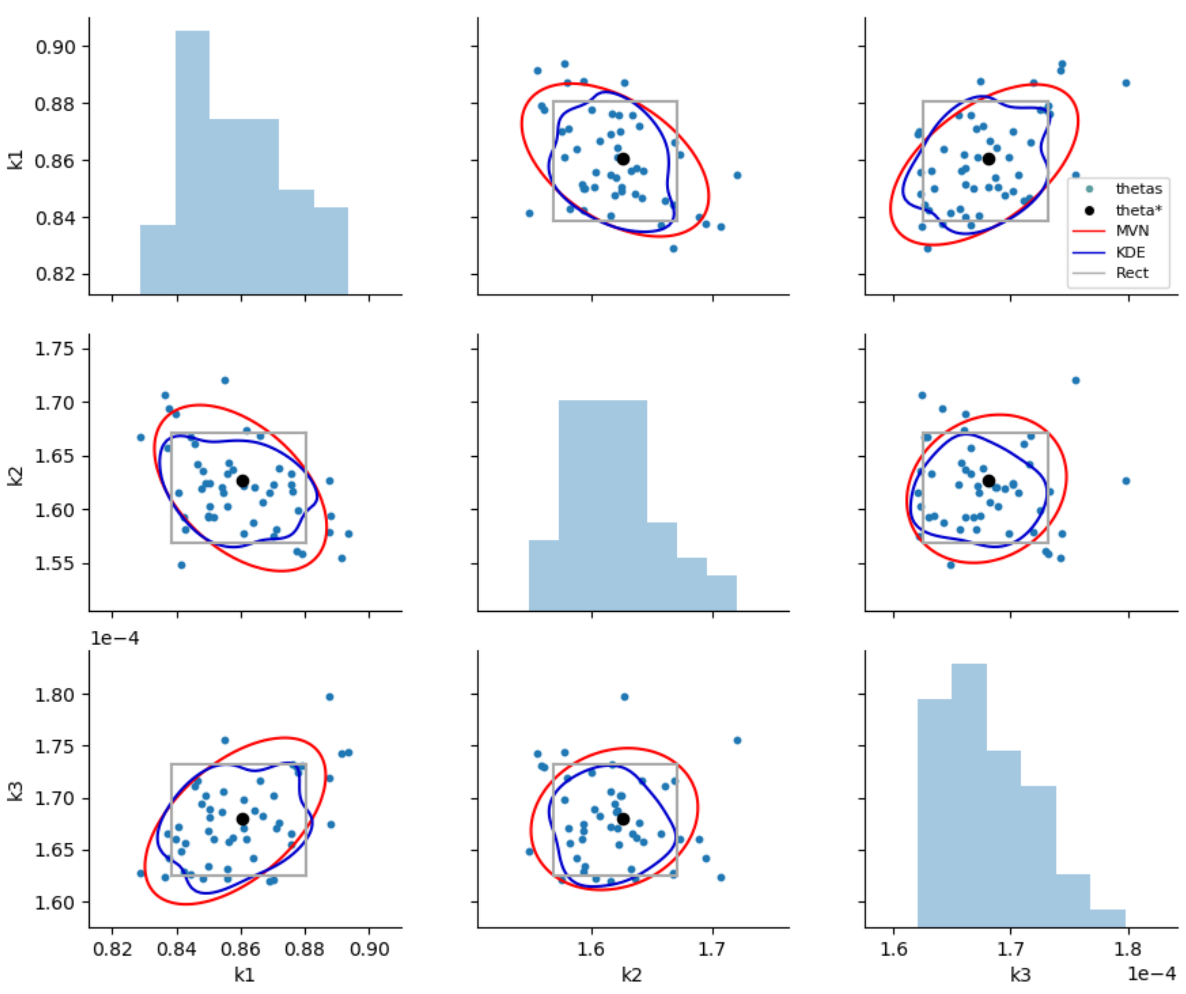


Fig. 3.4: Pairwise bootstrap plot with rectangular, multivariate normal and kernel density estimation confidence region.

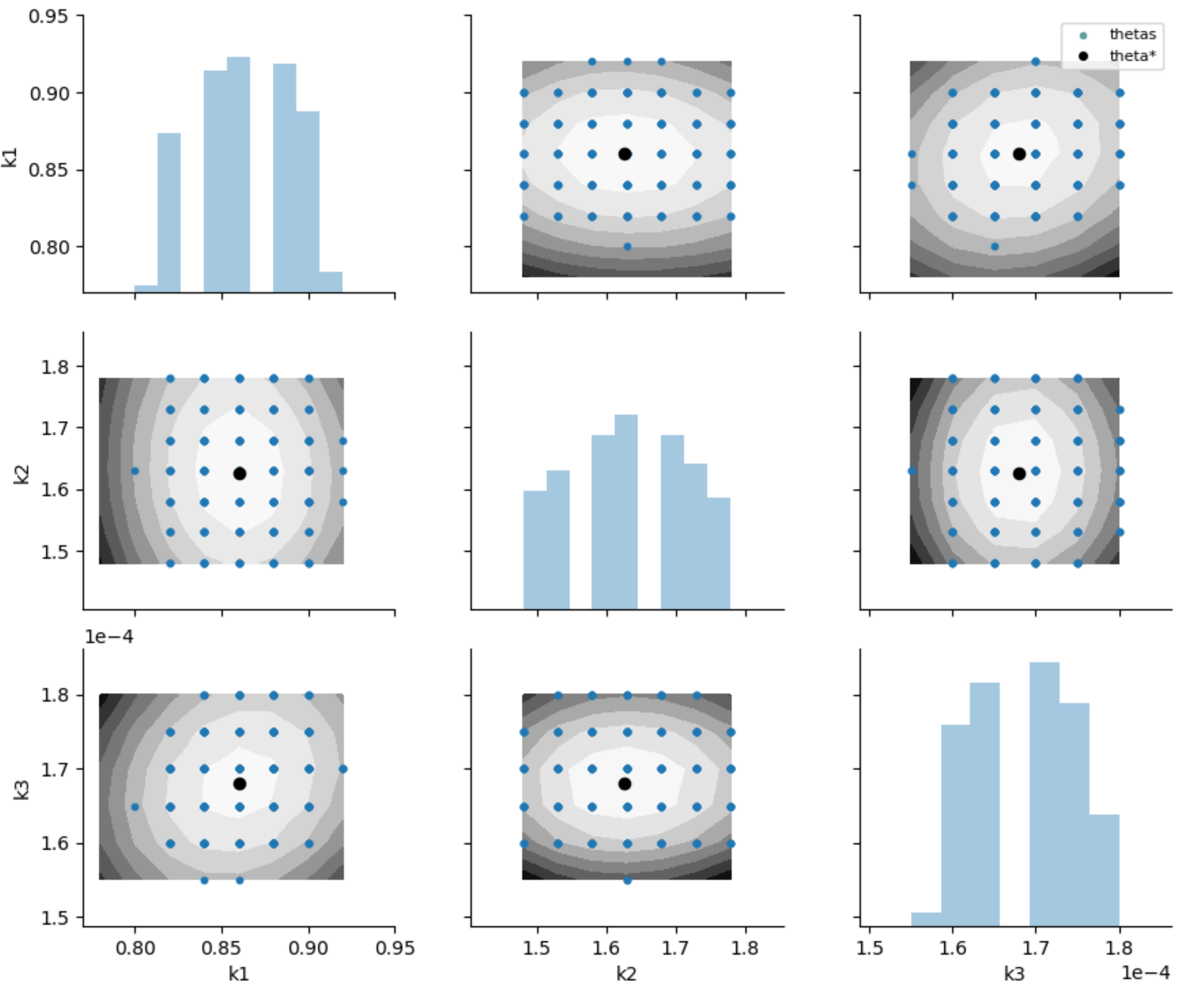


Fig. 3.5: Pairwise likelihood ratio plot with contours of the objective and points that lie within an alpha confidence region.

The example below uses the reactor design example. The file **reactor_design.py** includes a function which returns an populated instance of the Pyomo model. Note that the model is defined to maximize *cb* and that *k1*, *k2*, and *k3* are fixed. The _main_ program is included for easy testing of the model declaration.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________
"""
Continuously stirred tank reactor model, based on
pyomo/examples/doc/pyomobook/nonlinear-ch/react_design/ReactorDesign.py
"""

from pyomo.common.dependencies import pandas as pd
import pyomo.environ as pyo
import pyomo.contrib.parmest.parmest as parmest
from pyomo.contrib.parmest.experiment import Experiment


def reactor_design_model():

    # Create the concrete model
    model = pyo.ConcreteModel()

    # Rate constants, make unknown parameters variables
    model.k1 = pyo.Var(initialize=5.0 / 6.0, within=pyo.PositiveReals)  # min^-1
    model.k2 = pyo.Var(initialize=5.0 / 3.0, within=pyo.PositiveReals)  # min^-1
    model.k3 = pyo.Var(
        initialize=1.0 / 6000.0, within=pyo.PositiveReals
    )  # m^3/(gmol min)

    # Inlet concentration of A, gmol/m^3
    model.caf = pyo.Param(initialize=10000, within=pyo.PositiveReals, mutable=True)

    # Space velocity (flowrate/volume)
    model.sv = pyo.Param(initialize=1.0, within=pyo.PositiveReals, mutable=True)

    # Outlet concentration of each component
    model.ca = pyo.Var(initialize=5000.0, within=pyo.PositiveReals)
    model.cb = pyo.Var(initialize=2000.0, within=pyo.PositiveReals)
    model.cc = pyo.Var(initialize=2000.0, within=pyo.PositiveReals)
    model.cd = pyo.Var(initialize=1000.0, within=pyo.PositiveReals)

    # Objective
    model.obj = pyo.Objective(expr=model.cb, sense=pyo.maximize)

    # Constraints
    model.ca_bal = pyo.Constraint(
        expr=(
```

(continues on next page)

(continued from previous page)

```
            0
            == model.sv * model.caf
            - model.sv * model.ca
            - model.k1 * model.ca
            - 2.0 * model.k3 * model.ca**2.0
        )
    )

    model.cb_bal = pyo.Constraint(
        expr=(0 == -model.sv * model.cb + model.k1 * model.ca - model.k2 * model.cb)
    )

    model.cc_bal = pyo.Constraint(
        expr=(0 == -model.sv * model.cc + model.k2 * model.cb)
    )

    model.cd_bal = pyo.Constraint(
        expr=(0 == -model.sv * model.cd + model.k3 * model.ca**2.0)
    )

    return model


class ReactorDesignExperiment(Experiment):

    def __init__(self, data, experiment_number):
        self.data = data
        self.experiment_number = experiment_number
        self.data_i = data.loc[experiment_number, :]
        self.model = None

    def create_model(self):
        self.model = m = reactor_design_model()
        return m

    def finalize_model(self):
        m = self.model

        # Experiment inputs values
        m.sv = self.data_i['sv']
        m.caf = self.data_i['caf']

        # Experiment output values
        m.ca = self.data_i['ca']
        m.cb = self.data_i['cb']
        m.cc = self.data_i['cc']
        m.cd = self.data_i['cd']

        return m

    def label_model(self):
        m = self.model
```

(continues on next page)

(continued from previous page)

```
        m.experiment_outputs = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.experiment_outputs.update(
            [
                (m.ca, self.data_i['ca']),
                (m.cb, self.data_i['cb']),
                (m.cc, self.data_i['cc']),
                (m.cd, self.data_i['cd']),
            ]
        )

        m.unknown_parameters = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.unknown_parameters.update(
            (k, pyo.ComponentUID(k)) for k in [m.k1, m.k2, m.k3]
        )

        m.measurement_error = pyo.Suffix(direction=pyo.Suffix.LOCAL)
        m.measurement_error.update(
            [(m.ca, None), (m.cb, None), (m.cc, None), (m.cd, None)]
        )

        return m

    def get_labeled_model(self):
        m = self.create_model()
        m = self.finalize_model()
        m = self.label_model()

        return m


def main():

    # For a range of sv values, return ca, cb, cc, and cd
    results = []
    sv_values = [1.0 + v * 0.05 for v in range(1, 20)]
    caf = 10000
    for sv in sv_values:

        # make model
        model = reactor_design_model()

        # add caf, sv
        model.caf = caf
        model.sv = sv

        # solve model
        solver = pyo.SolverFactory("ipopt")
        solver.solve(model)

        # save results
        results.append([sv, caf, model.ca(), model.cb(), model.cc(), model.cd()])
```

(continues on next page)

(continued from previous page)

```
    results = pd.DataFrame(results, columns=["sv", "caf", "ca", "cb", "cc", "cd"])
    print(results)


if __name__ == "__main__":
    main()
```

The file **parameter_estimation_example.py** uses parmest to estimate values of *k1*, *k2*, and *k3* by minimizing the sum of squared error between model and observed values of *ca*, *cb*, *cc*, and *cd*. Additional example files use parmest to run parameter estimation with bootstrap resampling and perform a likelihood ratio test over a range of theta values.

```
# ____________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ____________________________________________________________________________________

from pyomo.common.dependencies import pandas as pd
from os.path import join, abspath, dirname
import pyomo.contrib.parmest.parmest as parmest
from pyomo.contrib.parmest.examples.reactor_design.reactor_design import (
    ReactorDesignExperiment,
)


def main():

    # Read in data
    file_dirname = dirname(abspath(str(__file__)))
    file_name = abspath(join(file_dirname, "reactor_data.csv"))
    data = pd.read_csv(file_name)

    # Create an experiment list
    exp_list = []
    for i in range(data.shape[0]):
        exp_list.append(ReactorDesignExperiment(data, i))

    # View one model
    # exp0_model = exp_list[0].get_labeled_model()
    # exp0_model.pprint()

    pest = parmest.Estimator(exp_list, obj_function='SSE')

    # Parameter estimation
    obj, theta = pest.theta_est()
    print("Least squares objective value:", obj)
    print("Estimated parameters (theta):\n")
    print(theta)
```

(continues on next page)

(continued from previous page)

```
    # Compute the covariance matrix at the estimated parameter
    cov = pest.cov_est()
    print("Covariance matrix:\n")
    print(cov)


if __name__ == "__main__":
    main()
```

The semibatch and Rooney Biegler examples are defined in a similar manner.

Parallel Implementation

Parallel implementation in parmest is **preliminary**. To run parmest in parallel, you need the mpi4py Python package and a *compatible* MPI installation. If you do NOT have mpi4py or a MPI installation, parmest still works (you should not get MPI import errors).

For example, the following command can be used to run the semibatch model in parallel:

```
mpiexec -n 4 python parallel_example.py
```

The file **parallel_example.py** is shown below. Results are saved to file for later analysis.

```
# ____________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ____________________________________________________________________________________

"""
The following script can be used to run semibatch parameter estimation in
parallel and save results to files for later analysis and graphics.
Example command: mpiexec -n 4 python parallel_example.py
"""

from pyomo.common.dependencies import numpy as np, pandas as pd
from itertools import product
from os.path import join, abspath, dirname
import pyomo.contrib.parmest.parmest as parmest
from pyomo.contrib.parmest.examples.semibatch.semibatch import generate_model


def main():
    # Vars to estimate
    theta_names = ['k1', 'k2', 'E1', 'E2']

    # Data, list of json file names
    data = []
    file_dirname = dirname(abspath(str(__file__)))
```

(continues on next page)

(continued from previous page)

```
    for exp_num in range(10):
        file_name = abspath(join(file_dirname, 'exp' + str(exp_num + 1) + '.out'))
        data.append(file_name)

    # Note, the model already includes a 'SecondStageCost' expression
    # for sum of squared error that will be used in parameter estimation

    pest = parmest.Estimator(generate_model, data, theta_names)

    ### Parameter estimation with bootstrap resampling
    bootstrap_theta = pest.theta_est_bootstrap(100)
    bootstrap_theta.to_csv('bootstrap_theta.csv')

    ### Compute objective at theta for likelihood ratio test
    k1 = np.arange(4, 24, 3)
    k2 = np.arange(40, 160, 40)
    E1 = np.arange(29000, 32000, 500)
    E2 = np.arange(38000, 42000, 500)
    theta_vals = pd.DataFrame(list(product(k1, k2, E1, E2)), columns=theta_names)

    obj_at_theta = pest.objective_at_theta(theta_vals)
    obj_at_theta.to_csv('obj_at_theta.csv')


if __name__ == "__main__":
    main()
```

Installation

The mpi4py Python package should be installed using conda. The following installation instructions were tested on a Mac with Python 3.5.

Create a conda environment and install mpi4py using the following commands:

```
conda create -n parmest-parallel python=3.5
source activate parmest-parallel
conda install -c conda-forge mpi4py
```

This should install libgfortran, mpi, mpi4py, and openmpi.

To verify proper installation, create a Python file with the following:

```
from mpi4py import MPI
import time
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print('Rank = ',rank)
time.sleep(10)
```

Save the file as test_mpi.py and run the following command:

```
time mpiexec -n 4 python test_mpi.py
time python test_mpi.py
```

The first one should be faster and should start 4 instances of Python.

Objective Options

Note

Detailed descriptions and example code for the objective options in parmest will be added in a future update.

Estimability Analysis

After estimating the model parameters with their associated uncertainty, as demonstrated in the *Parmest Quick Start Guide* and *Uncertainty Quantification* Sections, estimability analysis is required to identify parameters that cannot be reliably estimated from the available data due to limitations in the mathematical model structure. If such parameters exist, the model may need to be reformulated, replaced with an alternative structure, or augmented with additional prior information. In parmest, estimability analysis can be performed using eigen-decomposition of the parameter covariance matrix, profile likelihood methods, or multi-start initialization routines.

Eigen-decomposition

The estimability of model parameters can be analyzed through eigen-decomposition of the covariance matrix obtained from parameter estimation. This covariance matrix quantifies parameter uncertainty and captures both parameter variances and correlations. Eigen-decomposition of this matrix identifies principal directions in parameter space along which uncertainty is largest or smallest. These directions provide insight into parameter identifiability and reveal combinations of parameters that are either structurally identifiable or non-identifiable based on the underlying model formulation.

Note

Detailed descriptions and example code for this method will be added in a future update.

Profile Likelihood

Profile likelihood analysis evaluates parameter estimability by systematically varying one parameter while re-optimizing the remaining parameters to maintain consistency with the model and observed data. This approach is closely related to likelihood ratio–based uncertainty quantification and provides a robust characterization of practical identifiability through the shape of the likelihood surface. In addition, it can reveal structural non-identifiability when flat or unbounded profiles indicate parameter combinations that are not uniquely determined by the model formulation, particularly in nonlinear systems.

Note

Detailed descriptions and example code for this method will be added in a future update.

Multi-start Initialization

Multi-start initialization assesses parameter estimability by exploring a range of initial guesses. Because parameter estimation problems are often nonlinear and may exhibit multiple local minima, different initializations can lead to different parameter estimates. By solving the estimation problem from multiple starting points, one can evaluate the robustness of the solution and identify potential issues related to non-convexity or non-identifiability. Consistent convergence to a unique solution across initializations suggests that the parameters are structurally identifiable within the model formulation, whereas sensitivity to initialization indicates potential estimability issues.

Note

Detailed descriptions and example code for this method will be added in a future update.

API

parmest

enum pyomo.contrib.parmest.parmest.**CovarianceMethod**(*value*)

Bases: Enum

finite_difference = 'finite_difference'

automatic_differentiation_kaug = 'automatic_differentiation_kaug'

reduced_hessian = 'reduced_hessian'

enum pyomo.contrib.parmest.parmest.**ObjectiveType**(*value*)

Bases: Enum

SSE = 'SSE'

SSE_weighted = 'SSE_weighted'

enum pyomo.contrib.parmest.parmest.**RegularizationType**(*value*)

Bases: Enum

L2 = 'L2'

class pyomo.contrib.parmest.parmest.**Estimator**(*experiment_list*, *obj_function=None*, *tee=False*, *diagnostic_mode=False*, *solver_options=None*, *regularization=None*, *prior_FIM=None*, *theta_ref=None*, *regularization_weight=None*)

Bases: object

Parameter estimation class

Parameters

- **experiment_list** (*list of Experiments*) – A list of experiment objects which creates one labeled model for each experiment
- **obj_function** (*string or function (optional)*) – Built-in objective ("SSE" or "SSE_weighted") or custom function used to formulate parameter estimation objective. If no function is specified, the model is used "as is" and should be defined with a "FirstStageCost" and "SecondStageCost" expression that are used to build an objective. Default is None.
- **tee** (*bool, optional*) – If True, print the solver output to the screen. Default is False.
- **diagnostic_mode** (*bool, optional*) – If True, print diagnostics from the solver. Default is False.
- **solver_options** (*dict, optional*) – Provides options to the solver (also the name of an attribute). Default is None.
- **regularization** (*string, optional*) – Built-in regularization type ("L2"). If no regularization is specified, no regularization term is added to the objective. Default is None.

- **prior_FIM** (*pd.DataFrame, optional*) – Prior Fisher Information Matrix from previous experimental design to be added to the FIM of the current experiments for regularization. The prior_FIM should be a square matrix with parameter names as both row and column labels.
- **theta_ref** (*dict, optional*) – Reference parameter values used in regularization. If None, defaults to the current parameter values in the model.
- **regularization_weight** (*float, optional*) – Weighting factor for the regularization term. Used with `regularization="L2"`. Default is 1.0.

`confidence_region_test`(*theta_values*, *distribution*, *alphas*, *test_theta_values=None*, *seed=None*)

Confidence region test to determine if theta values are within a rectangular, multivariate normal, or Gaussian kernel density distribution for a range of alpha values

Parameters

- **theta_values** (*pd.DataFrame, columns = theta_names*) – Theta values used to generate a confidence region (generally returned by theta_est_bootstrap)
- **distribution** (*string*) – Statistical distribution used to define a confidence region, options = 'MVN' for multivariate_normal, 'KDE' for gaussian_kde, and 'Rect' for rectangular.
- **alphas** (*list*) – List of alpha values used to determine if theta values are inside or outside the region.
- **test_theta_values** (*pd.Series or pd.DataFrame, keys/columns = theta_names, optional*) – Additional theta values that are compared to the confidence region to determine if they are inside or outside.

Returns

- **training_results** (*pd.DataFrame*) – Theta value used to generate the confidence region along with True (inside) or False (outside) for each alpha
- **test_results** (*pd.DataFrame*) – If test_theta_values is not None, returns test theta value along with True (inside) or False (outside) for each alpha

`cov_est`(*method='finite_difference'*, *solver='ipopt'*, *step=0.001*)

Covariance matrix calculation using all scenarios in the data

Parameters

- **method** (*str, optional*) – Covariance calculation method. Options - 'finite_difference', 'reduced_hessian', and 'automatic_differentiation_kaug'. Default is 'finite_difference'
- **solver** (*str, optional*) – Solver name, e.g., 'ipopt'. Default is 'ipopt'
- **step** (*float, optional*) – Float used for relative perturbation of the parameters, e.g., step=0.02 is a 2% perturbation. Default is 1e-3

Returns

cov – Covariance matrix of the estimated parameters

Return type

pd.DataFrame

`leaveNout_bootstrap_test`(*lNo*, *lNo_samples*, *bootstrap_samples*, *distribution*, *alphas*, *seed=None*)

Leave-N-out bootstrap test to compare theta values where N data points are left out to a bootstrap analysis using the remaining data, results indicate if theta is within a confidence region determined by the bootstrap analysis

Parameters

- **lNo** (*int*) – Number of data points to leave out for parameter estimation
- **lNo_samples** (*int*) – Leave-N-out sample size. If lNo_samples=None, the maximum number of combinations will be used
- **bootstrap_samples** (*int:*) – Bootstrap sample size
- **distribution** (*string*) – Statistical distribution used to define a confidence region, options = 'MVN' for multivariate_normal, 'KDE' for gaussian_kde, and 'Rect' for rectangular.
- **alphas** (*list*) – List of alpha values used to determine if theta values are inside or outside the region.
- **seed** (*int or None, optional*) – Random seed

Returns

- *List of tuples with one entry per lNo_sample*
- ** The first item in each tuple is the list of N samples that are left* – out.
- ** The second item in each tuple is a DataFrame of theta estimated using* – the N samples.
- ** The third item in each tuple is a DataFrame containing results from* – the bootstrap analysis using the remaining samples.
- *For each DataFrame a column is added for each value of alpha which*
- *indicates if the theta estimate is in (True) or out (False) of the*
- *alpha region for a given distribution (based on the bootstrap results)*

likelihood_ratio_test(*obj_at_theta*, *obj_value*, *alphas*, *return_thresholds=False*)

Likelihood ratio test to identify theta values within a confidence region using the χ^2 distribution

Parameters

- **obj_at_theta** (*pd.DataFrame, columns = theta_names + 'obj'*) – Objective values for each theta value (returned by objective_at_theta)
- **obj_value** (*int or float*) – Objective value from parameter estimation using all data
- **alphas** (*list*) – List of alpha values to use in the chi2 test
- **return_thresholds** (*bool, optional*) – Return the threshold value for each alpha. Default is False.

Returns

- **LR** (*pd.DataFrame*) – Objective values for each theta value along with True or False for each alpha
- **thresholds** (*pd.Series*) – If return_threshold = True, the thresholds are also returned.

objective_at_theta(*theta_values=None*, *initialize_parmest_model=False*)

Objective value for each theta, solving extensive form problem with fixed theta values.

theta_est(*solver='ef_ipopt'*, *return_values=[]*, *calc_cov=NOTSET*, *cov_n=NOTSET*)

Parameter estimation using all scenarios in the data

Parameters

- **solver** (*str, optional*) – Currently only "ef_ipopt" is supported. Default is "ef_ipopt".
- **return_values** (*list, optional*) – List of Variable names, used to return values from the model for data reconciliation

- **calc_cov** (*boolean, optional*) – DEPRECATED.

  If True, calculate and return the covariance matrix (only for "ef_ipopt" solver). Default is NOTSET

- **cov_n** (*int, optional*) – DEPRECATED.

  If calc_cov=True, then the user needs to supply the number of datapoints that are used in the objective function. Default is NOTSET

Returns

- **obj_val** (*float*) – The objective function value
- **theta_vals** (*pd.Series*) – Estimated values for theta
- **var_values** (*pd.DataFrame*) – Variable values for each variable name in return_values (only for solver='ef_ipopt')

theta_est_bootstrap(*bootstrap_samples*, *samplesize=None*, *replacement=True*, *seed=None*, *return_samples=False*)

Parameter estimation using bootstrap resampling of the data

Parameters

- **bootstrap_samples** (*int*) – Number of bootstrap samples to draw from the data
- **samplesize** (*int or None, optional*) – Size of each bootstrap sample. If samplesize=None, samplesize will be set to the number of samples in the data
- **replacement** (*bool, optional*) – Sample with or without replacement. Default is True.
- **seed** (*int or None, optional*) – Random seed
- **return_samples** (*bool, optional*) – Return a list of sample numbers used in each bootstrap estimation. Default is False.

Returns

bootstrap_theta – Theta values for each sample and (if return_samples = True) the sample numbers used in each estimation

Return type

pd.DataFrame

theta_est_leaveNout(*lNo*, *lNo_samples=None*, *seed=None*, *return_samples=False*)

Parameter estimation where N data points are left out of each sample

Parameters

- **lNo** (*int*) – Number of data points to leave out for parameter estimation
- **lNo_samples** (*int*) – Number of leave-N-out samples. If lNo_samples=None, the maximum number of combinations will be used
- **seed** (*int or None, optional*) – Random seed
- **return_samples** (*bool, optional*) – Return a list of sample numbers that were left out. Default is False.

Returns

lNo_theta – Theta values for each sample and (if return_samples = True) the sample numbers left out of each estimation

Return type

pd.DataFrame

pyomo.contrib.parmest.parmest.**L2_regularized_objective**(*model*, *prior_FIM*, *theta_ref=None*, *regularization_weight=1.0*, *obj_function=<function SSE>*)

Calculates objective + regularization_weight*(theta - theta_ref)^T * prior_FIM * (theta - theta_ref) using label-based alignment for safety and subsets for efficiency.

Parameters

- **model** (ConcreteModel) – Annotated Pyomo model
- **prior_FIM** (*pd.DataFrame*) – Prior Fisher Information Matrix from previous experimental design
- **theta_ref** (*dict, optional*) – Reference parameter values used in regularization. If None, defaults to the current parameter values in the model.
- **regularization_weight** (*float, optional*) – Weighting factor for the regularization term. Default is 1.0.
- **obj_function** (*callable, optional*) – Built-in objective function selected by the user, e.g., *SSE*. Default is *SSE*.

Returns

expr – Expression representing the L2-regularized objective

Return type

Pyomo expression

pyomo.contrib.parmest.parmest.**SSE**(*model*)

Returns an expression that is used to compute the sum of squared errors ('SSE') objective, assuming Gaussian i.i.d. errors

Parameters

model (ConcreteModel) – Annotated Pyomo model

pyomo.contrib.parmest.parmest.**SSE_weighted**(*model*)

Returns an expression that is used to compute the 'SSE_weighted' objective, assuming Gaussian i.i.d. errors, with measurement error standard deviation defined in the annotated Pyomo model

Parameters

model (ConcreteModel) – Annotated Pyomo model

pyomo.contrib.parmest.parmest.**compute_covariance_matrix**(*experiment_list*, *method*, *obj_function*, *theta_vals*, *step*, *solver*, *tee*, *estimated_var=None*, *prior_FIM=None*, *regularization_weight=1.0*, *solver_options=None*)

Computes the covariance matrix of the estimated parameters using 'finite_difference' or 'automatic_differentiation_kaug' methods

Parameters

- **experiment_list** (*list*) – List of Experiment class objects containing the Pyomo model for the different experimental conditions
- **method** (*str*) – Covariance calculation method specified by the user, e.g., 'finite_difference'
- **obj_function** (*callable*) – Built-in objective function selected by the user, e.g., *SSE*
- **theta_vals** (*dict*) – Dictionary containing the estimates of the unknown parameters

- **step** (*float*) – Float used for relative perturbation of the parameters, e.g., step=0.02 is a 2% perturbation
- **solver** (*str*) – Solver name specified by the user, e.g., 'ipopt'
- **tee** (*bool*) – Boolean solver option to be passed for verbose output
- **estimated_var** (*float, optional*) – Value of the estimated variance of the measurement error in cases where the user does not supply the measurement error standard deviation
- **prior_FIM** (*pd.DataFrame, optional*) – Prior Fisher Information Matrix from previous experimental design to be added to the FIM of the current experiments for covariance estimation. The prior_FIM should be a square matrix with parameter names as both row and column labels.
- **regularization_weight** (*float, optional*) – Weighting factor for the regularization term. Default is 1.0.
- **solver_options** (*dict, optional*) – Dictionary of solver options to be passed for the finite difference calculations

Returns
cov – Covariance matrix of the estimated parameters

Return type
pd.DataFrame

`pyomo.contrib.parmest.parmest.`**`ef_nonants`**`(`*ef*`)`

`pyomo.contrib.parmest.parmest.`**`group_data`**`(`*data*, *groupby_column_name*, *use_mean=None*`)`

DEPRECATED.

Group data by scenario

Parameters

- **data** (*DataFrame*) – Data
- **groupby_column_name** (*strings*) – Name of data column which contains scenario numbers
- **use_mean** (*list of column names or None, optional*) – Name of data columns which should be reduced to a single value per scenario by taking the mean

Returns

- **grouped_data** (*list of dictionaries*) – Grouped data
- *.. deprecated:: 6.7.2* – This function (group_data) has been deprecated and may be removed in a future release.

`pyomo.contrib.parmest.parmest.`**`validate_experiment_outputs`**`(`*output_vars*`)`

Checks that: 1. All variables in the *experiment_outputs* attribute have the same number of indices 2. All variables in the *experiment_outputs* attribute share the same index set

Parameters
output_vars (`pyomo.core.base.suffix.Suffix`) – Experiment output variables

scenariocreator

`class pyomo.contrib.parmest.scenariocreator.`**`ParmestScen`**`(`*name*, *ThetaVals*, *probability*`)`

Bases: `object`

A little container for scenarios; the Args are the attributes.

Parameters

- **name** (*str*) – name for reporting; might be “”
- **ThetaVals** (*dict*) – ThetaVals[name]=val
- **probability** (*float*) – probability of occurrence “near” these ThetaVals

class pyomo.contrib.parmest.scenariocreator.**ScenarioCreator**(*pest*, *solvername*)

Bases: object

Create scenarios from parmest.

Parameters

- **pest** (Estimator) – the parmest object
- **solvername** (*str*) – name of the solver (e.g. “ipopt”)

ScenariosFromBootstrap(*addtoSet*, *numtomake*, *seed=None*)

Creates new self.Scenarios list using the experiments only.

Parameters

- **addtoSet** (ScenarioSet) – the scenarios will be added to this set
- **numtomake** (*int*) – number of scenarios to create

ScenariosFromExperiments(*addtoSet*)

Creates new self.Scenarios list using the experiments only.

Parameters
addtoSet (ScenarioSet) – the scenarios will be added to this set

Returns
a ScenarioSet

class pyomo.contrib.parmest.scenariocreator.**ScenarioSet**(*name*)

Bases: object

Class to hold scenario sets

Args: name (str): name of the set (might be “”)

ScenarioNumber(*scennum*)

Returns the scenario with the given, zero-based number

ScensIterator()

Usage: for scenario in ScensIterator()

addone(*scen*)

Add a scenario to the set

Parameters
scen (ParmestScen) – the scenario to add

append_bootstrap(*bootstrap_theta*)

Append a bootstrap theta df to the scenario set; equally likely

Parameters
bootstrap_theta (*dataframe*) – created by the bootstrap

Note: this can be cleaned up a lot with the list becomes a df,
which is why I put it in the ScenarioSet class.

write_csv(*filename*)

write a csv file with the scenarios in the set

Parameters

filename (*str*) – full path and full name of file

graphics

pyomo.contrib.parmest.graphics.**fit_kde_dist**(*theta_values*, *seed=None*)

Fit a Gaussian kernel-density distribution to theta values

Parameters

theta_values (*DataFrame*) – Theta values, columns = variable names

Return type

scipy.stats.gaussian_kde distribution

pyomo.contrib.parmest.graphics.**fit_mvn_dist**(*theta_values*, *seed=None*)

Fit a multivariate normal distribution to theta values

Parameters

theta_values (*DataFrame*) – Theta values, columns = variable names

Return type

scipy.stats.multivariate_normal distribution

pyomo.contrib.parmest.graphics.**fit_rect_dist**(*theta_values*, *alpha*)

Fit an alpha-level rectangular distribution to theta values

Parameters

- **theta_values** (*DataFrame*) – Theta values, columns = variable names
- **alpha** (*float, optional*) – Confidence interval value

Return type

tuple containing lower bound and upper bound for each variable

pyomo.contrib.parmest.graphics.**grouped_boxplot**(*data1*, *data2*, *normalize=False*, *group_names=['data1', 'data2']*, *filename=None*)

Plot a grouped boxplot to compare two datasets

The datasets can be normalized by the median and standard deviation of data1.

Parameters

- **data1** (*DataFrame*) – Data set, columns = variable names
- **data2** (*DataFrame*) – Data set, columns = variable names
- **normalize** (*bool, optional*) – Normalize both datasets by the median and standard deviation of data1
- **group_names** (*list, optional*) – Names used in the legend
- **filename** (*string, optional*) – Filename used to save the figure

pyomo.contrib.parmest.graphics.**grouped_violinplot**(*data1*, *data2*, *normalize=False*, *group_names=['data1', 'data2']*, *filename=None*)

Plot a grouped violinplot to compare two datasets

The datasets can be normalized by the median and standard deviation of data1.

Parameters

- **data1** (*DataFrame*) – Data set, columns = variable names
- **data2** (*DataFrame*) – Data set, columns = variable names
- **normalize** (*bool, optional*) – Normalize both datasets by the median and standard deviation of data1
- **group_names** (*list, optional*) – Names used in the legend
- **filename** (*string, optional*) – Filename used to save the figure

pyomo.contrib.parmest.graphics.**pairwise_plot**(*theta_values*, *theta_star=None*, *alpha=None*, *distributions=[]*, *axis_limits=None*, *title=None*, *add_obj_contour=True*, *add_legend=True*, *filename=None*, *seed=None*)

Plot pairwise relationship for theta values, and optionally alpha-level confidence intervals and objective value contours

Parameters

- **theta_values** (*DataFrame or tuple*) –
  - If theta_values is a DataFrame, then it contains one column for each theta variable and (optionally) an objective value column ('obj') and columns that contains Boolean results from confidence interval tests (labeled using the alpha value). Each row is a sample.
    - Theta variables can be computed from `theta_est_bootstrap`, `theta_est_leaveNout`, and `leaveNout_bootstrap_test`.
    - The objective value can be computed using the `likelihood_ratio_test`.
    - Results from confidence interval tests can be computed using the `leaveNout_bootstrap_test`, `likelihood_ratio_test`, and `confidence_region_test`.
  - If theta_values is a tuple, then it contains a mean, covariance, and number of samples (mean, cov, n) where mean is a dictionary or Series (indexed by variable name), covariance is a DataFrame (indexed by variable name, one column per variable name), and n is an integer. The mean and covariance are used to create a multivariate normal sample of n theta values. The covariance can be computed using `theta_est(calc_cov=True)`.
- **theta_star** (*dict or Series, optional*) – Estimated value of theta. The dictionary or Series is indexed by variable name. Theta_star is used to slice higher dimensional contour intervals in 2D
- **alpha** (*float, optional*) – Confidence interval value, if an alpha value is given and the distributions list is empty, the data will be filtered by True/False values using the column name whose value equals alpha (see results from `leaveNout_bootstrap_test`, `likelihood_ratio_test`, and `confidence_region_test`)
- **distributions** (*list of strings, optional*) – Statistical distribution used to define a confidence region, options = 'MVN' for multivariate_normal, 'KDE' for gaussian_kde, and 'Rect' for rectangular. Confidence interval is a 2D slice, using linear interpolation at theta_star.
- **axis_limits** (*dict, optional*) – Axis limits in the format {variable: [min, max]}
- **title** (*string, optional*) – Plot title
- **add_obj_contour** (*bool, optional*) – Add a contour plot using the column 'obj' in theta_values. Contour plot is a 2D slice, using linear interpolation at theta_star.

- **add_legend** (*bool, optional*) – Add a legend to the plot
- **filename** (*string, optional*) – Filename used to save the figure
- **seed** (*int, optional*) – Random seed used to generate theta values if theta_values is a tuple. If None, the seed is not set.

3.4.8 Sensitivity Toolbox

The sensitivity toolbox provides a Pyomo interface to sIPOPT and k_aug to very quickly compute approximate solutions to nonlinear programs with a small perturbation in model parameters.

See the sIPOPT documentation or the following paper for additional details:

H. Pirnay, R. Lopez-Negrete, and L.T. Biegler, Optimal Sensitivity based on IPOPT, Math. Prog. Comp., 4(4):307–331, 2012.

The details of *k_aug* can be found in the following link:

David Thierry (2020). k_aug, https://github.com/dthierry/k_aug

Using the Sensitivity Toolbox

We will start with a motivating example:

$$\begin{array}{ll} \min_{x_1,x_2,x_3} & x_1^2 + x_2^2 + x_3^2 \\ \text{s.t.} & 6x_1 + 3x_2 + 2x_3 - p_1 = 0 \\ & p_2 x_1 + x_2 - x_3 - 1 = 0 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Here x_1, x_2, and x_3 are the decision variables while p_1 and p_2 are parameters. At first, let's consider $p_1 = 4.5$ and $p_2 = 1.0$. Below is the model implemented in Pyomo.

```
# Import Pyomo and the sensitivity toolbox
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.sensitivity_toolbox.sens import sensitivity_calculation

# Create a concrete model
>>> m = pyo.ConcreteModel()

# Define the variables with bounds and initial values
>>> m.x1 = pyo.Var(initialize = 0.15, within=pyo.NonNegativeReals)
>>> m.x2 = pyo.Var(initialize = 0.15, within=pyo.NonNegativeReals)
>>> m.x3 = pyo.Var(initialize = 0.0, within=pyo.NonNegativeReals)

# Define the parameters
>>> m.eta1 = pyo.Param(initialize=4.5,mutable=True)
>>> m.eta2 = pyo.Param(initialize=1.0,mutable=True)

# Define the constraints and objective
>>> m.const1 = pyo.Constraint(expr=6*m.x1+3*m.x2+2*m.x3-m.eta1 ==0)
>>> m.const2 = pyo.Constraint(expr=m.eta2*m.x1+m.x2-m.x3-1 ==0)
>>> m.cost = pyo.Objective(expr=m.x1**2+m.x2**2+m.x3**2)
```

The solution of this optimization problem is $x_1^* = 0.5$, $x_2^* = 0.5$, and $x_3^* = 0.0$. But what if we change the parameter values to $\hat{p}_1 = 4.0$ and $\hat{p}_2 = 1.0$? Is there a quick way to approximate the new solution $\hat{x}_1^*$, $\hat{x}_2^*$, and $\hat{x}_3^*$? Yes! This is the main functionality of sIPOPT and k_aug.

Next we define the perturbed parameter values $\hat{p}_1$ and $\hat{p}_2$:

```
>>> m.perturbed_eta1 = pyo.Param(initialize = 4.0)
>>> m.perturbed_eta2 = pyo.Param(initialize = 1.0)
```

And finally we call sIPOPT or k_aug:

```
>>> m_sipopt = sensitivity_calculation('sipopt', m, [m.eta1, m.eta2], [m.perturbed_eta1,␣
↪m.perturbed_eta2], tee=False)
>>> m_kaug_dsdp = sensitivity_calculation('k_aug', m, [m.eta1, m.eta2], [m.perturbed_
↪eta1, m.perturbed_eta2], tee=False)
```

The first argument specifies the method, either 'sipopt' or 'k_aug'. The second argument is the Pyomo model. The third argument is a list of the original parameters. The fourth argument is a list of the perturbed parameters. It's important that these two lists are the same length and in the same order.

First, we can inspect the initial point:

```
>>> print("eta1 = %0.3f" % m.eta1())
eta1 = 4.500

>>> print("eta2 = %0.3f" % m.eta2())
eta2 = 1.000

# Initial point (not feasible):
>>> print("Objective = %0.3f" % m.cost())
Objective = 0.045

>>> print("x1 = %0.3f" % m.x1())
x1 = 0.150

>>> print("x2 = %0.3f" % m.x2())
x2 = 0.150

>>> print("x3 = %0.3f" % m.x3())
x3 = 0.000
```

Next, we inspect the solution x_1^*, x_2^*, and x_3^*:

```
# Solution with the original parameter values:
>>> print("Objective = %0.3f" % m_sipopt.cost())
Objective = 0.500

>>> print("x1 = %0.3f" % m_sipopt.x1())
x1 = 0.500

>>> print("x2 = %0.3f" % m_sipopt.x2())
x2 = 0.500

>>> print("x3 = %0.3f" % m_sipopt.x3())
x3 = 0.000
```

Note that k_aug does not save the solution with the original parameter values. Finally, we inspect the approximate solution $\hat{x}_1^*$, $\hat{x}_2^*$, and $\hat{x}_3^*$:

```
# *sIPOPT*
# New parameter values:
>>> print("eta1 = %0.3f" %m_sipopt.perturbed_eta1())
eta1 = 4.000

>>> print("eta2 = %0.3f" % m_sipopt.perturbed_eta2())
eta2 = 1.000

# (Approximate) solution with the new parameter values:
>>> x1 = m_sipopt.sens_sol_state_1[m_sipopt.x1]
>>> x2 = m_sipopt.sens_sol_state_1[m_sipopt.x2]
>>> x3 = m_sipopt.sens_sol_state_1[m_sipopt.x3]
>>> print("Objective = %0.3f" % (x1**2 + x2**2 + x3**2))
Objective = 0.556

>>> print("x1 = %0.3f" % x1)
x1 = 0.333

>>> print("x2 = %0.3f" % x2)
x2 = 0.667

>>> print("x3 = %0.3f" % x3)
x3 = 0.000

# *k_aug*
# New parameter values:
>>> print("eta1 = %0.3f" %m_kaug_dsdp.perturbed_eta1())
eta1 = 4.000

>>> print("eta2 = %0.3f" % m_kaug_dsdp.perturbed_eta2())
eta2 = 1.000

# (Approximate) solution with the new parameter values:
>>> x1 = m_kaug_dsdp.x1()
>>> x2 = m_kaug_dsdp.x2()
>>> x3 = m_kaug_dsdp.x3()
>>> print("Objective = %0.3f" % (x1**2 + x2**2 + x3**2))
Objective = 0.556

>>> print("x1 = %0.3f" % x1)
x1 = 0.333

>>> print("x2 = %0.3f" % x2)
x2 = 0.667

>>> print("x3 = %0.3f" % x3)
x3 = 0.000
```

Installing sIPOPT and k_aug

The sensitivity toolbox requires either sIPOPT or k_aug to be installed and available in your system PATH. See the sIPOPT and k_aug documentation for detailed instructions:

- https://coin-or.github.io/Ipopt/INSTALL.html
- https://coin-or.github.io/Ipopt/SPECIALS.html#SIPOPT
- https://coin-or.github.io/coinbrew/
- https://github.com/dthierry/k_aug

Note

If you get an error that `ipopt_sens` or `k_aug` and `dot_sens` cannot be found, double check your installation and make sure the build directories containing the executables were added to your system PATH.

Sensitivity Toolbox Interface

`pyomo.contrib.sensitivity_toolbox.sens.`**`sensitivity_calculation`**(*method*, *instance*, *paramList*, *perturbList*, *cloneModel=True*, *tee=False*, *keepfiles=False*, *solver_options=None*)

This function accepts a Pyomo ConcreteModel, a list of parameters, and their corresponding perturbation list. The model is then augmented with dummy constraints required to call sipopt or k_aug to get an approximation of the perturbed solution.

Parameters

- **method** (*string*) – 'sipopt' or 'k_aug'
- **instance** (Block) – pyomo block or model object
- **paramSubList** (*list*) – list of mutable parameters or fixed variables
- **perturbList** (*list*) – list of perturbed parameter values
- **cloneModel** (*bool, optional*) – indicator to clone the model. If set to False, the original model will be altered
- **tee** (*bool, optional*) – indicator to stream solver log
- **keepfiles** (*bool, optional*) – preserve solver interface files
- **solver_options** (*dict, optional*) – Provides options to the solver (also the name of an attribute)

Return type

The model that was manipulated by the sensitivity interface

3.4.9 NLP Initialization

Warning

This package lives in *pyomo.devel*. APIs, options, and behavior may change without notice.

The initialization module within `pyomo.devel.initialization` is intended to provide methods to help initialize nonconvex nonlinear programs (NLPs). The goal is to increase the chance of finding a local minimizer (i.e., decrease

the chance of getting stuck at a point that locally minimizes infeasibility). If you are already able to solve your problem with a local NLP solver, these tools will not help you. Example usage is shown below.

```
import pyomo.environ as pyo
import pyomo.devel.initialization as ini
from pyomo.contrib.solver.common.factory import SolverFactory


def build_model():
    m = pyo.ConcreteModel()
    m.x = pyo.Var(bounds=(-20, 20), initialize=-3.6)
    m.c = pyo.Constraint(expr=(m.x + 7) * (m.x + 5) * (m.x - 4) + 200 == 0)
    return m


def lp_init_ex():
    m = build_model()
    nlp_solver = SolverFactory('ipopt')
    lp_solver = SolverFactory('highs')
    results = ini.initialize_with_LP_approximation(
        nlp=m, nlp_solver=nlp_solver, lp_solver=lp_solver, seed=0
    )

    return results.solution_status, m.x.value


def pwl_init_ex():
    m = build_model()
    nlp_solver = SolverFactory('ipopt')
    mip_solver = SolverFactory('highs')
    results = ini.initialize_with_piecewise_linear_approximation(
        nlp=m, nlp_solver=nlp_solver, mip_solver=mip_solver
    )

    return results.solution_status, m.x.value


def global_init_ex():
    m = build_model()
    nlp_solver = SolverFactory('ipopt')
    global_solver = SolverFactory('scip_direct')
    results = ini.initialize_with_global_opt(
        nlp=m, nlp_solver=nlp_solver, global_solver=global_solver
    )

    return results.solution_status, m.x.value


if __name__ == '__main__':
    # stat, x = lp_init_ex()
    # stat, x = pwl_init_ex()
    stat, x = global_init_ex()
    print(stat, round(x, 4))
```

This example shows the three different initialization methods currently available. Each method tries to find a good starting point for the NLP solver and then attempts to solve the problem with the given NLP solver.

Note

Currently, this module only works with solvers from *pyomo.contrib.solver*.

Initialization Methods

There are currently three initialization methods available.

Note

Not all of the methods described below require all nonlinear variables to be bounded. However, all of the methods will perform better if all nonlinear variables are bounded (the tighter the bounds, the better).

Method `initialize_with_global_opt`

This method uses an MINLP solver to try to find a feasible solution. We adjust the solver parameters so that the solver will stop as soon as any feasible solution is found. We then initialize the NLP solver at that feasible solution. Many MINLP solvers will default to a very large time limit, so it can be useful to specify a time limit before calling *initialize_with_global_opt*:

```
import pyomo.environ as pyo
from pyomo.contrib.solver.common.factory import SolverFactory

global_solver = SolverFactory('scip_direct')
global_solver.config.time_limit = 600  # 10 minutes
# now call initialize_with_global_opt
```

This method currently works with the following solver interfaces for MINLP solvers:

- SCIP (*direct* and `persistent`)
- *Gurobi MINLP*

Advantages

- Currently, this is the method that is most likely to succeed in finding a feasible solution.
- Does not strictly require variable bounds

Disadvantages

- This method will only work if the model is completely algebraic. It will not work with external functions.

Method `initialize_with_piecewise_linear_approximation`

This method builds a piecewise linear (PWL) approximation of the model, solves it, and initializes the NLP solver at the solution. If the NLP solver does not converge, then the PWL approximation will be refined by adding additional "segments". This is repeated until either a feasible solution is found or the iteration limit is reached.

This method does not currently work as well as `global_opt`, but it does have a great deal of potential. We expect future versions of this method to perform significantly better.

Advantages

- Does not require an MINLP solver
- Future versions will work with external functions

Disadvantages

- Current implementation can be slow
- Requires all nonlinear variables to be bounded

Method `initialize_with_lp_approximation`

This method is similar to the PWL approximation method, but it builds an LP approximation instead and does not do any refinement. Another distinction is that the LP approximation uses a linear least-squares fit, so the approximation may not equal the original function at the variable bounds. This also means that variable bounds are not strictly necessary, though they do help improve the approximation.

Advantages

- Fast
- Future versions will work with external functions
- Does not strictly require variable bounds
- Does not require an MINLP or even an MILP solver

Disadvantages

- This method only attempts to initialize the problem once. If it does not succeed, it is done.

3.5 Modeling Utilities

3.5.1 “Flattening” a Pyomo model

pyomo.dae.flatten	A module for "flattening" the components in a block-hierarchical model with respect to common indexing sets

Motivation

The `pyomo.dae.flatten` module was originally developed to assist with dynamic optimization. A very common operation in dynamic or multi-period optimization is to initialize all time-indexed variables to their values at a specific time point. However, for variables indexed by time and arbitrary other indexing sets, this is difficult to do in a way that does does not depend on the variable we are initializing. Things get worse when we consider that a time index can exist on a parent block rather than the component itself.

By “reshaping” time-indexed variables in a model into references indexed only by time, the `flatten_dae_components` function allows us to perform operations that depend on knowledge of time indices without knowing anything about the variables that we are operating on.

This “flattened representation” of a model turns out to be useful for dynamic optimization in a variety of other contexts. Examples include constructing a tracking objective function and plotting results. This representation is also useful in cases where we want to preserve indexing along more than one set, as in PDE-constrained optimization.

The flatten_components_along_sets function allows partitioning components while preserving multiple indexing sets. In such a case, time and space-indexed data for a given variable is useful for purposes such as initialization, visualization, and stability analysis.

API reference

pyomo.dae.flatten.slice_component_along_sets(...)	This function generates all possible slices of the provided component along the provided sets.
pyomo.dae.flatten.flatten_components_along_sets(m, ...)	This function iterates over components (recursively) contained in a block and partitions their data objects into components indexed only by the specified sets.
pyomo.dae.flatten.flatten_dae_components(...)	Partitions components into ComponentData and Components indexed only by the provided set.

pyomo.dae.flatten.**slice_component_along_sets**(*component*, *sets*, *context_slice=None*, *normalize=None*)

This function generates all possible slices of the provided component along the provided sets. That is, it will iterate over the component's other indexing sets and, for each index, yield a slice along the sets specified in the call signature.

Parameters

- **component** (Component) – The component whose slices will be yielded
- **sets** (ComponentSet) – ComponentSet of Pyomo sets that will be sliced along
- **context_slice** (IndexedComponent_slice) – If provided, instead of creating a new slice, we will extend this one with appropriate getattr and getitem calls.
- **normalize** (*Bool*) – If False, the returned index (from the product of "other sets") is not normalized, regardless of the value of normalize_index.flatten. This is necessary to use this index with _fill_indices.

Yields

tuple – The first entry is the index in the product of "other sets" corresponding to the slice, and the second entry is the slice at that index.

pyomo.dae.flatten.**flatten_components_along_sets**(*m*, *sets*, *ctype*, *indices=None*, *active=None*)

This function iterates over components (recursively) contained in a block and partitions their data objects into components indexed only by the specified sets.

Parameters

- **m** (BlockData) – Block whose components (and their sub-components) will be partitioned
- **sets** (*Tuple of Pyomo Sets*) – Sets to be sliced. Returned components will be indexed by some combination of these sets, if at all.
- **ctype** (*Subclass of* Component) – Type of component to identify and partition
- **indices** (*Iterable or* ComponentMap) – Indices of sets to use when descending into subblocks. If an iterable is provided, the order corresponds to the order in sets. If a ComponentMap is provided, the keys must be in sets.
- **active** (*Bool or None*) – If not None, this is a boolean flag used to filter component objects by their active status. A reference-to-slice is returned if any data object defined by the slice matches this flag.

Returns

The first entry is a list of tuples of Pyomo Sets. The second is a list of lists of Components, indexed by the corresponding sets in the first list. If the components are unindexed, ComponentData are

returned and the tuple of sets contains only UnindexedComponent_set. If the components are indexed, they are references-to-slices.

Return type
List of tuples of Sets, list of lists of Components

pyomo.dae.flatten.**flatten_dae_components**(*model*, *time*, *ctype*, *indices=None*, *active=None*)

Partitions components into ComponentData and Components indexed only by the provided set.

Parameters

- **model** (BlockData) – Block whose components are partitioned
- **time** (Set) – Indexing by this set (and only this set) will be preserved in the returned components.
- **ctype** (*Subclass of* Component) – Type of component to identify, partition, and return
- **indices** (*Tuple or* ComponentMap) – Contains the index of the specified set to be used when descending into blocks
- **active** (*Bool or None*) – If provided, used as a filter to only return components with the specified active flag. A reference-to-slice is returned if any data object defined by the slice matches this flag.

Returns
The first list contains ComponentData for all components not indexed by the provided set. The second contains references-to -slices for all components indexed by the provided set.

Return type
List of *ComponentData*, list of *Component*

What does it mean to flatten a model?

When accessing components in a block-structured model, we use `component_objects` or `component_data_objects` to access all objects of a specific `Component` or `ComponentData` type. The generated objects may be thought of as a "flattened" representation of the model, as they may be accessed without any knowledge of the model's block structure. These methods are very useful, but it is still challenging to use them to access specific components. Specifically, we often want to access "all components indexed by some set," or "all component data at a particular index of this set." In addition, we often want to generate the components in a block that is indexed by our particular set, as these components may be thought of as "implicitly indexed" by this set. The `pyomo.dae.flatten` module aims to address this use case by providing utilities to generate all components indexed, explicitly or implicitly, by user-provided sets.

When we say "flatten a model," we mean "recursively generate all components in the model," where a component can be indexed only by user-specified indexing sets (or is not indexed at all).

Data structures

The components returned are either `ComponentData` objects, for components not indexed by any of the provided sets, or references-to-slices, for components indexed, explicitly or implicitly, by the provided sets. Slices are necessary as they can encode "implicit indexing" – where a component is contained in an indexed block. It is natural to return references to these slices, so they may be accessed and manipulated like any other component.

Citation

If you use the `pyomo.dae.flatten` module in your research, we would appreciate you citing the following paper, which gives more detail about the motivation for and examples of using this functionality.

```
@article{parker2023mpc,
title = {Model predictive control simulations with block-hierarchical differential-
↪algebraic process models},
journal = {Journal of Process Control},
volume = {132},
pages = {103113},
year = {2023},
issn = {0959-1524},
doi = {https://doi.org/10.1016/j.jprocont.2023.103113},
url = {https://www.sciencedirect.com/science/article/pii/S0959152423002007},
author = {Robert B. Parker and Bethany L. Nicholson and John D. Siirola and Lorenz T.␣
↪Biegler},
}
```

3.5.2 Latex Printing

Pyomo models can be printed to a LaTeX compatible format using the `pyomo.contrib.latex_printer.latex_printer` function:

`pyomo.contrib.latex_printer.latex_printer.`**`latex_printer`**(*pyomo_component*, *latex_component_map=None*, *ostream=None*, *use_equation_environment=False*, *explicit_set_summation=False*, *throw_templatization_error=False*)

This function produces a string that can be rendered as LaTeX

Prints a Pyomo component (Block, Model, Objective, Constraint, or Expression) to a LaTeX compatible string

Parameters

- **pyomo_component** (`BlockData` *or* `Model` *or* `Objective` *or* `Constraint` *or* `Expression`) – The Pyomo component to be printed
- **latex_component_map** (`pyomo.common.collections.component_map.ComponentMap`) – A map keyed by Pyomo component, values become the LaTeX representation in the printer
- **ostream** (*`io.TextIOWrapper`* *or* *`io.StringIO`* *or* *`str`*) – The object to print the LaTeX string to. Can be an open file object, string I/O object, or a string for a filename to write to
- **use_equation_environment** (*`bool`*) –

  If False, the equation/aligned construction is used to create a single
  LaTeX equation. If True, then the align environment is used in LaTeX and each constraint and objective will be given an individual equation number
- **explicit_set_summation** (*`bool`*) – If False, all sums will be done over 'index in set' or similar. If True, sums will be done over 'i=1' to 'N' or similar if the set is a continuous set
- **throw_templatization_error** (*`bool`*) – Option to throw an error on templatization failure rather than printing each constraint individually, useful for very large models

Returns
A LaTeX string of the pyomo_component

Return type
str

Note: If operating in a Jupyter Notebook, it may be helpful to use:

```
from IPython.display import display, Math
display(Math(latex_printer(m))
```

Examples

A Model

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer

>>> m = pyo.ConcreteModel(name = 'basicFormulation')
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.z = pyo.Var()
>>> m.c = pyo.Param(initialize=1.0, mutable=True)
>>> m.objective    = pyo.Objective( expr = m.x + m.y + m.z )
>>> m.constraint_1 = pyo.Constraint(expr = m.x**2 + m.y**2.0 - m.z**2.0 <= m.c )

>>> pstr = latex_printer(m)
```

A Constraint

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer

>>> m = pyo.ConcreteModel(name = 'basicFormulation')
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()

>>> m.constraint_1 = pyo.Constraint(expr = m.x**2 + m.y**2 <= 1.0)

>>> pstr = latex_printer(m.constraint_1)
```

A Constraint with Set Summation

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer
>>> m = pyo.ConcreteModel(name='basicFormulation')
>>> m.I = pyo.Set(initialize=[1, 2, 3, 4, 5])
>>> m.v = pyo.Var(m.I)

>>> def ruleMaker(m): return sum(m.v[i] for i in m.I) <= 0

>>> m.constraint = pyo.Constraint(rule=ruleMaker)

>>> pstr = latex_printer(m.constraint)
```

Using a ComponentMap to Specify Names

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer
>>> from pyomo.common.collections.component_map import ComponentMap

>>> m = pyo.ConcreteModel(name='basicFormulation')
>>> m.I = pyo.Set(initialize=[1, 2, 3, 4, 5])
>>> m.v = pyo.Var(m.I)

>>> def ruleMaker(m):  return sum(m.v[i] for i in m.I) <= 0

>>> m.constraint = pyo.Constraint(rule=ruleMaker)

>>> lcm = ComponentMap()
>>> lcm[m.v] = 'x'
>>> lcm[m.I] = ['\\mathcal{A}',['j','k']]

>>> pstr = latex_printer(m.constraint, latex_component_map=lcm)
```

An Expression

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer

>>> m = pyo.ConcreteModel(name = 'basicFormulation')
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()

>>> m.expression_1 = pyo.Expression(expr = m.x**2 + m.y**2)

>>> pstr = latex_printer(m.expression_1)
```

A Simple Expression

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib.latex_printer import latex_printer

>>> m = pyo.ConcreteModel(name = 'basicFormulation')
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()

>>> pstr = latex_printer(m.x + m.y)
```

3.5.3 Nonlinear Preprocessing Transformations

`pyomo.contrib.preprocessing` is a contributed library of preprocessing transformations intended to operate upon nonlinear and mixed-integer nonlinear programs (NLPs and MINLPs), as well as generalized disjunctive programs (GDPs).

This contributed package is maintained by Qi Chen and his colleagues from Carnegie Mellon University.

The following preprocessing transformations are available. However, some may later be deprecated or combined,

depending on their usefulness.

var_aggregator.VariableAggregator	Aggregate model variables that are linked by equality constraints.
bounds_to_vars.ConstraintToVarBoundTransform	Change constraints to be a bound on the variable.
induced_linearity.InducedLinearity	Reformulate nonlinear constraints with induced linearity.
constraint_tightener.TightenConstraintFromVars	DEPRECATED.
deactivate_trivial_constraints.TrivialConstraintDeactivator	Deactivates trivial constraints.
detect_fixed_vars.FixedVarDetector	Detects variables that are de-facto fixed but not considered fixed.
equality_propagate.FixedVarPropagator	Propagate variable fixing for equalities of type $x = y$.
equality_propagate.VarBoundPropagator	Propagate variable bounds for equalities of type $x = y$.
init_vars.InitMidpoint	Initialize non-fixed variables to the midpoint of their bounds.
init_vars.InitZero	Initialize non-fixed variables to zero.
remove_zero_terms.RemoveZeroTerms	Looks for $0v$ in a constraint and removes it.
strip_bounds.VariableBoundStripper	Strip bounds from variables.
zero_sum_propagator.ZeroSumPropagator	Propagates fixed-to-zero for sums of only positive (or negative) vars.

Variable Aggregator

The following code snippet demonstrates usage of the variable aggregation transformation on a concrete Pyomo model:

```
>>> import pyomo.environ as pyo
>>> m = pyo.ConcreteModel()
>>> m.v1 = pyo.Var(initialize=1, bounds=(1, 8))
>>> m.v2 = pyo.Var(initialize=2, bounds=(0, 3))
>>> m.v3 = pyo.Var(initialize=3, bounds=(-7, 4))
>>> m.v4 = pyo.Var(initialize=4, bounds=(2, 6))
>>> m.c1 = pyo.Constraint(expr=m.v1 == m.v2)
>>> m.c2 = pyo.Constraint(expr=m.v2 == m.v3)
>>> m.c3 = pyo.Constraint(expr=m.v3 == m.v4)
>>> pyo.TransformationFactory('contrib.aggregate_vars').apply_to(m)
```

To see the results of the transformation, you could then use the command

```
>>> m.pprint()
```

class pyomo.contrib.preprocessing.plugins.var_aggregator.**VariableAggregator**(***kwds*)

Aggregate model variables that are linked by equality constraints.

Before:

$$\begin{aligned} x &= y \\ a &= 2x + 6y + 7 \\ b &= 5y + 6 \end{aligned}$$

After:

$$z = x = y$$
$$a = 8z + 7$$
$$b = 5z + 6$$

> **Warning**
>
> TODO: unclear what happens to "capital-E" Expressions at this point in time.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

update_variables(*model*)

Update the values of the variables that were replaced by aggregates.

TODO: reduced costs

Explicit Constraints to Variable Bounds

```
>>> import pyomo.environ as pyo
>>> m = pyo.ConcreteModel()
>>> m.v1 = pyo.Var(initialize=1)
>>> m.v2 = pyo.Var(initialize=2)
>>> m.v3 = pyo.Var(initialize=3)
>>> m.c1 = pyo.Constraint(expr=m.v1 == 2)
>>> m.c2 = pyo.Constraint(expr=m.v2 >= -2)
>>> m.c3 = pyo.Constraint(expr=m.v3 <= 5)
>>> pyo.TransformationFactory('contrib.constraints_to_var_bounds').apply_to(m)
```

class pyomo.contrib.preprocessing.plugins.bounds_to_vars.**ConstraintToVarBoundTransform**(***kwds*)

Change constraints to be a bound on the variable.

Looks for constraints of form: $k * v + c_1 \leq c_2$. Changes variable lower bound on v to match $(c_2 - c_1)/k$ if it results in a tighter bound. Also does the same thing for lower bounds.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

- **tolerance** (*NonNegativeFloat, default=1e-13*) – tolerance on bound equality $(LB = UB)$
- **detect_fixed** (*bool, default=True*) – If True, fix variable when $|LB - UB| \leq tolerance$.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Induced Linearity Reformulation

class pyomo.contrib.preprocessing.plugins.induced_linearity.**InducedLinearity**(***kwds*)

Reformulate nonlinear constraints with induced linearity.

Finds continuous variables v where $v = d_1 + d_2 + d_3$, where d's are discrete variables. These continuous variables may participate nonlinearly in other expressions, which may then be induced to be linear.

The overall algorithm flow can be summarized as:

1. Detect effectively discrete variables and the constraints that imply discreteness.
2. Determine the set of valid values for each effectively discrete variable
3. Find nonlinear expressions in which effectively discrete variables participate.
4. Reformulate nonlinear expressions appropriately.

> **Note**
>
> Tasks 1 & 2 must incorporate scoping considerations (Disjuncts)

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

- **equality_tolerance** (*NonNegativeFloat, default=1e-06*) – Tolerance on equality constraints.
- **pruning_solver** (*default='glpk'*) – Solver to use when pruning possible values.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Constraint Bounds Tightener

This transformation was developed by Sunjeev Kale at Carnegie Mellon University.

class pyomo.contrib.preprocessing.plugins.constraint_tightener.**TightenConstraintFromVars**

DEPRECATED.

Tightens upper and lower bound on constraints based on variable bounds.

Iterates through each variable and tightens the constraint bounds using the inferred values from the variable bounds.

For now, this only operates on linear constraints.

Deprecated since version 5.7: Use of the constraint tightener transformation is deprecated. Its functionality may be partially replicated using *pyomo.contrib.fbbt.compute_bounds_on_expr(constraint.body)*.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Trivial Constraint Deactivation

class pyomo.contrib.preprocessing.plugins.deactivate_trivial_constraints.**TrivialConstraintDeactivator**(**

Deactivates trivial constraints.

Trivial constraints take form $k_1 = k_2$ or $k_1 \leq k_2$, where k_1 and k_2 are constants. These constraints typically arise when variables are fixed.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

- **tmp** (*bool, default=False*) – True to store a set of transformed constraints for future reversion of the transformation.
- **ignore_infeasible** (*bool, default=False*) – True to skip over trivial constraints that are infeasible instead of raising an InfeasibleConstraintException.
- **return_trivial** (*default=[]*) – a list to which the deactivated trivialconstraints are appended (side effect)
- **tolerance** (*NonNegativeFloat, default=1e-13*) – tolerance on constraint violations

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

revert(*instance*)

Revert constraints deactivated by the transformation.

Parameters
instance – the model instance on which trivial constraints were earlier deactivated.

Fixed Variable Detection

class pyomo.contrib.preprocessing.plugins.detect_fixed_vars.**FixedVarDetector**(***kwds*)

Detects variables that are de-facto fixed but not considered fixed.

For each variable v found on the model, check to see if its lower bound v^{LB} is within some tolerance of its upper bound v^{UB}. If so, fix the variable to the value of v^{LB}.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

- **tmp** (*bool, default=False*) – True to store the set of transformed variables and their old values so that they can be restored.
- **tolerance** (*NonNegativeFloat, default=1e-13*) – tolerance on bound equality (LB == UB)

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

revert(*instance*)

Revert variables fixed by the transformation.

Fixed Variable Equality Propagator

class pyomo.contrib.preprocessing.plugins.equality_propagate.**FixedVarPropagator**(***kwds*)

Propagate variable fixing for equalities of type $x = y$.

If x is fixed and y is not fixed, then this transformation will fix y to the value of x.

This transformation can also be performed as a temporary transformation, whereby the transformed variables are saved and can be later unfixed.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

tmp (*bool, default=False*) – True to store the set of transformed variables and their old states so that they can be later restored.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

revert(*instance*)

Revert variables fixed by the transformation.

Variable Bound Equality Propagator

class pyomo.contrib.preprocessing.plugins.equality_propagate.**VarBoundPropagator**(***kwds*)

Propagate variable bounds for equalities of type $x = y$.

If x has a tighter bound then y, then this transformation will adjust the bounds on y to match those of x.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

tmp (*bool, default=False*) – True to store the set of transformed variables and their old states so that they can be later restored.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

revert(*instance*)

Revert variable bounds.

Variable Midpoint Initializer

class pyomo.contrib.preprocessing.plugins.init_vars.**InitMidpoint**(***kwds*)

Initialize non-fixed variables to the midpoint of their bounds.

- If the variable does not have bounds, set the value to zero.
- If the variable is missing one bound, set the value to that of the existing bound.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Variable Zero Initializer

class pyomo.contrib.preprocessing.plugins.init_vars.**InitZero**(***kwds*)

Initialize non-fixed variables to zero.

- If setting the variable value to zero will violate a bound, set the variable value to the relevant bound value.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Zero Term Remover

class pyomo.contrib.preprocessing.plugins.remove_zero_terms.**RemoveZeroTerms**(***kwds*)

Looks for $0v$ in a constraint and removes it.

Currently limited to processing linear constraints of the form $x_1 = 0x_3$, occurring as a result of fixing $x_2 = 0$.

Note

TODO: support nonlinear expressions

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

Variable Bound Remover

class pyomo.contrib.preprocessing.plugins.strip_bounds.**VariableBoundStripper**(***kwds*)

Strip bounds from variables.

Keyword arguments below are specified for the `apply_to` and `create_using` functions.

Keyword Arguments

- **strip_domains** (`bool`, *default=True*) – strip the domain for discrete variables as well
- **reversible** (`bool`, *default=False*) – Whether the bound stripping will be temporary. If so, store information for reversion.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

revert(*instance*)

Revert variable bounds and domains changed by the transformation.

Zero Sum Propagator

class pyomo.contrib.preprocessing.plugins.zero_sum_propagator.**ZeroSumPropagator**(***kwds*)

Propagates fixed-to-zero for sums of only positive (or negative) vars.

If z is fixed to zero and $z = x_1 + x_2 + x_3$ and x_1, x_2, x_3 are all non-negative or all non-positive, then x_1, x_2, and x_3 will be fixed to zero.

apply_to(*model*, ***kwds*)

Apply the transformation to the given model.

create_using(*model*, ***kwds*)

Create a new model with this transformation

3.5.4 Model Scaling Transformation

Good scaling of models can greatly improve the numerical properties of a problem and thus increase reliability and convergence. The `core.scale_model` transformation allows users to separate scaling of a model from the declaration of the model variables and constraints which allows for models to be written in more natural forms and to be scaled and rescaled as required without having to rewrite the model code.

pyomo.core.plugins.transform.scaling.ScaleModel(**kwds)	Transformation to scale a model.

Setting Scaling Factors

Scaling factors for components in a model are declared using *Suffixes*, as shown in the example above. In order to define a scaling factor for a component, a `Suffix` named `scaling_factor` must first be created to hold the scaling factor(s). Scaling factor suffixes can be declared at any level of the model hierarchy, but scaling factors declared on the higher-level `models` or `Blocks` take precedence over those declared at lower levels.

Scaling suffixes are dict-like where each key is a Pyomo component and the value is the scaling factor to be applied to that component.

In the case of indexed components, scaling factors can either be declared for an individual index or for the indexed component as a whole (with scaling factors for individual indices taking precedence over overall scaling factors).

Note

In the case that a scaling factor is declared for a component on at multiple levels of the hierarchy, the highest level scaling factor will be applied.

Note

It is also possible (but not encouraged) to define a "default" scaling factor to be applied to any component for which a specific scaling factor has not been declared by setting a entry in a Suffix with a key of `None`. In this case, the default value declared closest to the component to be scaled will be used (i.e., the first default value found when walking up the model hierarchy).

Applying Model Scaling

The `core.scale_model` transformation provides two approaches for creating a scaled model.

In-Place Scaling

The `apply_to(model)` method can be used to apply scaling directly to an existing model. When using this method, all the variables, constraints and objectives within the target model are replaced with new scaled components and the appropriate scaling factors applied. The model can then be sent to a solver as usual, however the results will be in terms of the scaled components and must be un-scaled by the user.

Creating a New Scaled Model

Alternatively, the `create_using(model)` method can be used to create a new, scaled version of the model which can be solved. In this case, a clone of the original model is generated with the variables, constraints and objectives replaced by scaled equivalents. Users can then send the scaled model to a solver after which the `propagate_solution` method can be used to map the scaled solution back onto the original model for further analysis.

The advantage of this approach is that the original model is maintained separately from the scaled model, which facilitates rescaling and other manipulation of the original model after a solution has been found. The disadvantage of this approach is that cloning the model may result in memory issues when dealing with larger models.

3.5.5 aslfunctions

Pyomo provides a set of AMPL user-defined functions that commonly occur but cannot be easily written as Pyomo expressions.

Using These AMPL External Functions

Build

You must build the Pyomo extensions to use these functions. Run `pyomo build-extensions` in the terminal and make sure the `aslfunctions` build status is "ok."

Example

```
>>> import pyomo.environ as pyo
>>> from pyomo.common.fileutils import find_library
>>> flib = find_library("aslfunctions")
>>> m = pyo.ConcreteModel(name = 'AMPLExternalFunctions')
>>> m.sinc = pyo.ExternalFunction(library=flib, function="sinc")
>>> m.x = pyo.Var()
>>> m.z = pyo.Var()
>>> m.constraint = pyo.Constraint(expr = m.z == m.sinc(m.x))
```

Functions

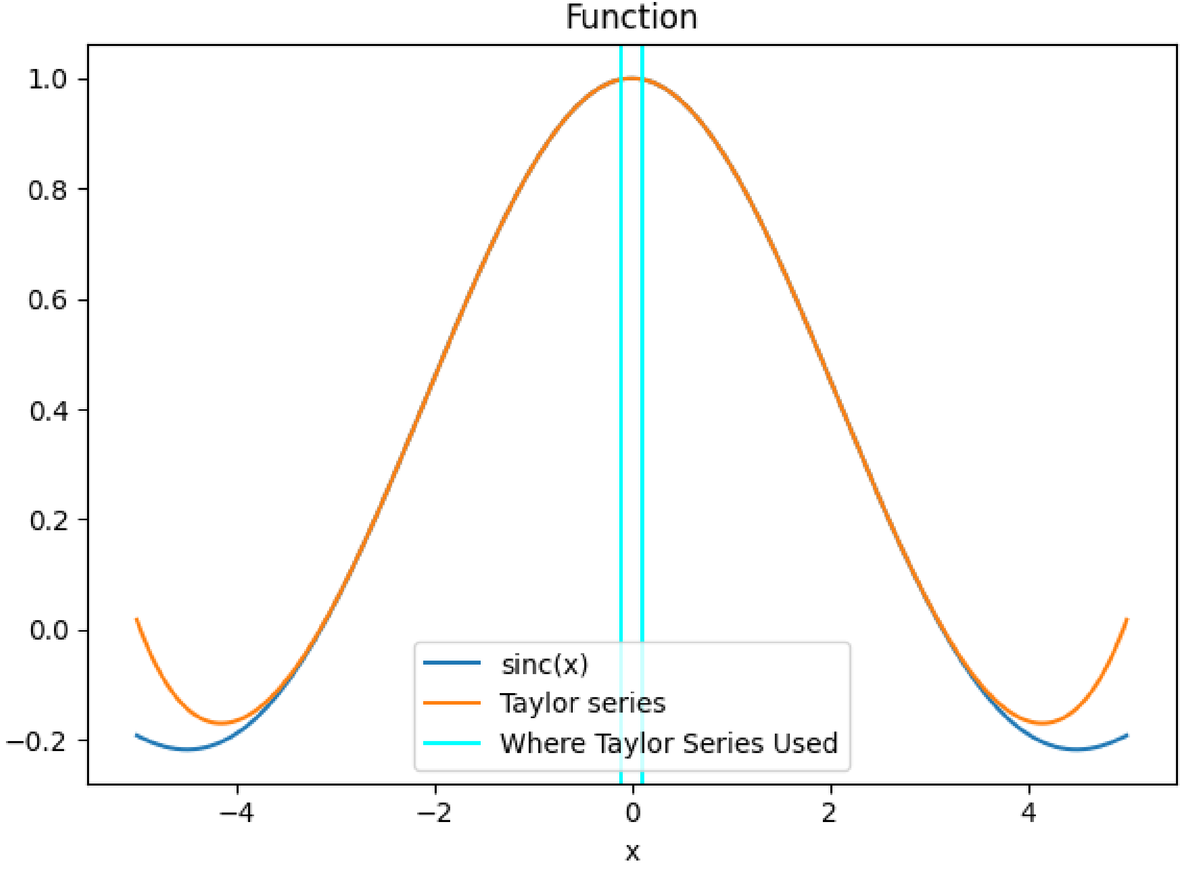
sinc(x)

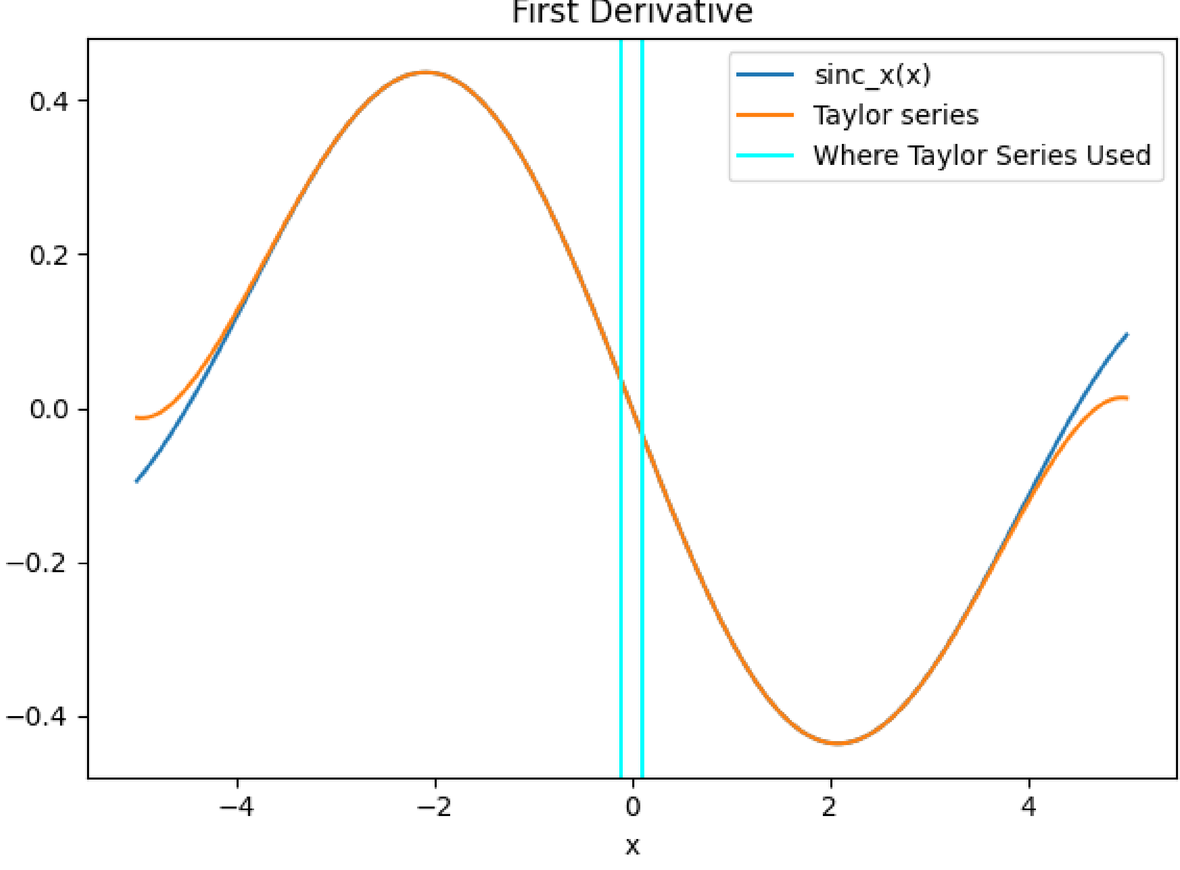
This function is defined as:

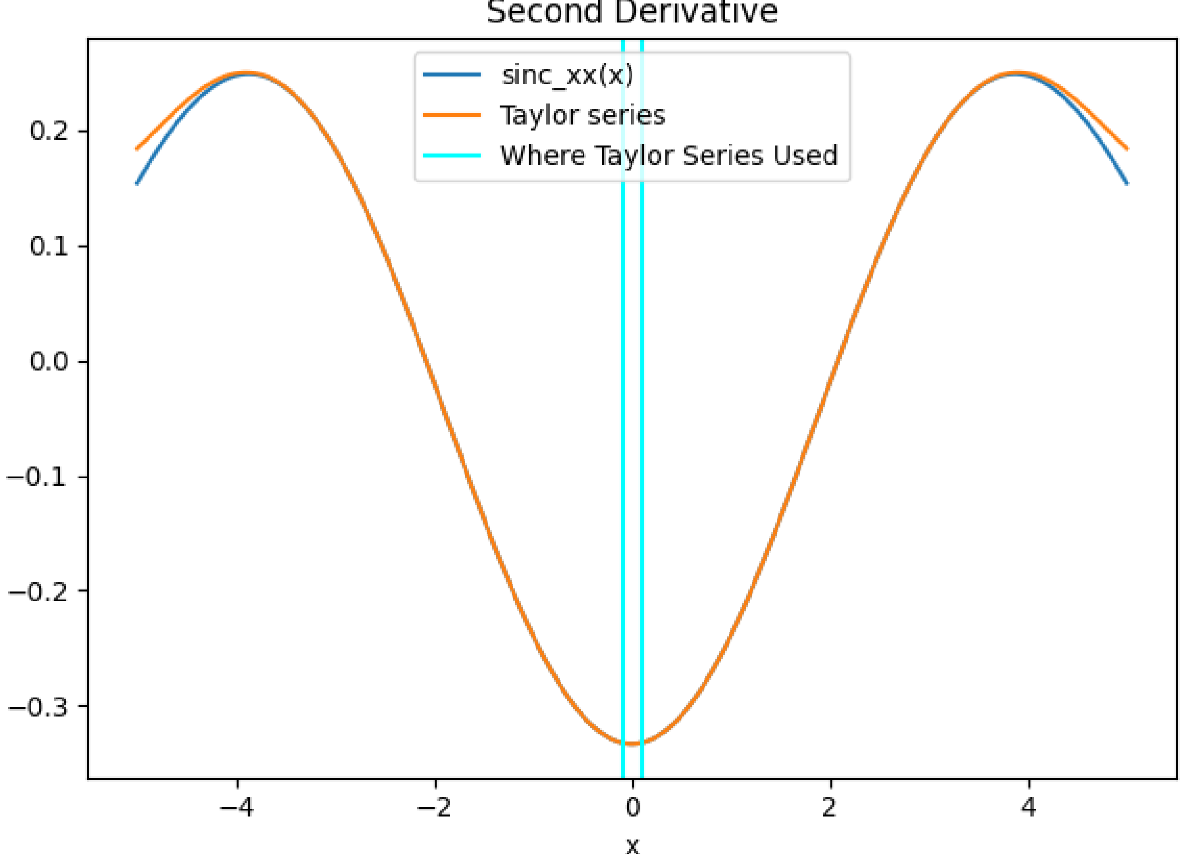
$$\operatorname{sinc}(x) = \begin{cases} \sin(x)/x & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

In this implementation, the region $-0.1 < x < 0.1$ is replaced by a Taylor series with enough terms that the function should be at least C^2 smooth. The difference between the function and the Tayor series is near the limits of machine precision, about 1×10^{-16} for the function value, 1×10^{-16} for the first derivative, and 1×10^{-14} for the second derivative.

These figures show the sinc(x) function, the Taylor series and where the Taylor series is used.


Function
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−4
−2
0
2
4
x
sinc(x)
Taylor series
Where Taylor Series Used


First Derivative
sinc_x(x)
Taylor series
Where Taylor Series Used
0.4
0.2
0.0
−0.2
−0.4
−4
−2
0
2
4
x



sgnsqr(x)

This function is defined as:

$$\text{sgnsqr}(x) = \text{sgn}(x)x^2$$

This function is only C^1 smooth because at 0 the second derivative is undefined and the jumps from -2 to 2.

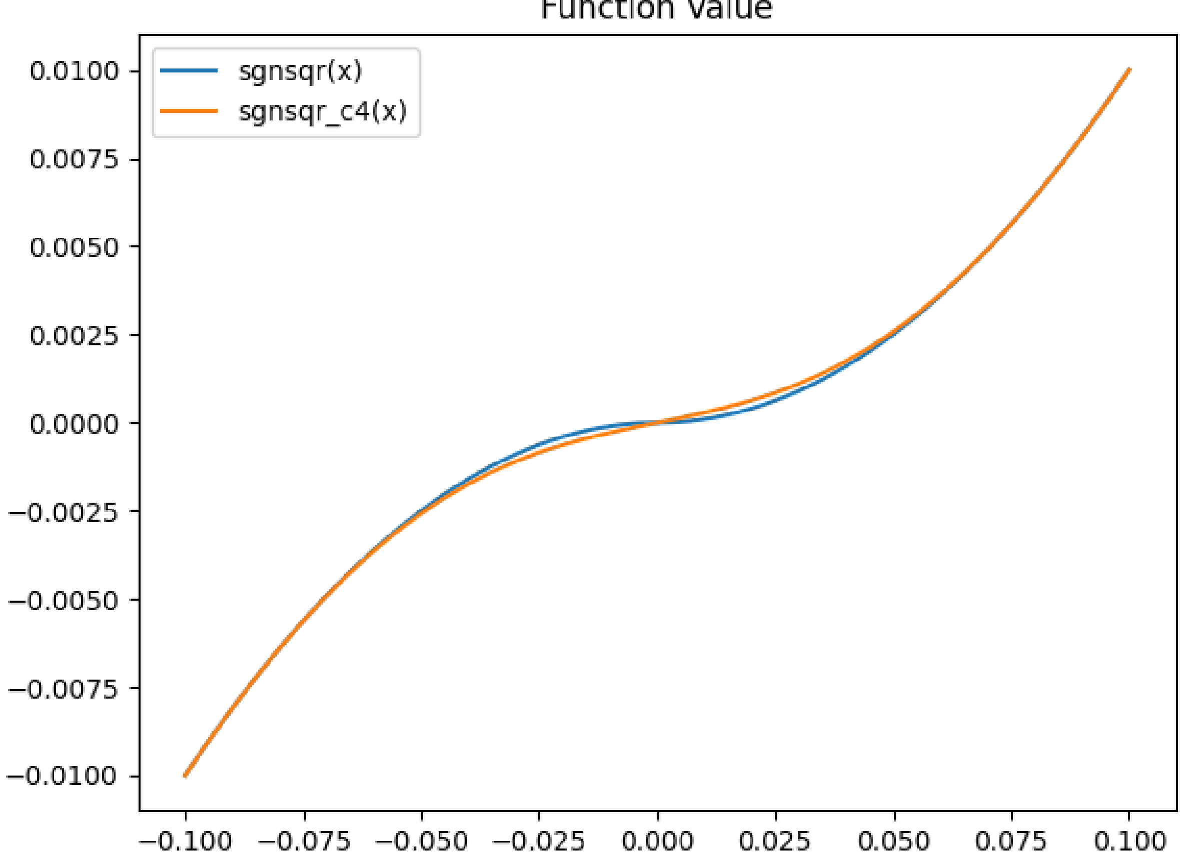
sgnsqr_c4(x)

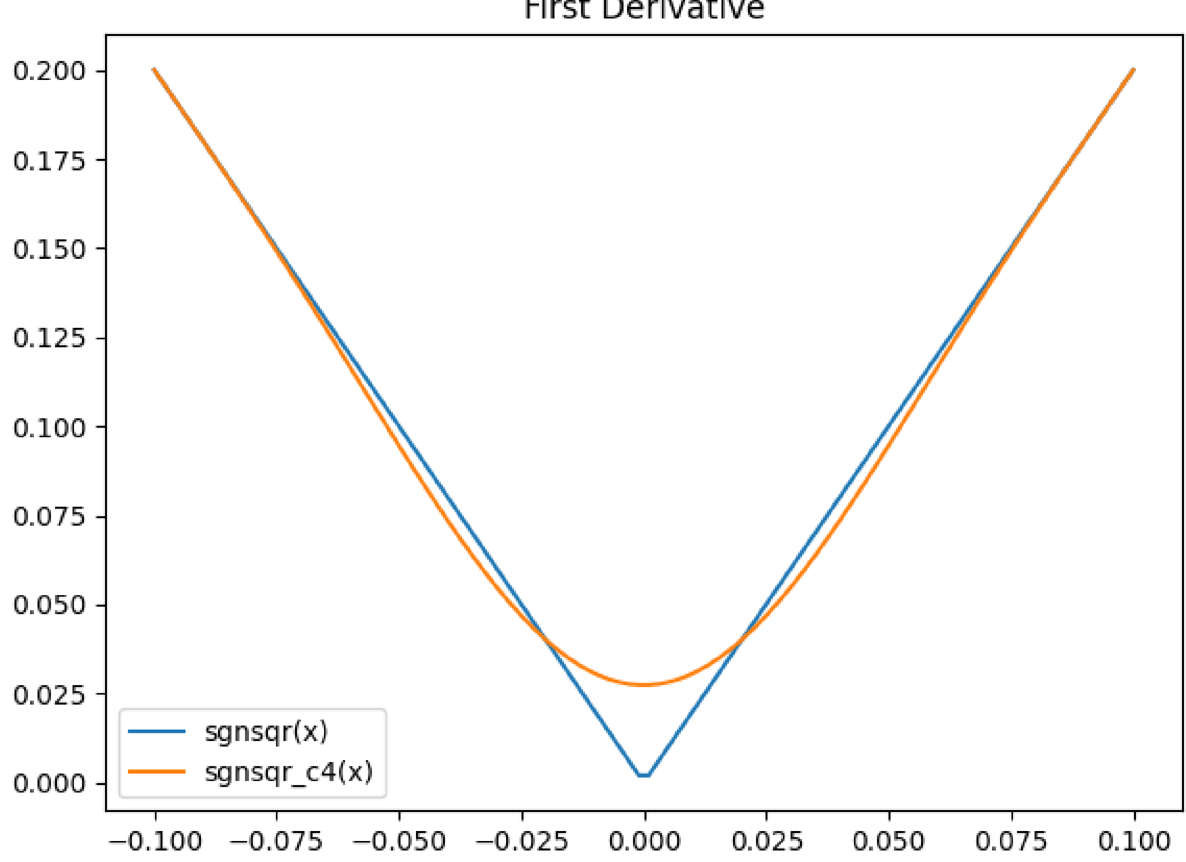
This function is defined as:

$$\text{sgnsqr_c4}(x) = \begin{cases} \text{sgn}(x)\,x^2 & \text{if } |x| \geq 0.1, \\ \sum_{i=0}^{11} c_i x^i & \text{if } |x| < 0.1 \end{cases}$$

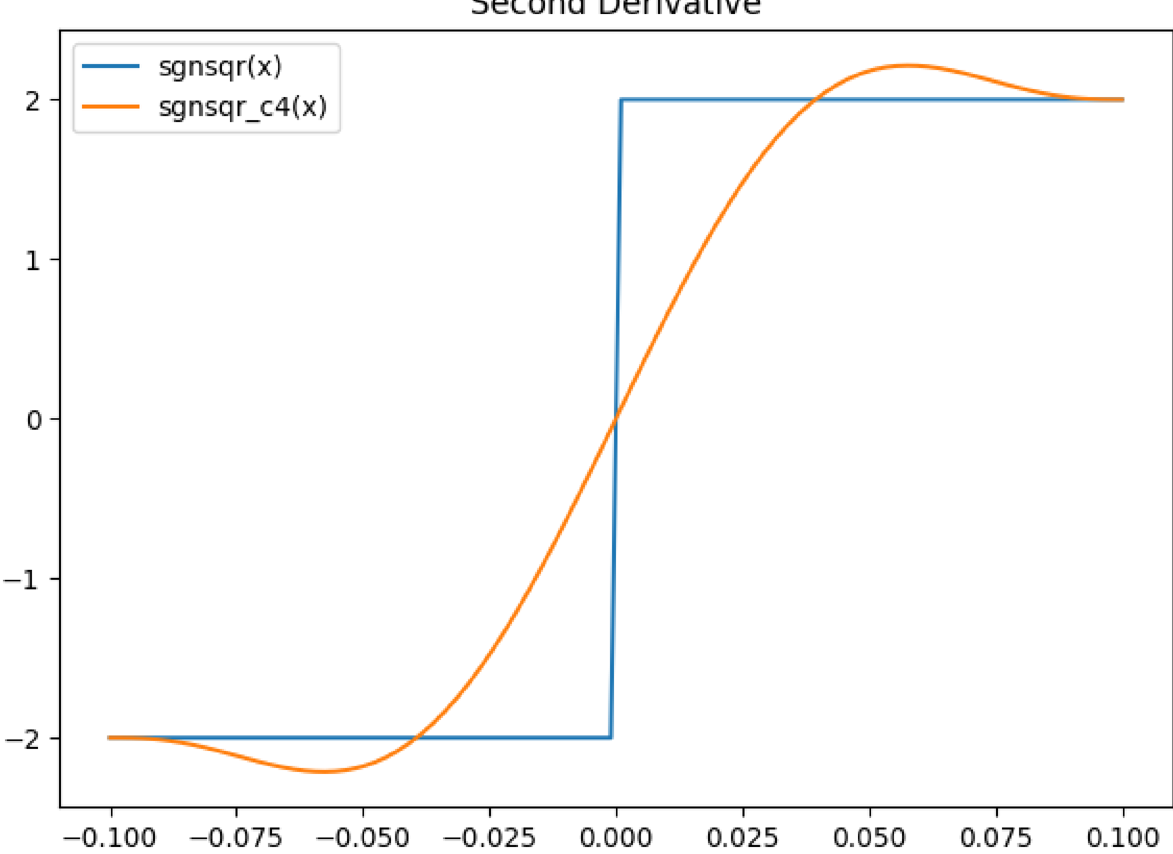
This function is C^4 smooth. The region $-0.1 < x < 0.1$ is replaced by an 11th order polynomial that approximates $\text{sgn}(x)x^2$. This function has well behaved derivatives at $x = 0$. If you need to use this function with very small numbers and high accuracy is important, you can scale the argument up (e.g. $\text{sgnsqr_c4}(sx)/s^2$).

These figures show the sgnsqr(x) function compared to the smooth approximation sgnsqr_c4(x).


Function Value
sgnsqr(x)
sgnsqr_c4(x)
0.0100
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100



First Derivative
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
sgnsqr(x)
sgnsqr_c4(x)
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100



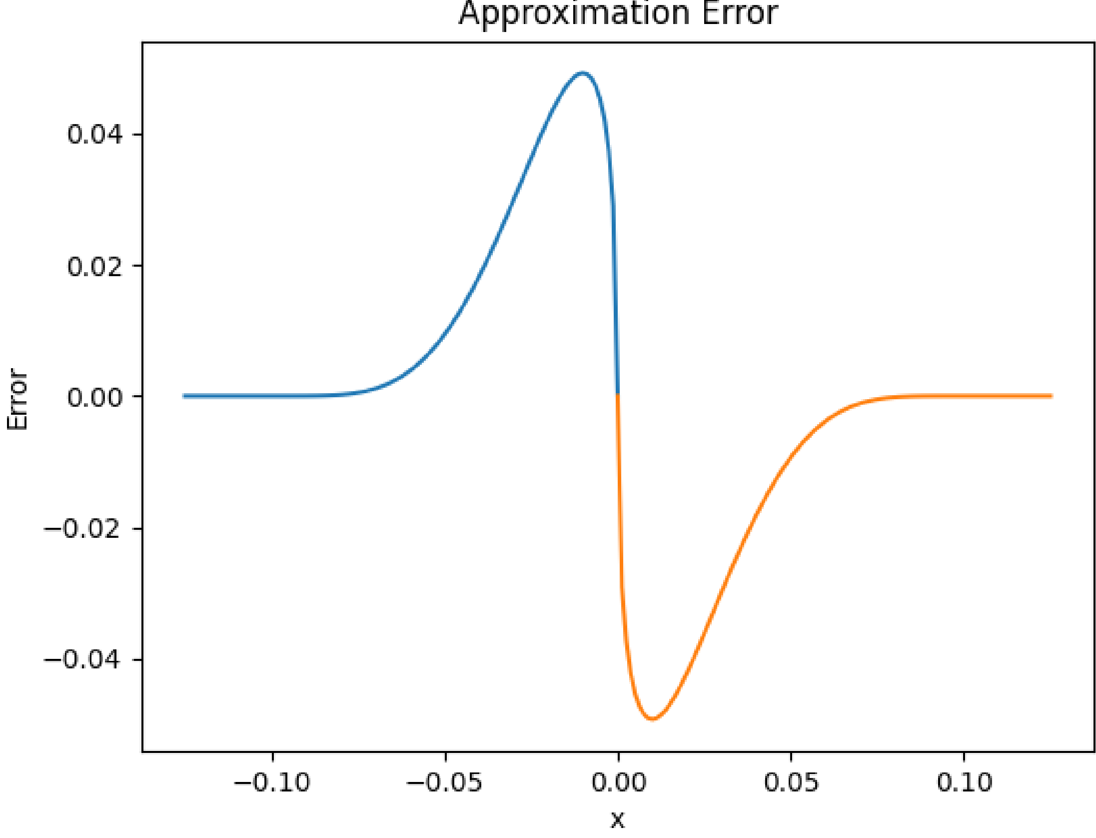
sgnsqrt_c4(x)

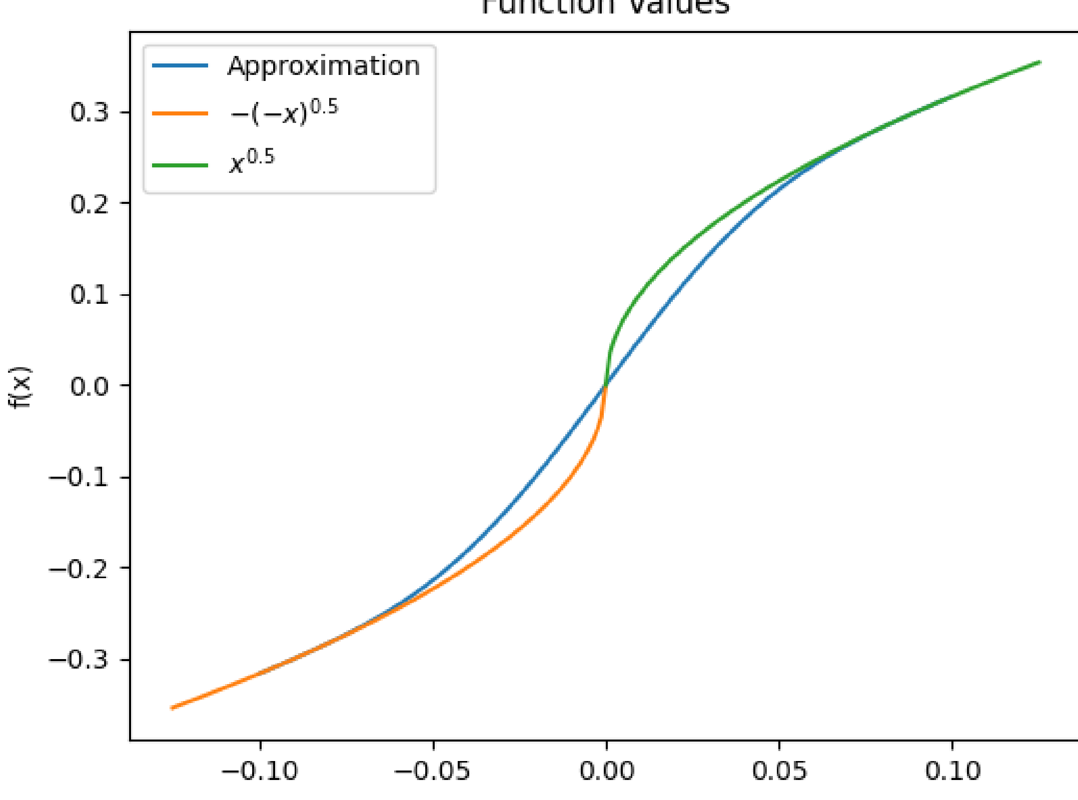
This function is a signed square root approximation defined as:

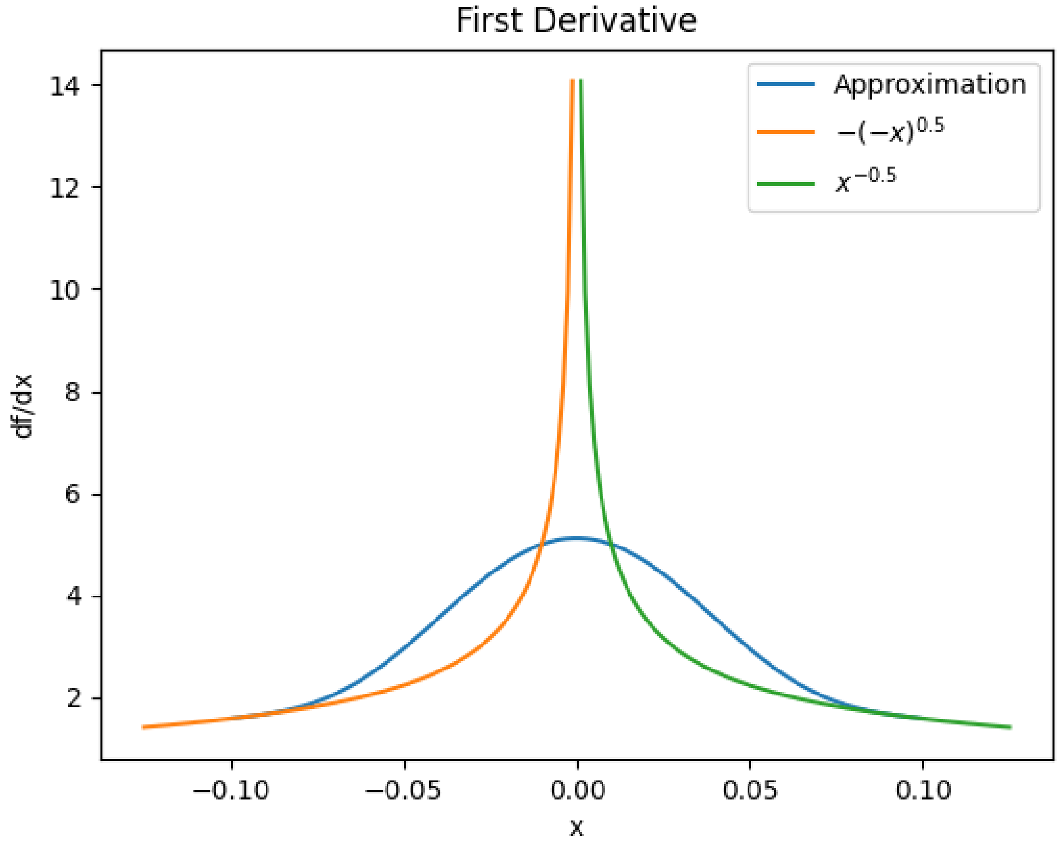
$$\text{sgnsqrt_c4}(x) = \begin{cases} \text{sgn}(x)\,|x|^{0.5} & \text{if } |x| \geq 0.1, \\ \sum_{i=0}^{11} c_i x^i & \text{if } |x| < 0.1 \end{cases}$$

This function is C^4 smooth. The region $-0.1 < x < 0.1$ is replaced by an 11th order polynomial that approximates $\text{sgn}(x)|x|^{0.5}$. This function has well behaved derivatives at $x = 0$. If you need to use this function with very small numbers and high accuracy is important, you can scale the argument up (e.g. $\text{sgnsqrt_c4}(sx)/s^{0.5}$).

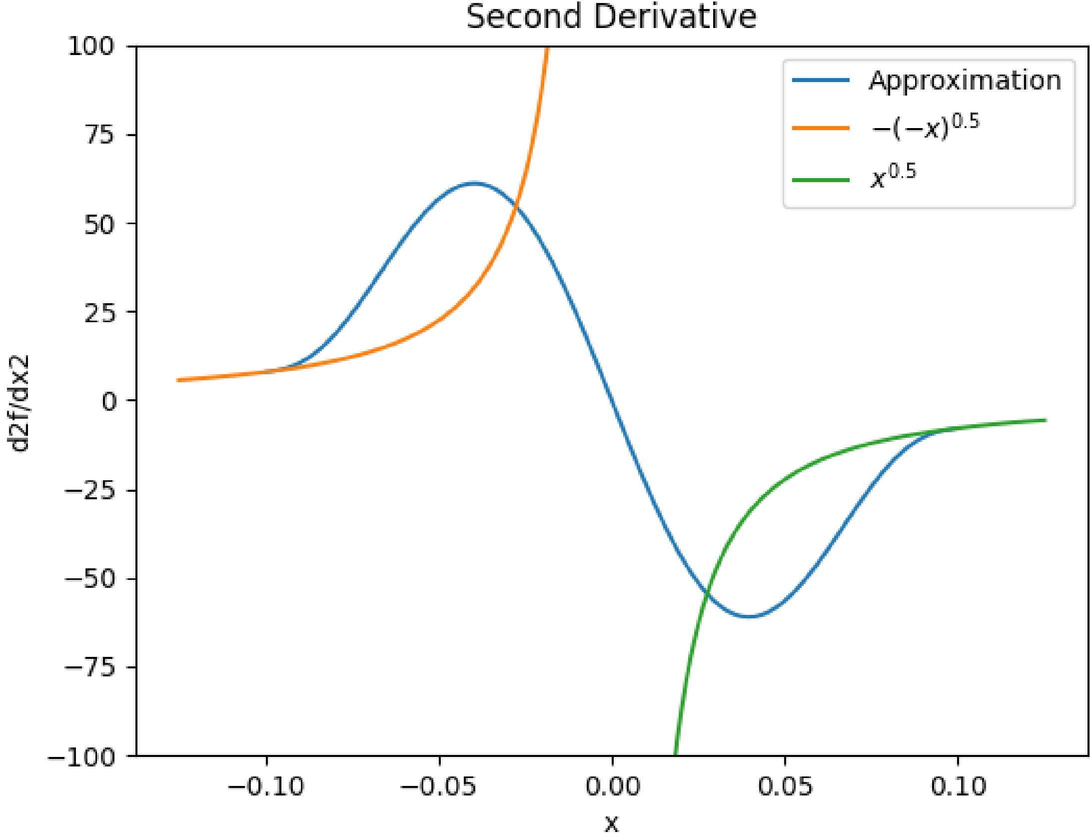
These figures show the signed square root function compared to the smooth approximation sgnsqrt_c4(x).


Approximation Error
Error
x
0.04
0.02
0.00
−0.02
−0.04
−0.10
−0.05
0.00
0.05
0.10


Function Values
Approximation
$-(-x)^{0.5}$
$x^{0.5}$
f(x)
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.10
−0.05
0.00
0.05
0.10
x



First Derivative
Approximation
$-(-x)^{0.5}$
$x^{-0.5}$
df/dx
x
14
12
10
8
6
4
2
−0.10
−0.05
0.00
0.05
0.10



3.6 Developer Utilities

3.6.1 The Pyomo Configuration System

The Pyomo configuration system provides a set of three classes (*ConfigDict*, *ConfigList*, and *ConfigValue*) for managing and documenting structured configuration information and user input. The system is based around the *ConfigValue* class, which provides storage for a single configuration entry. *ConfigValue* objects can be grouped using two containers (*ConfigDict* and *ConfigList*) that provide functionality analogous to Python's `dict` and `list` classes, respectively.

At its simplest, the configuration system allows for developers to specify a dictionary of documented configuration entries:

```
from pyomo.common.config import (
    ConfigDict, ConfigList, ConfigValue
)
config = ConfigDict()
config.declare('filename', ConfigValue(
    default=None,
    domain=str,
    description="Input file name",
))
config.declare("bound tolerance", ConfigValue(
```

(continues on next page)

(continued from previous page)

```
    default=1E-5,
    domain=float,
    description="Bound tolerance",
    doc="Relative tolerance for bound feasibility checks"
))
config.declare("iteration limit", ConfigValue(
    default=30,
    domain=int,
    description="Iteration limit",
    doc="Number of maximum iterations in the decomposition methods"
))
```

Users can then provide values for those entries, and retrieve the current values:

```
>>> config['filename'] = 'tmp.txt'
>>> print(config['filename'])
tmp.txt
>>> print(config['iteration limit'])
30
```

For convenience, *ConfigDict* objects support read/write access via attributes (with spaces in the declaration names replaced by underscores):

```
>>> print(config.filename)
tmp.txt
>>> print(config.iteration_limit)
30
>>> config.iteration_limit = 20
>>> print(config.iteration_limit)
20
```

Domain validation

All Config objects support a `domain` keyword that accepts a callable object (type, function, or callable instance). The domain callable should take a single argument (the incoming data value) and map it onto the desired domain, optionally performing domain validation (see *ConfigValue*, *ConfigDict*, and *ConfigList* for more information). This allows client code to accept a very flexible set of inputs without “cluttering” the code with input validation:

```
>>> config.iteration_limit = 35.5
>>> print(config.iteration_limit)
35
>>> print(type(config.iteration_limit).__name__)
int
```

In addition to common types (like `int`, `float`, `bool`, and `str`), the configuration system provides a number of custom domain validators for common use cases:

Bool(val)	Domain validator for bool-like objects.
Integer(val)	Domain validation function admitting integers
PositiveInt(val)	Domain validation function admitting strictly positive integers

continues on next page

Table 3.17 – continued from previous page

NegativeInt(val)	Domain validation function admitting strictly negative integers
NonNegativeInt(val)	Domain validation function admitting integers >= 0
NonPositiveInt(val)	Domain validation function admitting integers <= 0
PositiveFloat(val)	Domain validation function admitting strictly positive numbers
NegativeFloat(val)	Domain validation function admitting strictly negative numbers
NonPositiveFloat(val)	Domain validation function admitting numbers less than or equal to 0
NonNegativeFloat(val)	Domain validation function admitting numbers greater than or equal to 0
In(domain[, cast])	Domain validation class admitting a Container of possible values
InEnum(domain)	Domain validation class admitting an enum value/name.
IsInstance(*bases[, document_full_base_names])	Domain validator for type checking.
ListOf([itemtype, domain, string_lexer])	Domain validator for lists of a specified type
Module([basePath, expandPath])	Domain validator for modules.
Path([basePath, expandPath])	Domain validator for a path-like object.
PathList([basePath, expandPath])	Domain validator for a list of path-like objects.
DynamicImplicitDomain(callback)	Implicit domain that can return a custom domain based on the key.

Configuring class hierarchies

A feature of the configuration system is that the core classes all implement `__call__`, and can themselves be used as `domain` values. Beyond providing domain verification for complex hierarchical structures, this feature allows *ConfigDict* objects to cleanly support extension and the configuration of derived classes. Consider the following example:

```
>>> class Base:
...     CONFIG = ConfigDict()
...     CONFIG.declare('filename', ConfigValue(
...         default='input.txt',
...         domain=str,
...     ))
...     def __init__(self, **kwds):
...         self.cfg = self.CONFIG(kwds)
...         self.cfg.display()
...
>>> class Derived(Base):
...     CONFIG = Base.CONFIG()
...     CONFIG.declare('pattern', ConfigValue(
...         default=None,
...         domain=str,
...     ))
...
>>> tmp = Base(filename='foo.txt')
filename: foo.txt
>>> tmp = Derived(pattern='.*warning')
filename: input.txt
pattern: .*warning
```

Here, the base class Base declares a class-level attribute CONFIG as a *ConfigDict* containing a single entry (filename). The derived class (Derived) then starts by making a copy of the base class' CONFIG, and then defines an additional entry (pattern). Instances of the base class will still create cfg attributes that only have the single filename entry, whereas instances of the derived class will have cfg attributes with two entries: the pattern entry declared by the derived class, and the filename entry "inherited" from the base class.

An extension of this design pattern provides a clean approach for handling "ephemeral" instance options. Consider an interface to an external "solver". Our class implements a solve() method that takes a problem and sends it to the solver along with some solver configuration options. We would like to be able to set those options "persistently" on instances of the interface class, but still override them "temporarily" for individual calls to solve(). We implement this by creating copies of the class's configuration for both specific instances and for use by each solve() call:

```
class Solver:
    CONFIG = ConfigDict()
    CONFIG.declare('iterlim', ConfigValue(
        default=10,
        domain=int,
    ))

    def __init__(self, **kwds):
        self.config = self.CONFIG(kwds)

    def solve(self, model, **options):
        config = self.config(options)
        # Solve the model with the specified iterlim
        config.display()
```

```
>>> solver = Solver()
>>> solver.solve(None)
iterlim: 10
>>> solver.config.iterlim = 20
>>> solver.solve(None)
iterlim: 20
>>> solver.solve(None, iterlim=50)
iterlim: 50
>>> solver.solve(None)
iterlim: 20
```

This design pattern is widely used across Pyomo; particularly for configuring solver interfaces and transformations. We provide a decorator to simplify the process of documenting these CONFIG attributes:

```
from pyomo.common.config import document_class_CONFIG

@document_class_CONFIG(methods=['solve'])
class MySolver:
    """Interface to My Solver"""
    #
    #: Global class configuration; see :ref:`MySolver_CONFIG`
    CONFIG = ConfigDict()
    CONFIG.declare('iterlim', ConfigValue(
        default=10,
        domain=int,
        doc="Solver iteration limit",
    ))
```

(continues on next page)

(continued from previous page)

```
    #
    def __init__(self, **kwds):
        #: Instance configuration; see :ref:`MySolver_CONFIG`
        self.config = self.CONFIG(kwds)
    #
    def solve(self, model, **options):
        """Solve `model` using My Solver"""
        #
        config = self.config(options)
        # Solve the model with the specified iterlim
        config.display()
```

```
>>> print(MySolver.__doc__)
Interface to My Solver

**Class configuration**

This class leverages the Pyomo Configuration System for managing
configuration options.  See the discussion on :ref:`configuring class
hierarchies <class_config>` for more information on how configuration
class attributes, instance attributes, and method keyword arguments
interact.

.. _MySolver::CONFIG:

CONFIG
------
iterlim: int, default=10

    Solver iteration limit

>>> print(MySolver.solve.__doc__)
Solve `model` using My Solver

Keyword Arguments
-----------------
iterlim: int, default=10

    Solver iteration limit
```

Interacting with argparse

In addition to basic storage and retrieval, the configuration system provides hooks to the argparse command-line argument parsing system. Individual configuration entries can be declared as `argparse` arguments using the *declare_as_argument()* method. To make declaration simpler, the *declare()* method returns the declared configuration object so that the argument declaration can be done inline:

```
import argparse
config = ConfigDict()
config.declare('iterlim', ConfigValue(
    domain=int,
    default=100,
```

(continues on next page)

(continued from previous page)

```
    description="iteration limit",
)).declare_as_argument()
config.declare('lbfgs', ConfigValue(
    domain=bool,
    description="use limited memory BFGS update",
)).declare_as_argument()
config.declare('linesearch', ConfigValue(
    domain=bool,
    default=True,
    description="use line search",
)).declare_as_argument()
config.declare('relative tolerance', ConfigValue(
    domain=float,
    description="relative convergence tolerance",
)).declare_as_argument('--reltol', '-r', group='Tolerances')
config.declare('absolute tolerance', ConfigValue(
    domain=float,
    description="absolute convergence tolerance",
)).declare_as_argument('--abstol', '-a', group='Tolerances')
```

The *ConfigDict* can then be used to initialize (or augment) an `argparse.ArgumentParser` object:

```
parser = argparse.ArgumentParser("tester")
config.initialize_argparse(parser)
```

Key information from the *ConfigDict* is automatically transferred over to the `ArgumentParser` object:

```
>>> print(parser.format_help())
usage: tester [-h] [--iterlim INT] [--lbfgs] [--disable-linesearch]
              [--reltol FLOAT] [--abstol FLOAT]
...
  -h, --help            show this help message and exit
  --iterlim INT         iteration limit
  --lbfgs               use limited memory BFGS update
  --disable-linesearch  [DON'T] use line search

Tolerances:
  --reltol... -r FLOAT  relative convergence tolerance
  --abstol... -a FLOAT  absolute convergence tolerance
```

Parsed arguments can then be imported back into the *ConfigDict*:

```
>>> args=parser.parse_args(['--lbfgs', '--reltol', '0.1', '-a', '0.2'])
>>> args = config.import_argparse(args)
>>> config.display()
iterlim: 100
lbfgs: true
linesearch: true
relative tolerance: 0.1
absolute tolerance: 0.2
```

Accessing user-specified values

It is frequently useful to know which values a user explicitly set, and which values a user explicitly set but have never been retrieved. The configuration system provides two generator methods to return the items that a user explicitly set (`user_values()`) and the items that were set but never retrieved (`unused_user_values()`):

```
>>> print([val.name() for val in config.user_values()])
['lbfgs', 'relative tolerance', 'absolute tolerance']
>>> print(config.relative_tolerance)
0.1
>>> print([val.name() for val in config.unused_user_values()])
['lbfgs', 'absolute tolerance']
```

Outputting the current state

Configuration objects support two methods for generating output: *display()* and *generate_yaml_template()*. The simpler is *display()*, which prints out the current values of the configuration object (and if it is a container type, all of its children). *generate_yaml_template()* is similar to *display()*, but also includes the description fields as formatted comments.

```
solver_config = config
config = ConfigDict()
config.declare('output', ConfigValue(
    default='results.yml',
    domain=str,
    description='output results filename'
))
config.declare('verbose', ConfigValue(
    default=0,
    domain=int,
    description='output verbosity',
    doc='This sets the system verbosity.  The default (0) only logs '
    'warnings and errors.  Larger integer values will produce '
    'additional log messages.',
))
config.declare('solvers', ConfigList(
    domain=solver_config,
    description='list of solvers to apply',
))
```

```
>>> config.display()
output: results.yml
verbose: 0
solvers: []
>>> print(config.generate_yaml_template())
output: results.yml  # output results filename
verbose: 0           # output verbosity
solvers: []          # list of solvers to apply
```

It is important to note that both methods document the current state of the configuration object. So, in the example above, since the `solvers` list is empty, you will not get any information on the elements in the list. Of course, if you add a value to the list, then the data will be output:

```
>>> tmp = config()
>>> tmp.solvers.append({})
>>> tmp.display()
output: results.yml
verbose: 0
solvers:
  -
    iterlim: 100
    lbfgs: true
    linesearch: true
    relative tolerance: 0.1
    absolute tolerance: 0.2
>>> print(tmp.generate_yaml_template())
output: results.yml          # output results filename
verbose: 0                   # output verbosity
solvers:                     # list of solvers to apply
  -
    iterlim: 100             # iteration limit
    lbfgs: true              # use limited memory BFGS update
    linesearch: true         # use line search
    relative tolerance: 0.1  # relative convergence tolerance
    absolute tolerance: 0.2  # absolute convergence tolerance
```

Generating documentation

One of the most useful features of the Configuration system is the ability to automatically generate documentation. To accomplish this, we rely on a series of formatters derived from *ConfigFormatter* that implement a visitor pattern for walking the hierarchy of configuration containers (*ConfigDict* and *ConfigList*) and documenting the members. As the *ConfigFormatter* was designed to generate reference documentation, it behaves differently from *display()* or *generate_yaml_template()*):

- For each configuration item, the doc field is output. If the item has no doc, then the description field is used.
- List containers have their *domain* documented and not their current values.

The simplest interface for generating documentation is to call the *generate_documentation()* method. This method retrieves the specified formatter, instantiates it, and returns the result from walking the configuration object. The documentation format can be configured through optional arguments. The defaults generate LaTeX documentation:

```
>>> print(config.generate_documentation())
\begin{description}[topsep=0pt,parsep=0.5em,itemsep=-0.4em]
  \item[{output}]\hfill
    \\output results filename
  \item[{verbose}]\hfill
    \\This sets the system verbosity.  The default (0) only logs warnings and
    errors.  Larger integer values will produce additional log messages.
  \item[{solvers}]\hfill
    \\list of solvers to apply
  \begin{description}[topsep=0pt,parsep=0.5em,itemsep=-0.4em]
    \item[{iterlim}]\hfill
      \\iteration limit
    \item[{lbfgs}]\hfill
      \\use limited memory BFGS update
    \item[{linesearch}]\hfill
```

(continues on next page)

(continued from previous page)

```
      \\use line search
    \item[{relative tolerance}]\hfill
      \\relative convergence tolerance
    \item[{absolute tolerance}]\hfill
      \\absolute convergence tolerance
  \end{description}
\end{description}
```

More useful is actually documenting the source code itself. To this end, the Configuration system provides three decorators that append documentation of the referenced *ConfigDict* (in numpydoc format) for the most common situations:

document_configdict([section, ...])	Class decorator for documenting classes derived from *ConfigDict*.
document_class_CONFIG([section, ...])	Class decorator for documenting CONFIG class attributes.
document_kwargs_from_configdict(config[, ...])	Decorator to append the documentation of a ConfigDict to a class, method, or function docstring.

3.6.2 Deprecation and Removal of Functionality

During the course of development, there may be cases where it becomes necessary to deprecate or remove functionality from the standard Pyomo offering.

Deprecation

We offer a set of tools to help with deprecation in *pyomo.common.deprecation*.

By policy, when deprecating or moving an existing capability, one of the following utilities should be leveraged. Each has a required `version` argument that should be set to current development version (e.g., `"6.6.2.dev0"`). This version will be updated to the next actual release as part of the Pyomo release process. The current development version can be found by running

```
pyomo --version
```

on your local fork/branch.

deprecated([msg, logger, version, remove_in])	Decorator to indicate that a function, method, or class is deprecated.
deprecation_warning(msg[, logger, version, ...])	Standardized function for formatting and emitting deprecation warnings.
moved_module(old_name, new_name[, msg, ...])	Provide a deprecation path for moved / renamed modules
relocated_module_attribute(local, target, ...)	Provide a deprecation path for moved / renamed module attributes
RenamedClass(name, bases, classdict, *args, ...)	Metaclass to provide a deprecation path for renamed classes

Removal

By policy, functionality should be deprecated with reasonable warning, pending extenuating circumstances. The functionality should be deprecated, following the information above.

If the functionality is documented in the most recent edition of *Pyomo - Optimization Modeling in Python*, it may not be removed until the next major version release.

For other functionality, it is preferred that ample time is given before removing the functionality. At minimum, significant functionality removal will result in a minor version bump.

3.7 Experimental features

Warning

The `pyomo.kernel` API is still in the beta phase of development. It is fully tested and functional; however, the interface may change as it becomes further integrated with the rest of Pyomo.

Warning

Models built with `pyomo.kernel` components are not yet compatible with pyomo extension modules (e.g., `PySP`, `pyomo.dae`, `pyomo.gdp`).

3.7.1 The Kernel Library

The `pyomo.kernel` library is an experimental modeling interface designed to provide a better experience for users doing concrete modeling and advanced application development with Pyomo. It includes the basic set of *modeling components* necessary to build algebraic models, which have been redesigned from the ground up to make it easier for users to customize and extend. For a side-by-side comparison of `pyomo.kernel` and `pyomo.environ` syntax, visit the link below.

Syntax Comparison Table (pyomo.kernel vs pyomo.environ)

	pyomo.kernel	pyomo.environ
Import	`import pyomo.kernel as pmo`	`import pyomo.environ as aml`

Model[1]	`def create(data):`
	`    instance = pmo.block()`
	`    # ... define instance ...`
	`    return instance`
	
	`instance = create(data)`
		`m = aml.AbstractModel()`
		`# ... define model ...`
		`instance = m.create_instance(datafile)`

	`m = pmo.block()` `m.b = pmo.block()`	`m = aml.ConcreteModel()` `m.b = aml.Block()`
Set[2]	`m.s = [1, 2]`	`m.s = aml.Set(initialize=[1, 2], ordered=True)`
	`# [0,1,2]` `m.q = range(3)`	`# [1,2,3]` `m.q = aml.RangeSet(1, 3)`
Parameter[3]	`m.p = pmo.parameter(0)`	`m.p = aml.Param(mutable=True, initialize=0)`

	`# pd[1] = 0, pd[2] = 1`
	`m.pd = pmo.parameter_dict()`
	`for k, i in enumerate(m.s):`
	`    m.pd[i] = pmo.parameter(k)`
		`# pd[1] = 0, pd[2] = 1`
		`def pd_(m, i):`
		`    return m.s.ord(i) - 1`
		
		`m.pd = aml.Param(m.s, mutable=True, rule=pd_)`

	`# uses 0-based indexing`
	`# pl[0] = 0, pl[0] = 1, ...`
	`m.pl = pmo.parameter_list()`
	`for j in m.q:`
	`    m.pl.append(pmo.parameter(j))`
		`#`
		`# No ParamList exists`
		`#`

Variable	`m.v = pmo.variable(value=1, lb=1, ub=4)`	`m.v = aml.Var(initialize=1.0, bounds=(1, 4))`

	`m.vd = pmo.variable_dict()`
	`for i in m.s:`
	`    m.vd[i] = pmo.variable(ub=9)`
		`m.vd = aml.Var(m.s, bounds=(None, 9))`

	`# used 0-based indexing` `m.vl = pmo.variable_list()`	`# used 1-based indexing`

Models built from pyomo.kernel components are fully compatible with the standard solver interfaces included with Pyomo. A minimal example script that defines and solves a model is shown below.

```
# ___________________________________________________________________________________
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia, LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________________

import pyomo.kernel as pmo

model = pmo.block()
model.x = pmo.variable()
model.c = pmo.constraint(model.x >= 1)
model.o = pmo.objective(model.x)

opt = pmo.SolverFactory("ipopt")

result = opt.solve(model)
assert str(result.solver.termination_condition) == "optimal"
```

Notable Improvements

More Control of Model Structure

Containers in pyomo.kernel are analogous to indexed components in pyomo.environ. However, pyomo.kernel containers allow for additional layers of structure as they can be nested within each other as long as they have compatible categories. The following example shows this using pyomo.kernel.variable containers.

```
vlist = pyomo.kernel.variable_list()
vlist.append(pyomo.kernel.variable_dict())
vlist[0]['x'] = pyomo.kernel.variable()
```

As the next section will show, the standard modeling component containers are also compatible with user-defined classes that derive from the existing modeling components.

Sub-Classing

The existing components and containers in pyomo.kernel are designed to make sub-classing easy. User-defined classes that derive from the standard modeling components and containers in pyomo.kernel are compatible with existing containers of the same component category. As an example, in the following code we see that the pyomo.kernel.block_list container can store both pyomo.kernel.block objects as well as a user-defined Widget object that derives from pyomo.kernel.block. The Widget object can also be placed on another block object as an attribute and treated itself as a block.

[1] pyomo.kernel does not include an alternative to the AbstractModel component from pyomo.environ. All data necessary to build a model must be imported by the user.

[2] pyomo.kernel does not include an alternative to the Pyomo Set component from pyomo.environ.

[3] pyomo.kernel.parameter objects are always mutable.

[4] Special Ordered Sets

[5] Both pyomo.kernel.piecewise and pyomo.kernel.piecewise_nd create objects that are sub-classes of pyomo.kernel.block. Thus, these objects can be stored in containers such as pyomo.kernel.block_dict and pyomo.kernel.block_list.

```
class Widget(pyomo.kernel.block):
    ...

model = pyomo.kernel.block()
model.blist = pyomo.kernel.block_list()
model.blist.append(Widget())
model.blist.append(pyomo.kernel.block())
model.w = Widget()
model.w.x = pyomo.kernel.variable()
```

The next series of examples goes into more detail on how to implement derived components or containers.

The following code block shows a class definition for a non-negative variable, starting from `pyomo.kernel.variable` as a base class.

```
class NonNegativeVariable(pyomo.kernel.variable):
    """A non-negative variable."""

    __slots__ = ()

    def __init__(self, **kwds):
        if 'lb' not in kwds:
            kwds['lb'] = 0
        if kwds['lb'] < 0:
            raise ValueError("lower bound must be non-negative")
        super(NonNegativeVariable, self).__init__(**kwds)

    #
    # restrict assignments to x.lb to non-negative numbers
    #
    @property
    def lb(self):
        # calls the base class property getter
        return pyomo.kernel.variable.lb.fget(self)

    @lb.setter
    def lb(self, lb):
        if lb < 0:
            raise ValueError("lower bound must be non-negative")
        # calls the base class property setter
        pyomo.kernel.variable.lb.fset(self, lb)
```

The `NonNegativeVariable` class prevents negative values from being stored into its lower bound during initialization or later on through assignment statements (e.g, `x.lb = -1` fails). Note that the `__slots__ == ()` line at the beginning of the class definition is optional, but it is recommended if no additional data members are necessary as it reduces the memory requirement of the new variable type.

The next code block defines a custom variable container called `Point` that represents a 3-dimensional point in Cartesian space. The new type derives from the `pyomo.kernel.variable_tuple` container and uses the `NonNegativeVariable` type we defined previously in the *z* coordinate.

```
class Point(pyomo.kernel.variable_tuple):
```

(continues on next page)

(continued from previous page)

```
    """A 3-dimensional point in Cartesian space with the
    z coordinate restricted to non-negative values."""

    __slots__ = ()

    def __init__(self):
        super(Point, self).__init__(
            (pyomo.kernel.variable(), pyomo.kernel.variable(), NonNegativeVariable())
        )

    @property
    def x(self):
        return self[0]

    @property
    def y(self):
        return self[1]

    @property
    def z(self):
        return self[2]
```

The `Point` class can be treated like a tuple storing three variables, and it can be placed inside of other variable containers or added as attributes to blocks. The property methods included in the class definition provide an additional syntax for accessing the three variables it stores, as the next code example will show.

The following code defines a class for building a convex second-order cone constraint from a `Point` object. It derives from the `pyomo.kernel.constraint` class, overriding the constructor to build the constraint expression and utilizing the property methods on the point class to increase readability.

```
class SOC(pyomo.kernel.constraint):
    """A convex second-order cone constraint"""

    __slots__ = ()

    def __init__(self, point):
        assert isinstance(point.z, NonNegativeVariable)
        super(SOC, self).__init__(point.x**2 + point.y**2 <= point.z**2)
```

Reduced Memory Usage

The `pyomo.kernel` library offers significant opportunities to reduce memory requirements for highly structured models. The situation where this is most apparent is when expressing a model in terms of many small blocks consisting of singleton components. As an example, consider expressing a model consisting of a large number of voltage transformers. One option for doing so might be to define a *Transformer* component as a subclass of `pyomo.kernel.block`. The example below defines such a component, including some helper methods for connecting input and output voltage variables and updating the transformer ratio.

```
class Transformer(pyomo.kernel.block):
    def __init__(self):
        super(Transformer, self).__init__()
        self._a = pyomo.kernel.parameter()
        self._v_in = pyomo.kernel.expression()
        self._v_out = pyomo.kernel.expression()
        self._c = pyomo.kernel.constraint(self._a * self._v_out == self._v_in)

    def set_ratio(self, a):
        assert a > 0
        self._a.value = a

    def connect_v_in(self, v_in):
        self._v_in.expr = v_in

    def connect_v_out(self, v_out):
        self._v_out.expr = v_out
```

A simplified version of this using `pyomo.environ` components might look like what is below.

```
def Transformer():
    b = pyo.Block(concrete=True)
    b._a = pyo.Param(mutable=True)
    b._v_in = pyo.Expression()
    b._v_out = pyo.Expression()
    b._c = pyo.Constraint(expr=b._a * b._v_out == b._v_in)
    return b
```

The transformer expressed using `pyomo.kernel` components requires roughly 2 KB of memory, whereas the `pyomo.environ` version requires roughly 8.4 KB of memory (an increase of more than 4x). Additionally, the `pyomo.kernel` transformer is fully compatible with all existing `pyomo.kernel` block containers.

Direct Support For Conic Constraints with Mosek

Pyomo 5.6.3 introduced support into `pyomo.kernel` for six conic constraint forms that are directly recognized by the new Mosek solver interface. These are

- `conic.quadratic`:

  $$\sum_i x_i^2 \leq r^2, \quad r \geq 0$$

- `conic.rotated_quadratic`:

  $$\sum_i x_i^2 \leq 2 r_1 r_2, \quad r_1, r_2 \geq 0$$

- `conic.primal_exponential`:

  $$x_1 \exp(x_2/x_1) \leq r, \quad x_1, r \geq 0$$

- `conic.primal_power` (α is a constant):

  $$||x||_2 \leq r_1^{\alpha} r_2^{1-\alpha}, \quad r_1, r_2 \geq 0, \; 0 < \alpha < 1$$

- `conic.dual_exponential`:

$$-x_2\exp((x_1/x_2)-1)\leq r,\quad x_2\leq 0,\ r\geq 0$$

- `conic.dual_power` (α is a constant):

$$||x||_2\leq (r_1/\alpha)^{\alpha}(r_2/(1-\alpha))^{1-\alpha},\quad r_1,r_2\geq 0,\ 0<\alpha<1$$

Other solver interfaces will treat these objects as general nonlinear or quadratic constraints, and may or may not have the ability to identify their convexity. For instance, Gurobi will recognize the expressions produced by the `quadratic` and `rotated_quadratic` objects as representing convex domains as long as the variables involved satisfy the convexity conditions. However, other solvers may not include this functionality.

Each of these conic constraint classes are of the same category type as standard `pyomo.kernel.constraint` object, and, thus, are directly supported by the standard constraint containers (`constraint_tuple`, `constraint_list`, `constraint_dict`).

Each conic constraint class supports two methods of instantiation. The first method is to directly instantiate a conic constraint object, providing all necessary input variables:

```
import pyomo.kernel as pmo

m = pmo.block()
m.x1 = pmo.variable(lb=0)
m.x2 = pmo.variable()
m.r = pmo.variable(lb=0)
m.q = pmo.conic.primal_exponential(x1=m.x1, x2=m.x2, r=m.r)
```

This method may be limiting if utilizing the Mosek solver as the user must ensure that additional conic constraints do not use variables that are directly involved in any existing conic constraints (this is a limitation the Mosek solver itself).

To overcome this limitation, and to provide a more general way of defining conic domains, each conic constraint class provides the `as_domain` class method. This alternate constructor has the same argument signature as the class, but in place of each variable, one can optionally provide a constant, a linear expression, or `None`. The `as_domain` class method returns a `block` object that includes the core conic constraint, auxiliary variables used to express the conic constraint, as well as auxiliary constraints that link the inputs (that are not `None`) to the auxiliary variables. Example:

```
import pyomo.kernel as pmo
import math

m = pmo.block()
m.x = pmo.variable(lb=0)
m.y = pmo.variable(lb=0)
m.b = pmo.conic.primal_exponential.as_domain(
    x1=math.sqrt(2) * m.x, x2=2.0, r=2 * (m.x + m.y)
)
```

3.7.2 Future Solver Interface Changes

Note

The new solver interfaces are still under active development. They are included in the releases as development previews. Please be aware that APIs and functionality may change with no notice.

We welcome any feedback and ideas as we develop this capability. Please post feedback on Issue 1030.

Pyomo offers interfaces into multiple solvers, both commercial and open source. To support better capabilities for solver interfaces, the Pyomo team is actively redesigning the existing interfaces to make them more maintainable and

intuitive for use. A preview of the redesigned interfaces can be found in `pyomo.contrib.solver`.

New Interface Usage

The new interfaces are not completely backwards compatible with the existing Pyomo solver interfaces. However, to aid in testing and evaluation, we are distributing versions of the new solver interfaces that are compatible with the existing ("legacy") solver interface. These "legacy" interfaces are registered with the current `SolverFactory` using slightly different names (to avoid conflicts with existing interfaces).

Table 3.20: Available Redesigned Solvers and Names Registered in the SolverFactories

Solver	Name registered in the `pyomo.contrib.solver.common.factory.SolverFactory`	Name registered in the `pyomo.opt.base.solvers.LegacySolverFactory`
Ipopt	`ipopt`	`ipopt_v2`
GAMS	`gams`	`gams_v2`
Gurobi (persistent)	`gurobi_persistent`	`gurobi_persistent_v2`
Gurobi (direct)	`gurobi_direct`	`gurobi_direct_v2`
HiGHS	`highs`	`highs`
KNITRO	`knitro_direct`	`knitro_direct`
SCIP (direct)	`scip_direct`	`scip_direct`
SCIP (persistent)	`scip_persistent`	`scip_persistent`

Using the new interfaces through the legacy interface

Here we use the new interface as exposed through the existing (legacy) solver factory and solver interface wrapper. This provides an API that is compatible with the existing (legacy) Pyomo solver interface and can be used with other Pyomo tools / capabilities.

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(model):
    return (1.0 - model.x) ** 2 + 100.0 * (model.y - model.x**2) ** 2

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

status = pyo.SolverFactory('ipopt_v2').solve(model)
pyo.assert_optimal_termination(status)
model.pprint()
```

In keeping with our commitment to backwards compatibility, both the legacy and future methods of specifying solver options are supported:

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(model):
    return (1.0 - model.x) ** 2 + 100.0 * (model.y - model.x**2) ** 2

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

# Backwards compatible
status = pyo.SolverFactory('ipopt_v2').solve(model, options={'max_iter' : 6})
# Forwards compatible
status = pyo.SolverFactory('ipopt_v2').solve(model, solver_options={'max_iter' : 6})
model.pprint()
```

Using the new interfaces directly

Here we use the new interface by importing it directly:

```
# Direct import
import pyomo.environ as pyo
from pyomo.contrib.solver.solvers.ipopt import Ipopt

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(model):
    return (1.0 - model.x) ** 2 + 100.0 * (model.y - model.x**2) ** 2

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

opt = Ipopt()
status = opt.solve(model)
pyo.assert_optimal_termination(status)
# Displays important results information; only available through the new interfaces
status.display()
model.pprint()
```

Using the new interfaces through the "new" SolverFactory

Here we use the new interface by retrieving it from the new `SolverFactory`:

```
# Import through new SolverFactory
import pyomo.environ as pyo
from pyomo.contrib.solver.common.factory import SolverFactory

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)
```

(continues on next page)

(continued from previous page)

```
def rosenbrock(model):
    return (1.0 - model.x) ** 2 + 100.0 * (model.y - model.x**2) ** 2

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

opt = SolverFactory('ipopt')
status = opt.solve(model)
pyo.assert_optimal_termination(status)
# Displays important results information; only available through the new interfaces
status.display()
model.pprint()
```

Switching all of Pyomo to use the new interfaces

We also provide a mechanism to get a "preview" of the future where we replace the existing (legacy) SolverFactory and utilities with the new (development) version (see *Accessing preview features*):

```
# Change default SolverFactory version
import pyomo.environ as pyo
from pyomo.__future__ import solver_factory_v3

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(model):
    return (1.0 - model.x) ** 2 + 100.0 * (model.y - model.x**2) ** 2

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

status = pyo.SolverFactory('ipopt').solve(model)
pyo.assert_optimal_termination(status)
# Displays important results information; only available through the new interfaces
status.display()
model.pprint()
```

Linear Presolve and Scaling

The new interface allows access to new capabilities in the various problem writers, including the linear presolve and scaling options recently incorporated into the redesigned NL writer. For example, you can control the NL writer in the new `ipopt` interface through the solver's `writer_config` configuration option (see the *Ipopt* interface documentation).

```
from pyomo.contrib.solver.solvers.ipopt import Ipopt
opt = Ipopt()
opt.config.writer_config.display()
```

```
show_section_timing: false
skip_trivial_constraints: true
file_determinism: FileDeterminism.ORDERED
```

(continues on next page)

(continued from previous page)

```
symbolic_solver_labels: false
scale_model: true
export_nonlinear_variables: null
row_order: null
column_order: null
export_defined_variables: true
linear_presolve: true
```

Note that, by default, both `linear_presolve` and `scale_model` are enabled. Users can manipulate `linear_presolve` and `scale_model` to their preferred states by changing their values.

```
>>> opt.config.writer_config.linear_presolve = False
```

Interface Implementation

All new interfaces should be built upon one of two classes (currently): *SolverBase* or *PersistentSolverBase*.

All solvers should have the following:

class pyomo.contrib.solver.common.base.**SolverBase**(***kwds*)

The base class for "new-style" Pyomo solver interfaces.

This base class defines the methods all derived solvers are expected to implement:

- *available()*
- *is_persistent()*
- *solve()*
- *version()*

Class Configuration

All derived concrete implementations of this class must define a class attribute `CONFIG` containing the `ConfigDict` that specifies the solver's configuration options. By convention, the `CONFIG` should derive from (or implement a superset of the options from) one of the following:

- *SolverConfig*
- *BranchAndBoundConfig*
- *PersistentSolverConfig*
- *PersistentBranchAndBoundConfig*

classmethod api_version()

Return the public API supported by this interface.

Returns
A solver API enum object

Return type
SolverAPIVersion

available() → *Availability*

Test if the solver is available on this system.

Nominally, this will return *True* if the solver interface is valid and can be used to solve problems and *False* if it cannot. Note that for licensed solvers there are a number of "levels" of available: depending on the license, the solver may be available with limitations on problem size or runtime (e.g., 'demo' vs. 'community' vs.

'full'). In these cases, the solver may return a subclass of enum.IntEnum, with members that resolve to True if the solver is available (possibly with limitations). The Enum may also have multiple members that all resolve to False indicating the reason why the interface is not available (not found, bad license, unsupported version, etc).

Returns
available – An enum that indicates "how available" the solver is. Note that the enum can be cast to bool, which will be True if the solver is runnable at all and False otherwise.

Return type
Availability

`is_persistent()` → bool

`True` if this supports a persistent interface to the solver.

Returns
is_persistent – True if the solver is a persistent solver.

Return type
bool

`solve`(*model:* BlockData, ***kwargs*) → *Results*

Solve a Pyomo model.

Parameters

- **model** (`BlockData`) – The Pyomo model to be solved
- ****kwargs** – Additional keyword arguments (including solver_options - passthrough options; delivered directly to the solver (with no validation))

Returns
results – A results object

Return type
Results

`version()` → tuple

Return the solver version found on the system.

Returns
version – A tuple representing the version

Return type
tuple

`config`

Instance configuration; see CONFIG documentation on derived class

Persistent solvers include additional members as well as other configuration options:

`class pyomo.contrib.solver.common.base.PersistentSolverBase`(***kwds*)

Bases: *SolverBase*

Base class upon which persistent solvers can be built. This inherits the methods from the solver base class and adds those methods that are necessary for persistent solvers.

Example usage can be seen in the Gurobi interface.

`add_block`(*block:* BlockData)

Add a block to the model.

add_constraints(*cons: list[*ConstraintData*]*)

Add constraints to the model.

is_persistent() → bool

Returns

is_persistent – True if the solver is a persistent solver.

Return type

bool

remove_block(*block:* BlockData)

Remove a block from the model.

remove_constraints(*cons: list[*ConstraintData*]*)

Remove constraints from the model.

set_instance(*model:* BlockData)

Set an instance of the model.

set_objective(*obj:* ObjectiveData)

Set current objective for the model.

solve(*model:* BlockData, ***kwargs*) → *Results*

Solve a Pyomo model.

Parameters

- **model** (BlockData) – The Pyomo model to be solved
- ****kwargs** – Additional keyword arguments (including solver_options - passthrough options; delivered directly to the solver (with no validation))

Returns

results – A results object

Return type

Results

update_parameters()

Update parameters on the model.

update_variables(*variables: list[*VarData*]*)

Update variables on the model.

Results

Every solver, at the end of a *solve* call, will return a *Results* object. This object is a *pyomo.common.config.ConfigDict*, which can be manipulated similar to a standard `dict` in Python.

class pyomo.contrib.solver.common.results.**Results**(*description=None*, *doc=None*, *implicit=False*, *implicit_domain=None*, *visibility=0*)

Bases: *ConfigDict*

Base class for all solver results

display(*content_filter=None*, *indent_spacing=2*, *ostream=None*, *visibility=0*)

Print the current Config value, in YAML format.

The current values stored in this Config object are output to `ostream` (or `sys.stdout` if `ostream` is `None`). If `visibility` is not `None`, then only items with visibility less than or equal to `visibility` will be output. Output can be further filtered by providing a `content_filter`.

extra_info: *ConfigDict*

incumbent_objective: float | None

objective_bound: float | None

solution_status: *SolutionStatus*

solver_config: *ConfigDict*

solver_log: str

solver_name: str | None

solver_version: Tuple[int, ...] | None

termination_condition: *TerminationCondition*

timing_info: *ConfigDict*

The new interface has condensed *SolverStatus*, *TerminationCondition*, and *SolutionStatus* into *TerminationCondition* and *SolutionStatus* to reduce complexity. As a result, several legacy SolutionStatus values are no longer achievable. These are detailed in the table below.

Table 3.21: Mapping from unachievable SolutionStatus to future statuses

Legacy SolutionStatus	*TerminationCondition*	*SolutionStatus*
other	unknown	noSolution
unsure	unknown	noSolution
locallyOptimal	convergenceCriteriaSatisfied	optimal
globallyOptimal	convergenceCriteriaSatisfied	optimal
bestSoFar	convergenceCriteriaSatisfied	feasible

Termination Conditions

Pyomo offers a standard set of termination conditions to map to solver returns. The intent of *TerminationCondition* is to notify the user of why the solver exited. The user is expected to inspect the *Results* object or any returned solver messages or logs for more information.

class pyomo.contrib.solver.common.results.**TerminationCondition**(**values*)

Bases: Enum

An Enum that enumerates all possible exit statuses for a solver call.

Solution Status

Pyomo offers a standard set of solution statuses to map to solver output. The intent of *SolutionStatus* is to notify the user of what the solver returned at a high level. The user is expected to inspect the *Results* object or any returned solver messages or logs for more information.

class pyomo.contrib.solver.common.results.**SolutionStatus**(**values*)

Bases: Enum

An enumeration for interpreting the result of a termination. This describes the designated status by the solver to be loaded back into the model.

Solution

Solutions can be loaded back into a model using a `SolutionLoader`. A specific loader should be written for each unique case. Several have already been implemented. For example, for `ipopt`:

class `pyomo.contrib.solver.solvers.ipopt.`**`IpoptSolutionLoader`**(*sol_data:* ASLSolFileData, *nl_info:* NLWriterInfo, *pyomo_model*)

Bases: *ASLSolFileSolutionLoader*

`get_duals`(*cons_to_load: Sequence[*ConstraintData*]* | *None* = *None*) → dict[*ConstraintData*, float]

Returns a dictionary mapping constraint to dual value.

Parameters

cons_to_load (*Sequence[*`ConstraintData`*]*) – A list of the constraints whose duals should be retrieved. If *cons_to_load* is `None`, then the duals for all constraints will be retrieved.

Returns

duals – Maps constraints to dual values

Return type

dict[*ConstraintData*, float]

`get_number_of_solutions`() → int

The number of solutions available through this `SolutionLoader`

Returns

num_solutions – Indicates the number of solutions found

Return type

int

`get_reduced_costs`(*vars_to_load: Sequence[*VarData*]* | *None* = *None*) → Mapping[*VarData*, float]

Returns a ComponentMap mapping variable to reduced cost.

Parameters

vars_to_load (*Sequence[*`VarData`*]*) – A list of the variables whose reduced cost should be retrieved. If *vars_to_load* is `None`, then the reduced costs for all variables will be retrieved.

Returns

reduced_costs – Maps variables to reduced costs

Return type

ComponentMap[*VarData*, float]

`get_solution_ids`() → list[Any]

Return the list of available solution identdiers.

If there are multiple solutions available, this will return a list of the solution identifiers that can be passed to *`solution()`* to activate individual solutions from the solver's solution pool. If only one solution is available, this will return `[None]`. If no solutions are available, this will return `[]`

Returns

solutions_ids – The identifiers for multiple solutions

Return type

list[Any]

`get_vars`(*vars_to_load: Sequence[*VarData*]* | *None* = *None*) → Mapping[*VarData*, float]

Returns a ComponentMap mapping variable to var value.

Parameters
: **vars_to_load** (*Sequence[*VarData*]*) – A list of the Pyomo variables whose solution value should be retrieved. If *vars_to_load* is None, then the values for all variables will be retrieved.

Returns
: **primals** – Maps variables to solution values

Return type
: *ComponentMap*[*VarData*, float]

load_import_suffixes()
: Clear import suffixes on the model and load data returned by the solver.

load_solution() → None
: Load the solution (everything that can be) back into the model

load_vars(*vars_to_load: Sequence[*VarData*] | None = None*) → None
: Load the primal variable values at the solution into the Pyomo model `Var` objects

Parameters
: **vars_to_load** (*Sequence[*VarData*]*) – A list of the minimum set of Pyomo variables whose solution should be loaded. If *vars_to_load* is None, then the solution to all primal variables will be loaded. Even if *vars_to_load* is specified, the values of other variables may also be loaded depending on the interface.

solution(*solution_id: Any*) → *SolutionLoaderView*
: Return a view object that can be used to access a specific solution

The resulting `SolutionLoaderView` object can be used in two ways. First, as a context manager:

```
results = solver.solve(model)
with results.solution(2) as soln:
    soln.load_vars()
    soln.load_import_suffixes()
```

or

```
results = solver.solve(model)
with results.solution(2):
    results.load_vars()
    results.load_import_suffixes()
```

Or as if it were a `SolutionLoader`:

Parameters
: **solution_id** (*Any*) – The solution identifier to "activate" and make available

Dual Sign Convention

For all future solver interfaces, Pyomo adopts the following sign convention. Given the problem

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & c_i(x) = 0 \quad \forall i \in \mathcal{E} \\ & g_i(x) \leq 0 \quad \forall i \in \mathcal{U} \\ & h_i(x) \geq 0 \quad \forall i \in \mathcal{L} \end{aligned}$$

We define the Lagrangian as

$$L(x, \lambda, \nu, \delta) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i \, c_i(x) - \sum_{i \in \mathcal{U}} \nu_i \, g_i(x) - \sum_{i \in \mathcal{L}} \delta_i \, h_i(x)$$

Then, the KKT conditions are [NW99]

$$\begin{aligned}
\nabla_x L(x, \lambda, \nu, \delta) &= 0 \\
c(x) &= 0 \\
g(x) &\leq 0 \\
h(x) &\geq 0 \\
\nu &\leq 0 \\
\delta &\geq 0 \\
\nu_i \, g_i(x) &= 0 \\
\delta_i \, h_i(x) &= 0
\end{aligned}$$

Note that this sign convention is based on the `(lower, body, upper)` representation of constraints rather than the expression provided by a user. Users can specify constraints with variables on both the left- and right-hand sides of equalities and inequalities. However, the `(lower, body, upper)` representation ensures that all variables appear in the `body`, matching the form of the problem above.

For maximization problems of the form

$$\begin{aligned}
\max \quad & f(x) \\
\text{s.t.} \quad & c_i(x) = 0 \quad \forall i \in \mathcal{E} \\
& g_i(x) \leq 0 \quad \forall i \in \mathcal{U} \\
& h_i(x) \geq 0 \quad \forall i \in \mathcal{L}
\end{aligned}$$

we define the Lagrangian to be the same as above:

$$L(x, \lambda, \nu, \delta) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i \, c_i(x) - \sum_{i \in \mathcal{U}} \nu_i \, g_i(x) - \sum_{i \in \mathcal{L}} \delta_i \, h_i(x)$$

As a result, the signs of the duals change. The KKT conditions are

$$\begin{aligned}
\nabla_x L(x, \lambda, \nu, \delta) &= 0 \\
c(x) &= 0 \\
g(x) &\leq 0 \\
h(x) &\geq 0 \\
\nu &\geq 0 \\
\delta &\leq 0 \\
\nu_i \, g_i(x) &= 0 \\
\delta_i \, h_i(x) &= 0
\end{aligned}$$

Pyomo also supports "range constraints" which are inequalities with both upper and lower bounds, where the bounds are not equal. For example,

$$-1 \leq x + y \leq 1$$

These are handled very similarly to variable bounds in terms of dual sign conventions. For these, at most one "side" of the inequality can be active at a time. If neither side is active, then the dual will be zero. If the dual is nonzero, then the dual corresponds to the side of the constraint that is active. The dual for the other side will be implicitly zero. When accessing duals, the keys are the constraints. As a result, there is only one key for a range constraint, even though it is really two constraints. Therefore, the dual for the inactive side will not be reported explicitly. Again, the sign convention is based on the `(lower, body, upper)` representation of the constraint. Therefore, the left side of this inequality belongs to $\mathcal{L}$ and the right side belongs to $\mathcal{U}$.

CHAPTER

FOUR

REFERENCE GUIDES

4.1 Topical Reference

Pyomo is being increasingly used as a library to support Python scripts. This section describes library APIs for key elements of Pyomo's core library. This documentation serves as a reference for both (1) Pyomo developers and (2) advanced users who are developing Python scripts using Pyomo.

4.1.1 AML Library Reference

The following modeling components make up the core of the Pyomo Algebraic Modeling Language (AML). These classes are all available through the *pyomo.environ* namespace.

ConcreteModel(*args, **kwds)	A concrete optimization model that does not defer construction of components.
AbstractModel(*args, **kwds)	An abstract optimization model that defers construction of components.
Block(*args, **kwds)	Blocks are indexed components that contain other components (including blocks).
Set(*args, **kwds)	A component used to index other Pyomo components.
RangeSet(*args, **kwds)	A set object that represents a set of numeric values
Param(*args, **kwds)	A parameter value, which may be defined over an index.
Var(*args, **kwargs)	A numeric variable, which may be defined over an index.
Objective(*args, **kwds)	This modeling component defines an objective expression.
Constraint(*args, **kwds)	This modeling component defines a constraint expression using a rule function.
ExternalFunction(*args, **kwargs)	Interface to an external (non-algebraic) function.
Reference(reference[, ctype])	Creates a component that references other components
SOSConstraint(*args, **kwds)	Implements constraints for special ordered sets (SOS).

4.1.2 Expression Reference

Utilities to Build Expressions

pyomo.core.util.prod(terms)	A utility function to compute the product of a list of terms.
pyomo.core.util.quicksum(args[, start, linear])	A utility function to compute a sum of Pyomo expressions.
pyomo.core.util.sum_product(*args, **kwds)	A utility function to compute a generalized dot product.

continues on next page

Table 4.2 – continued from previous page

pyomo.core.util.summation(*args, **kwds)	An alias for `sum_product`
pyomo.core.util.dot_product(*args, **kwds)	An alias for `sum_product`

Utilities to Manage and Analyze Expressions

Functions

`pyomo.core.expr.expression_to_string`(expr[, ...])	Return a string representation of an expression.
`pyomo.core.expr.decompose_term`(expr)	A function that returns a tuple consisting of (1) a flag indicating whether the expression is linear, and (2) a list of tuples that represents the terms in the linear expression.
`pyomo.core.expr.clone_expression`(expr[, ...])	A function that is used to clone an expression.
`pyomo.core.expr.evaluate_expression`(exp[, ...])	Evaluate the value of the expression.
`pyomo.core.expr.identify_components`(expr, ...)	A generator that yields a sequence of nodes in an expression tree that belong to a specified set.
`pyomo.core.expr.identify_variables`(expr[, ...])	A generator that yields a sequence of variables in an expression tree.
`pyomo.core.expr.differentiate`(expr[, wrt, ...])	Return derivative of expression.

Classes

pyomo.core.expr.symbol_map.SymbolMap([labeler])	A class for tracking assigned labels for modeling components.

Context Managers

`pyomo.core.expr.nonlinear_expression`()	Context manager for mutable nonlinear sums.
`pyomo.core.expr.linear_expression`()	Context manager for mutable linear sums.

Core Classes

The following are the two core classes documented here:

- *NumericValue*
- *NumericExpression*

The remaining classes are the public classes for expressions, which developers may need to know about. The methods for these classes are not documented because they are described in the *NumericExpression* class.

Sets with Expression Types

The following sets can be used to develop visitor patterns for Pyomo expressions.

native_numeric_types	Python set used to identify numeric constants.
native_types	Python set used to identify numeric constants and related native types.

continues on next page

Table 4.6 – continued from previous page

nonpyomo_leaf_types	Python set used to identify numeric constants, boolean values, strings and instances of *NonNumericValue*, which is commonly used in code that walks Pyomo expression trees.

NumericValue and NumericExpression

NumericValue()	This is the base class for numeric values used in Pyomo.
NumericExpression(args)	The base class for Pyomo expressions.

Other Public Classes

NegationExpression(args)	Negation expressions.
AbsExpression(arg)	An expression object for the `abs()` function.
UnaryFunctionExpression(args[, name, fcn])	An expression object for intrinsic (math) functions (e.g. sin, cos, tan).
ProductExpression(args)	Product expressions.
DivisionExpression(args)	Division expressions.
SumExpression(args)	Sum expression.
Expr_ifExpression(args)	A numeric ternary (if-then-else) expression.
ExternalFunctionExpression(args[, fcn])	External function expressions
pyomo.core.expr.relational_expr.EqualityExpression(args)	Equality expression.
pyomo.core.expr.relational_expr.InequalityExpression(...)	Inequality expressions, which define less-than or less-than-or-equal relations.
pyomo.core.expr.relational_expr.RangedExpression(...)	Ranged expressions, which define relations with a lower and upper bound.
pyomo.core.expr.template_expr.GetItemExpression([args])	Expression to call `__getitem__()` on the base object.

Visitor Classes

pyomo.core.expr.StreamBasedExpressionVisitor	This class implements a generic stream-based expression walker.
pyomo.core.expr.ExpressionValueVisitor()	
pyomo.core.expr.ExpressionReplacementVisitor	

4.1.3 Solver Interfaces

GAMS

GAMSShell Solver

GAMSShell(**kwds)	A generic shell interface to GAMS solvers.

continues on next page

Table 4.10 – continued from previous page

GAMSShell.available([exception_flag])	True if the solver is available.
GAMSShell.executable()	Returns the executable used by this solver.
GAMSShell.solve(*args, **kwds)	Solve a model via the GAMS executable.
GAMSShell.version()	Returns a 4-tuple describing the solver executable version.
GAMSShell.warm_start_capable()	True is the solver can accept a warm-start solution.

GAMSDirect Solver

GAMSDirect(**kwds)	A generic python interface to GAMS solvers.
GAMSDirect.available([exception_flag])	True if the solver is available.
GAMSDirect.solve(*args, **kwds)	Solve a model via the GAMS Python API.
GAMSDirect.version()	Returns a 4-tuple describing the solver executable version.
GAMSDirect.warm_start_capable()	True is the solver can accept a warm-start solution.

GAMS Writer

This class is most commonly accessed and called upon via model.write("filename.gms", ...), but is also utilized by the GAMS solver interfaces.

ProblemWriter_gams()	

CPLEXPersistent

pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent(**kwds)	A class that provides a persistent interface to Cplex.

GurobiDirect

Interface

GurobiDirect([manage_env])	A direct interface to Gurobi using gurobipy.

Methods

GurobiDirect.available([exception_flag])	Returns True if the solver is available.
GurobiDirect.close()	Frees local Gurobi resources used by this solver instance.
GurobiDirect.close_global()	Frees all Gurobi models used by this solver, and frees the global default Gurobi environment.
GurobiDirect.solve(*args, **kwds)	Solve the problem
GurobiDirect.version()	Returns a 4-tuple describing the solver executable version.

GurobiPersistent

Interface

GurobiPersistent(**kwds)	A class that provides a persistent interface to Gurobi.

Methods

GurobiPersistent.add_block(block)	Add a single Pyomo Block to the solver's model.
GurobiPersistent.add_constraint(con)	Add a single constraint to the solver's model.
GurobiPersistent.set_objective(obj)	Set the solver's objective.
GurobiPersistent.add_sos_constraint(con)	Add a single SOS constraint to the solver's model (if supported).
GurobiPersistent.add_var(var)	Add a single variable to the solver's model.
GurobiPersistent.available([exception_flag])	Returns True if the solver is available.
GurobiPersistent.has_capability(cap)	Returns a boolean value representing whether a solver supports a specific feature.
GurobiPersistent.has_instance()	True if set_instance has been called and this solver interface has a pyomo model and a solver model.
GurobiPersistent.load_vars([vars_to_load])	Load the values from the solver's variables into the corresponding pyomo variables.
GurobiPersistent.problem_format()	Returns the current problem format.
GurobiPersistent.remove_block(block)	Remove a single block from the solver's model.
GurobiPersistent.remove_constraint(con)	Remove a single constraint from the solver's model.
GurobiPersistent.remove_sos_constraint(con)	Remove a single SOS constraint from the solver's model.
GurobiPersistent.remove_var(var)	Remove a single variable from the solver's model.
GurobiPersistent.reset()	Reset the state of the solver
GurobiPersistent.results_format()	Returns the current results format.
GurobiPersistent.set_callback([func])	Specify a callback for gurobi to use.
GurobiPersistent.set_instance(model, **kwds)	This method is used to translate the Pyomo model provided to an instance of the solver's Python model.
GurobiPersistent.set_problem_format(format)	Set the current problem format (if it's valid) and update the results format to something valid for this problem format.
GurobiPersistent.set_results_format(format)	Set the current results format (if it's valid for the current problem format).
GurobiPersistent.solve(*args, **kwds)	Solve the model.
GurobiPersistent.update_var(var)	Update a single variable in the solver's model.
GurobiPersistent.version()	Returns a 4-tuple describing the solver executable version.
GurobiPersistent.write(filename)	Write the model to a file (e.g., and lp file).

XpressPersistent

pyomo.solvers.plugins.solvers.xpress_persistent.XpressPersistent(**kwds)	A class that provides a persistent interface to Xpress.

4.1.4 Model Data Management

class pyomo.dataportal.DataPortal.**DataPortal**(**args*, ***kwds*)

An object that manages loading and storing data from external data sources. This object interfaces to plugins that manipulate the data in a manner that is dependent on the data format.

Internally, the data in a DataPortal object is organized as follows:

```
data[namespace][symbol][index] -> value
```

All data is associated with a symbol name, which may be indexed, and which may belong to a namespace. The default namespace is None.

Parameters

- **model** – The model for which this data is associated. This is used for error checking (e.g. object names must exist in the model, set dimensions must match, etc.). Default is None.
- **filename** (*str*) – A file from which data is loaded. Default is None.
- **data_dict** (*dict*) – A dictionary used to initialize the data in this object. Default is None.

__getitem__(**args*)

Return the specified data value.

If a single argument is given, then this is the symbol name:

```
dp = DataPortal()
dp[name]
```

If a two arguments are given, then the first is the namespace and the second is the symbol name:

```
dp = DataPortal()
dp[namespace, name]
```

Parameters

***args** (*str*) – A tuple of arguments.

Returns

If a single argument is given, then the data associated with that symbol in the namespace None is returned. If two arguments are given, then the data associated with symbol in the given namespace is returned.

__init__(**args*, ***kwds*)

Constructor

__setitem__(*name*, *value*)

Set the value of name with the given value.

Parameters

- **name** (*str*) – The name of the symbol that is set.
- **value** – The value of the symbol.

connect(***kwds*)

Construct a data manager object that is associated with the input source. This data manager is used to process future data imports and exports.

Parameters

- **filename** (*str*) – A filename that specifies the data source. Default is `None`.
- **server** (*str*) – The name of the remote server that hosts the data. Default is `None`.
- **using** (*str*) – The name of the resource used to load the data. Default is `None`.

Other keyword arguments are passed to the data manager object.

data(*name=None*, *namespace=None*)

Return the data associated with a symbol and namespace

Parameters

- **name** (*str*) – The name of the symbol that is returned. Default is `None`, which indicates that the entire data in the namespace is returned.
- **namespace** (*str*) – The name of the namespace that is accessed. Default is `None`.

Returns

If `name` is `None`, then the dictionary for the namespace is returned. Otherwise, the data associated with `name` in given namespace is returned. The return value is a constant if `None` if there is a single value in the symbol dictionary, and otherwise the symbol dictionary is returned.

disconnect()

Close the data manager object that is associated with the input source.

items(*namespace=None*)

Return an iterator of (name, value) tuples from the data in the specified namespace.

Yields

The next (name, value) tuple in the namespace. If the symbol has a simple data value, then that is included in the tuple. Otherwise, the tuple includes a dictionary mapping symbol indices to values.

keys(*namespace=None*)

Return an iterator of the data keys in the specified namespace.

Yields

A string name for the next symbol in the specified namespace.

load(***kwds*)

Import data from an external data source.

Parameters

model – The model object for which this data is associated. Default is `None`.

Other keyword arguments are passed to the `connect()` method.

namespaces()

Return an iterator for the namespaces in the data portal.

Yields

A string name for the next namespace.

store(***kwds*)

Export data to an external data source.

Parameters

model – The model object for which this data is associated. Default is `None`.

Other keyword arguments are passed to the `connect()` method.

values(*namespace=None*)

Return an iterator of the data values in the specified namespace.

Yields

The data value for the next symbol in the specified namespace. This may be a simple value, or a dictionary of values.

__weakref__

list of weak references to the object

class pyomo.dataportal.TableData.**TableData**

A class used to read/write data from/to a table in an external data source.

__init__()

Constructor

add_options(***kwds*)

Add the keyword options to the `Options` object in this object.

available()

Returns

Return `True` if the data manager is available.

clear()

Clear the data that was extracted from this table

close()

Close the data manager.

initialize(***kwds*)

Initialize the data manager with keyword arguments.

The *filename* argument is recognized here, and other arguments are passed to the `add_options()` method.

open()

Open the data manager.

process(*model*, *data*, *default*)

Process the data that was extracted from this data manager and return it.

read()

Read data from the data manager.

write(*data*)

Write data to the data manager.

__weakref__

list of weak references to the object

4.1.5 APPSI

Auto-Persistent Pyomo Solver Interfaces

APPSI Base Classes

pyomo.contrib.appsi.base.TerminationCondition(value)	An enumeration for checking the termination condition of solvers
pyomo.contrib.appsi.base.Results()	Base class for all APPSI solver results
pyomo.contrib.appsi.base.Solver()	
pyomo.contrib.appsi.base.PersistentSolver()	
pyomo.contrib.appsi.base.SolverConfig([...])	Common configuration options for all APPSI solver interfaces
pyomo.contrib.appsi.base.MIPSolverConfig([...])	Configuration options common to all MIP solvers
pyomo.contrib.appsi.base.UpdateConfig([...])	Config options common to all persistent solvers

Solvers

Gurobi

Handling Gurobi licenses through the APPSI interface

In order to obtain performance benefits when re-solving a Pyomo model with Gurobi repeatedly, Pyomo has to keep a reference to a gurobipy model between calls to *solve()*. Depending on the Gurobi license type, this may "consume" a license as long as any APPSI-Gurobi interface exists (i.e., has not been garbage collected). To release a Gurobi license for other processes, use the `release_license()` method as shown below. Note that `release_license()` must be called on every instance for this to actually release the license. However, releasing the license will delete the gurobipy model which will have to be reconstructed from scratch the next time *solve()* is called, negating any performance benefit of the persistent solver interface.

```
>>> opt = appsi.solvers.Gurobi()
>>> results = opt.solve(model)
>>> opt.release_license()
```

Also note that both the *available()* and *solve()* methods will construct a gurobipy model, thereby (depending on the type of license) "consuming" a license. The *available()* method has to do this so that the availability does not change between calls to *available()* and *solve()*, leading to unexpected errors.

pyomo.contrib.appsi.solvers.gurobi.GurobiResults(solver)	
pyomo.contrib.appsi.solvers.gurobi.Gurobi([...])	Interface to Gurobi

Ipopt

pyomo.contrib.appsi.solvers.ipopt.IpoptConfig([...])
pyomo.contrib.appsi.solvers.ipopt.Ipopt([...])

Cplex

Cbc

pyomo.contrib.appsi.solvers.cbc.CbcConfig([...])	
pyomo.contrib.appsi.solvers.cbc.Cbc([...])	

HiGHS

pyomo.contrib.appsi.solvers.highs.HighsResults(solver)	
pyomo.contrib.appsi.solvers.highs.Highs([...])	Interface to HiGHS

MAiNGO

pyomo.contrib.appsi.solvers.maingo.MAiNGOConfig([...])	
pyomo.contrib.appsi.solvers.maingo.MAiNGO([...])	Interface to MAiNGO

APPSI solver interfaces are designed to work very similarly to most Pyomo solver interfaces but are very efficient for resolving the same model with small changes. This is very beneficial for applications such as Benders' Decomposition, Optimization-Based Bounds Tightening, Progressive Hedging, Outer-Approximation, and many others. Here is an example of using an APPSI solver interface.

```
>>> import pyomo.environ as pyo
>>> from pyomo.contrib import appsi
>>> import numpy as np
>>> from pyomo.common.timing import HierarchicalTimer
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.y = pyo.Var()
>>> m.p = pyo.Param(mutable=True)
>>> m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
>>> m.c1 = pyo.Constraint(expr=m.y >= pyo.exp(m.x))
>>> m.c2 = pyo.Constraint(expr=m.y >= (m.x - m.p)**2)
>>> opt = appsi.solvers.Ipopt()
>>> timer = HierarchicalTimer()
>>> for p_val in np.linspace(1, 10, 100):
>>>     m.p.value = float(p_val)
>>>     res = opt.solve(m, timer=timer)
```

(continues on next page)

(continued from previous page)

```
>>>     assert res.termination_condition == appsi.base.TerminationCondition.optimal
>>>     print(res.best_feasible_objective)
>>> print(timer)
```

Alternatively, you can access the APPSI solvers through the classic `SolverFactory` using the pattern `appsi_solvername`.

```
>>> import pyomo.environ as pyo
>>> opt_ipopt = pyo.SolverFactory('appsi_ipopt')
>>> opt_highs = pyo.SolverFactory('appsi_highs')
```

Extra performance improvements can be made if you know exactly what changes will be made in your model. In the example above, only parameter values are changed, so we can setup the *UpdateConfig* so that the solver does not check for changes in variables or constraints.

```
>>> timer = HierarchicalTimer()
>>> opt.update_config.check_for_new_or_removed_constraints = False
>>> opt.update_config.check_for_new_or_removed_vars = False
>>> opt.update_config.update_constraints = False
>>> opt.update_config.update_vars = False
>>> for p_val in np.linspace(1, 10, 100):
>>>     m.p.value = float(p_val)
>>>     res = opt.solve(m, timer=timer)
>>>     assert res.termination_condition == appsi.base.TerminationCondition.optimal
>>>     print(res.best_feasible_objective)
>>> print(timer)
```

Solver independent options can be specified with the *SolverConfig* or derived classes. For example:

```
>>> opt.config.stream_solver = True
```

Solver specific options can be specified with the `solver_options()` attribute. For example:

```
>>> opt.solver_options['max_iter'] = 20
```

Installation

There are a few ways to install Appsi listed below.

Option1:

```
pyomo build-extensions
```

Option2:

```
cd pyomo/contrib/appsi/
python build.py
```

Option3:

```
python
>>> from pyomo.contrib.appsi.build import build_appsi
>>> build_appsi()
```

Pyomo is under active ongoing development. The following API documentation describes *Beta* functionality.

Warning

The `pyomo.kernel` API is still in the beta phase of development. It is fully tested and functional; however, the interface may change as it becomes further integrated with the rest of Pyomo.

Warning

Models built with `pyomo.kernel` components are not yet compatible with pyomo extension modules (e.g., `PySP`, `pyomo.dae`, `pyomo.gdp`).

4.1.6 The Kernel Library API Reference

Modeling Components:

Blocks

Summary

pyomo.core.kernel.block.block()	A generalized container for defining hierarchical models by adding modeling components as attributes.
pyomo.core.kernel.block.block_tuple(*args, ...)	A tuple-style container for objects with category type IBlock
pyomo.core.kernel.block.block_list(*args, **kwds)	A list-style container for objects with category type IBlock
pyomo.core.kernel.block.block_dict(*args, **kwds)	A dict-style container for objects with category type IBlock

Variables

Summary

pyomo.core.kernel.variable.variable([...])	A decision variable
pyomo.core.kernel.variable.variable_tuple(...)	A tuple-style container for objects with category type IVariable
pyomo.core.kernel.variable.variable_list(...)	A list-style container for objects with category type IVariable
pyomo.core.kernel.variable.variable_dict(...)	A dict-style container for objects with category type IVariable

Constraints

Summary

pyomo.core.kernel.constraint.constraint([...])	A general algebraic constraint

continues on next page

Table 4.28 – continued from previous page

pyomo.core.kernel.constraint.linear_constraint([...])	A linear constraint
pyomo.core.kernel.constraint.constraint_tuple(...)	A tuple-style container for objects with category type IConstraint
pyomo.core.kernel.constraint.constraint_list(...)	A list-style container for objects with category type IConstraint
pyomo.core.kernel.constraint.constraint_dict(...)	A dict-style container for objects with category type IConstraint
pyomo.core.kernel.matrix_constraint.matrix_constraint(A)	A container for constraints of the form lb <= Ax <= ub.

Parameters

Summary

pyomo.core.kernel.parameter.parameter([value])	A object for storing a mutable, numeric value that can be used to build a symbolic expression.
pyomo.core.kernel.parameter.functional_value([fn])	An object for storing a numeric function that can be used in a symbolic expression.
pyomo.core.kernel.parameter.parameter_tuple(...)	A tuple-style container for objects with category type IParameter
pyomo.core.kernel.parameter.parameter_list(...)	A list-style container for objects with category type IParameter
pyomo.core.kernel.parameter.parameter_dict(...)	A dict-style container for objects with category type IParameter

Objectives

Summary

pyomo.core.kernel.objective.objective([...])	An optimization objective.
pyomo.core.kernel.objective.objective_tuple(...)	A tuple-style container for objects with category type IObjective
pyomo.core.kernel.objective.objective_list(...)	A list-style container for objects with category type IObjective
pyomo.core.kernel.objective.objective_dict(...)	A dict-style container for objects with category type IObjective

Expressions

Summary

pyomo.core.kernel.expression.expression([expr])	A named, mutable expression.
pyomo.core.kernel.expression.expression_tuple(...)	A tuple-style container for objects with category type IExpression

continues on next page

Table 4.31 – continued from previous page

pyomo.core.kernel.expression.expression_list(...)	A list-style container for objects with category type IExpression
pyomo.core.kernel.expression.expression_dict(...)	A dict-style container for objects with category type IExpression

Special Ordered Sets

Summary

pyomo.core.kernel.sos.sos(variables[, ...])	A Special Ordered Set of type n.
pyomo.core.kernel.sos.sos1(variables[, weights])	A Special Ordered Set of type 1.
pyomo.core.kernel.sos.sos2(variables[, weights])	A Special Ordered Set of type 2.
pyomo.core.kernel.sos.sos_tuple(*args, **kwds)	A tuple-style container for objects with category type ISOS
pyomo.core.kernel.sos.sos_list(*args, **kwds)	A list-style container for objects with category type ISOS
pyomo.core.kernel.sos.sos_dict(*args, **kwds)	A dict-style container for objects with category type ISOS

Suffixes

pyomo.core.kernel.suffix

Piecewise Function Library

Modules

Single-variate Piecewise Functions

Summary

pyomo.core.kernel.piecewise_library.transforms.piecewise(...)	Models a single-variate piecewise linear function.
pyomo.core.kernel.piecewise_library.transforms.PiecewiseLinearFunction(...)	A piecewise linear function
pyomo.core.kernel.piecewise_library.transforms.TransformedPiecewiseLinearFunction(f)	Base class for transformed piecewise linear functions
pyomo.core.kernel.piecewise_library.transforms.piecewise_convex(...)	Simple convex piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_sos2(...)	Discrete SOS2 piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_dcc(...)	Discrete DCC piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_cc(...)	Discrete CC piecewise representation

continues on next page

Table 4.34 – continued from previous page

pyomo.core.kernel.piecewise_library.transforms.piecewise_mc(...)	Discrete MC piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_inc(...)	Discrete INC piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_dlog(...)	Discrete DLOG piecewise representation
pyomo.core.kernel.piecewise_library.transforms.piecewise_log(...)	Discrete LOG piecewise representation

Multi-variate Piecewise Functions

Summary

pyomo.core.kernel.piecewise_library.transforms_nd.piecewise_nd(...)	Models a multi-variate piecewise linear function.
pyomo.core.kernel.piecewise_library.transforms_nd.PiecewiseLinearFunctionND(...)	A multi-variate piecewise linear function
pyomo.core.kernel.piecewise_library.transforms_nd.TransformedPiecewiseLinearFunctionND(f)	Base class for transformed multi-variate piecewise linear functions
pyomo.core.kernel.piecewise_library.transforms_nd.piecewise_nd_cc(...)	Discrete CC multi-variate piecewise representation

Utilities for Piecewise Functions

pyomo.core.kernel.piecewise_library.util

Conic Constraints

A collection of classes that provide an easy and performant way to declare conic constraints. The Mosek solver interface includes special handling of these objects that recognizes them as convex constraints. Other solver interfaces will treat these objects as general nonlinear or quadratic expressions, and may or may not have the ability to identify their convexity.

Summary

pyomo.core.kernel.conic.quadratic(r, x)	A quadratic conic constraint of the form:
pyomo.core.kernel.conic.rotated_quadratic(r1, ...)	A rotated quadratic conic constraint of the form:
pyomo.core.kernel.conic.primal_exponential(r, ...)	A primal exponential conic constraint of the form:
pyomo.core.kernel.conic.primal_power(r1, r2, ...)	A primal power conic constraint of the form:
pyomo.core.kernel.conic.dual_exponential(r, ...)	A dual exponential conic constraint of the form:
pyomo.core.kernel.conic.dual_power(r1, r2, ...)	A dual power conic constraint of the form:

Base API:

Base Object Storage Interface

pyomo.core.kernel.base

Homogeneous Object Containers

pyomo.core.kernel.homogeneous_container

Heterogeneous Object Containers

pyomo.core.kernel.heterogeneous_container

Containers:

Tuple-like Object Storage

pyomo.core.kernel.tuple_container.TupleContainer(*args)	A partial implementation of the IHomogeneousContainer interface that provides tuple-like storage functionality.

List-like Object Storage

pyomo.core.kernel.list_container.ListContainer(*args)	A partial implementation of the IHomogeneousContainer interface that provides list-like storage functionality.

Dict-like Object Storage

pyomo.core.kernel.dict_container.DictContainer(...)	A partial implementation of the IHomogeneousContainer interface that provides dict-like storage functionality.

4.2 Development Principles

The Pyomo development team follows a set of development principles. In order to promote overall transparency, this page is intended to document those principles to the best of our ability, for users and potential contributors alike. Please also review Pyomo's recent publication on the history of its development for a holistic view into the changes of these principles over time [MHJ+25].

- *Backwards Compatibility*
  - *Commitment to Published APIs*
  - *Core APIs*
  - *Stable Extensions*
  - *Experimental Functionality*
  - *Unsupported Functionality*
- *Dependency Management*
  - *Minimal Core Dependencies*
  - *Optional Dependencies*
  - *Optional Dependency Groups*
  - *Solvers*
- *Miscellaneous Conventions*

4.2.1 Backwards Compatibility

Commitment to Published APIs

We treat functionality and examples published in the most recent edition of the Pyomo book [PyomoBookIII] ("The Book") as our public API commitment. The interfaces and APIs appearing in The Book will be supported (although possibly in a deprecated form) until the next major Pyomo release, which will generally coincide with a new edition of the book.

This commitment ensures that teaching materials, training resources, and long-term codebases built following those examples will remain valid across an entire major release.

Core APIs

Functionality that is part of the Pyomo source tree but not explicitly included in the book is also expected to be stable if it resides outside the `pyomo.addons`, `pyomo.devel`, and `pyomo.unsupported` namespaces. This functionality is referred to as "core" by the Pyomo development team.

When changes to core APIs become necessary, we will endeavor to follow one (or both) of the following steps:

1. **Deprecation warnings** are added in advance of functionality removal. These are visible to users at import or execution time, with clear guidance on replacement functionality. For core functionality not mentioned in the published Pyomo book, deprecated interfaces are generally expected to remain available for **at least two minor Pyomo releases** following the introduction of the deprecation warning. For example, functionality deprecated in release `X.Y.Z` should not be removed before release `X.(Y+2).0`.
2. **Relocation warnings** are provided for any relocated functionality. These modules import from their old locations and print a warning about the relocation in order to assist users' transition.

Ideally, changes in this fashion allow users and downstream packages to adapt gradually without abrupt breakage.

Note

For detailed guidance on how to mark functionality for deprecation or removal within the Pyomo codebase, see *Deprecation and Removal of Functionality*.

Stable Extensions

The pyomo.addons namespace contains extensions that are intended to be **mostly stable and reliable for downstream use**, while remaining outside the Pyomo core.

Functionality in pyomo.addons is expected to follow Pyomo's coding, testing, documentation, and backward-compatibility standards. While not held to the same guarantees as core APIs or Book-published interfaces, users should be able to rely on pyomo.addons functionality across minor Pyomo releases.

Experimental Functionality

The pyomo.devel namespace contains experimental or rapidly evolving functionality intended for active research, prototyping, and early-stage development.

APIs under pyomo.devel may change or be removed between releases without deprecation warnings. Users should not rely on functionality in this namespace for production workflows.

Unsupported Functionality

The pyomo.unsupported namespace contains code that no longer has an active maintainer or future development plans.

Functionality under this namespace may not work and is **NOT** routinely tested through the standard Pyomo test harness. No compatibility or stability guarantees are provided.

Historical Note

Earlier versions of Pyomo placed all experimental and non-core functionality under a single namespace, pyomo.contrib. While this approach enabled rapid sharing of new modeling tools and research code, it made it difficult for users to distinguish between stable, maintained functionality and experimental or unsupported features.

The new namespace structure (pyomo.addons, pyomo.devel, and pyomo.unsupported) has been introduced to provide clearer signals about stability, maintenance expectations, and compatibility guarantees, while preserving Pyomo's long-standing support for community-driven development.

This historical namespace is documented here for context only. New development should follow the current namespace guidelines described above.

4.2.2 Dependency Management

Minimal Core Dependencies

The core Pyomo codebase is designed to be a *Pure Python* library with minimal dependencies outside the standard Python library (currently, there are no hard external dependencies).

This approach simplifies installation, reduces the burden on derived packages, and lessens the likelihood of triggering dependency conflicts. Additionally, this allows users to install and run Pyomo in resource-constrained or isolated environments (such as teaching containers or HPC systems).

Optional Dependencies

Some extended Pyomo functionality relies on additional optional Python packages. An optional dependency must not be imported (or required) for the Pyomo environment. That is:

```
import pyomo.environ
```

should not raise an ImportError if the dependency is missing. Further, the Pyomo test harness (pytest pyomo) must run without error/failure if any optional dependencies are missing (except for the dependencies required by the tests *dependency group*).

Pyomo makes extensive use of `attempt_import()` to support the standardized and convenient use of optional dependencies. Further, many common dependencies are directly importable through `pyomo.common.dependencies` without immediately triggering the dependency import; for example:

```
# Importing numpy from dependencies does not trigger the import
from pyomo.common.dependencies import numpy as np, numpy_available

# but testing the availability or using the module will trigger the import
if numpy_available:
    a = np.array([1, 2, 3])
```

Optional Dependency Groups

Pyomo defines three dependency groups to simplify installation of optional dependencies:

- **tests** – Dependencies needed only to run the automated test suite and continuous integration infrastructure (e.g., `pytest`, `parameterized`, `coverage`).
- **docs** – Dependencies needed to build the Sphinx-based documentation (e.g., `sphinx`, `sphinx_rtd_theme`, `numpydoc`).
- **optional** – Dependencies that enable extended functionality throughout the Pyomo codebase, such as numerical computations or data handling (e.g., `numpy`, `pandas`, `matplotlib`).

These optional dependencies can be installed selectively using standard `pip` commands. For example:

```
pip install pyomo[optional]
pip install pyomo[tests,docs]
```

Pyomo's continuous integration infrastructure regularly tests against the most recent versions of all optional dependencies to ensure that Pyomo remains compatible with current releases in the Python ecosystem. When incompatibilities are identified, the setup configuration is updated accordingly.

Note

For more information on installing Pyomo with optional extras, see *Installation*.

Solvers

Pyomo does not bundle or directly distribute optimization solvers. We recognize that solver installation can be challenging for new users. To assist with this process, see the solver availability table and installation guidance in *Using Solvers with Pyomo*. This table lists solvers that can be installed via `pip` or `conda` where available, but Pyomo itself does not include or require any specific solver as a dependency.

4.2.3 Miscellaneous Conventions

There are a variety of long-standing conventions that have become standard across the project. This list will be amended as conventions come up, so please refer to it regularly for updates:

- **Fail loudly:** Silent failure is strongly discouraged. Code should defensively guard against unexpected or unsupported cases and raise explicit exceptions (e.g., `NotImplementedError`) rather than silently producing incorrect or ambiguous results. Some of the most difficult Pyomo bugs to diagnose arise from silently incorrect mathematics.
- **Document and enforce assumptions:** When behavior that a user could reasonably expect is ambiguous, code should clearly document the assumptions being made and fail loudly when those assumptions are violated.

- **Print statements:** Avoid printing or writing directly to `stdout`. Pyomo is a library, not an application, and copious output can interfere with downstream tools and workflows. Use the appropriate logger instead. Print information only when the user has enabled or requested it.
- **Logging:** Use Python's `logging` framework for diagnostic and informational output. Loggers should generally be created using `logging.getLogger(__name__)` to ensure messages are properly scoped and can be enabled or filtered at the module or package level.
- **Active components define the model:** Pyomo models are defined as the set of *active* components reachable from the root `Block`. Not all modeling objects are active components (notably, `Var` objects are not). Algorithms and writers should respect this distinction when traversing or interpreting models.
- **Avoid iterating over variables directly:** Direct iteration over variables using `m.component_data_objects(Var, ...)` is rarely appropriate, as it returns all variables declared on the model hierarchy regardless of their mathematical relevance. When gathering variables associated with a model formulation, prefer utilities such as `get_vars_from_components` with `Constraint` or `(Constraint, Objective)` as the component types.
- **Writers must validate component types:** Writers and solver-facing infrastructure should explicitly detect and warn about active Pyomo components they do not recognize. Pyomo supports extended modeling environments (e.g., DAE and GDP), and silently ignoring unexpected structures can result in invalid solver input. The utility `categorize_valid_components` in `pyomo.repn.util` may be used to assist with this validation.
- **Do not use names as identifiers:** Component names and strings should generally not be used to track Pyomo components within algorithms or writers. Names are not guaranteed to be unique or stable across model transformations. Prefer data structures from `pyomo.common.collections` that support using components directly as keys.
- **Environment imports:** Import the main Pyomo environment as `import pyomo.environ as pyo`. Avoid all uses of `import *`.
- **Export lists:** Do not define `__all__` in modules. Public symbols are determined by naming and documentation, not explicit lists.
- **Circular imports:** Make every effort to avoid circular imports. When circular imports are absolutely necessary, leverage `attempt_import()` to explicitly break the cycle. To help with this, some module namespaces have additional requirements:
  - `pyomo.common`: modules within *pyomo.common* *must* not import *any* Pyomo modules outside of *pyomo.common*
  - `pyomo.core.expr`: modules within *pyomo.core.expr* should not import modules outside of *pyomo.common* or *pyomo.core.expr*.
  - `pyomo.core.base`: modules within *pyomo.core.base* should not import modules outside of *pyomo.common*, *pyomo.core.expr*, or *pyomo.core.base*.
- **Naming conventions:** Follow PEP 8 naming conventions, including descriptive `snake_case` for functions and variables and `PascalCase` for classes. Functions should generally be named as verb phrases, while classes should be noun-like representations of concepts.
- **Avoid code duplication:** Repeated or copy-pasted code is strongly discouraged. Duplication increases maintenance burden and often leads to inconsistent behavior. When code patterns begin to repeat, contributors are encouraged to refactor common functionality or discuss design alternatives with the core development team.
- **URLs:** All links in code, comments, and documentation must use `https` rather than `http` wherever possible.
- **File headers:** Every `.py` file must begin with the standard Pyomo copyright header:

```
# ___________________________________________________________________________
↪__
```

(continues on next page)

(continued from previous page)

```
#
# Pyomo: Python Optimization Modeling Objects
# Copyright (c) 2008-2026 National Technology and Engineering Solutions of Sandia,␣
↪LLC
# Under the terms of Contract DE-NA0003525 with National Technology and Engineering
# Solutions of Sandia, LLC, the U.S. Government retains certain rights in this
# software.  This software is distributed under the 3-clause BSD License.
# ___________________________________________________________________________
↪__
```

Update the year range as appropriate when modifying files.

- **Full commit history:** We do **not** squash-merge Pull Requests, preferring to retain the entire commit history.
- **Pull Request naming:** Pull Request titles are added to the CHANGELOG and the release notes. The Pyomo development team reserves the right to alter titles as appropriate to ensure they fit the look and feel of other titles in the CHANGELOG.

4.3 Accessing preview features

4.3.1 Preview capabilities through `pyomo.__future__`

This module provides a uniform interface for gaining access to future ("preview") capabilities that are either slightly incompatible with the current official offering, or are still under development with the intent to replace the current offering.

Currently supported `__future__` offerings include:

`solver_factory()`	Get (or set) the active implementation of the SolverFactory

4.4 Common Warnings/Errors

4.4.1 Warnings

W1001: Setting Var value not in domain

When setting `Var` values (by either calling `Var.set_value()` or setting the `value` attribute), Pyomo will validate the incoming value by checking that the value is `in` the `Var.domain`. Any values not in the domain will generate this warning:

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(domain=pyo.Integers)
>>> m.x = 0.5
WARNING (W1001): Setting Var 'x' to a value `0.5` (float) not in domain
     Integers.
     See also https://pyomo.readthedocs.io/en/stable/errors.html#w1001
>>> print(m.x.value)
0.5
```

Users can bypass all domain validation by setting the value using:

```
>>> m.x.set_value(0.75, skip_validation=True)
>>> print(m.x.value)
0.75
```

W1002: Setting Var value outside the bounds

When setting `Var` values (by either calling `set_value()` or setting the `value` attribute), Pyomo will validate the incoming value by checking that the value is within the range specified by `Var.bounds`. Any values outside the bounds will generate this warning:

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var(domain=pyo.Integers, bounds=(1, 5))
>>> m.x = 0
WARNING (W1002): Setting Var 'x' to a numeric value `0` outside the bounds
    (1, 5).
    See also https://pyomo.readthedocs.io/en/stable/errors.html#w1002
>>> print(m.x.value)
0
```

Users can bypass all domain validation by setting the value using:

```
>>> m.x.set_value(10, skip_validation=True)
>>> print(m.x.value)
10
```

W1003: Unexpected RecursionError walking an expression tree

Pyomo leverages a recursive walker (the *StreamBasedExpressionVisitor*) to traverse (walk) expression trees. For most expressions, this recursive walker is the most efficient. However, Python has a relatively shallow recursion limit (generally, 1000 frames). The recursive walker is designed to monitor the stack depth and cleanly switch to a nonrecursive walker before hitting the stack limit. However, there are two (rare) cases where the Python stack limit can still generate a `RecursionError` exception:

1. Starting the walker with fewer than `pyomo.core.expr.visitor.RECURSION_LIMIT` available frames.
2. Callbacks that require more than 2 * `pyomo.core.expr.visitor.RECURSION_LIMIT` frames.

The (default) recursive walker will catch the exception and restart the walker from the beginning in non-recursive mode, issuing this warning. The caution is that any partial work done by the walker before the exception was raised will be lost, potentially leaving the walker in an inconsistent state. Users can avoid this by

- avoiding recursive callbacks
- restructuring the system design to avoid triggering the walker with few available stack frames
- directly calling the *walk_expression_nonrecursive()* walker method

```
>>> import sys
>>> import pyomo.core.expr.visitor as visitor
>>> from pyomo.core.tests.unit.test_visitor import fill_stack
>>> expression_depth = visitor.StreamBasedExpressionVisitor(
...     exitNode=lambda node, data: max(data) + 1 if data else 1)
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> @m.Expression(range(35))
... def e(m, i):
```

(continues on next page)

(continued from previous page)

```
...     return m.e[i-1] if i else m.x
>>> expression_depth.walk_expression(m.e[34])
36
>>> fill_stack(sys.getrecursionlimit() - visitor.get_stack_depth() - 30,
...            expression_depth.walk_expression,
...            m.e[34])
WARNING (W1003): Unexpected RecursionError walking an expression tree.
    See also https://pyomo.readthedocs.io/en/stable/errors.html#w1003
36
>>> fill_stack(sys.getrecursionlimit() - visitor.get_stack_depth() - 30,
...            expression_depth.walk_expression_nonrecursive,
...            m.e[34])
36
```

4.4.2 Errors

E2001: Variable domains must be an instance of a Pyomo Set

Variable domains are always Pyomo `Set` or `RangeSet` objects. This includes global sets like `Reals`, `Integers`, `Binary`, `NonNegativeReals`, etc., as well as model-specific `Set` instances. The `Var.domain` setter will attempt to convert assigned values to a Pyomo *Set*, with any failures leading to this warning (and an exception from the converter):

```
>>> m = pyo.ConcreteModel()
>>> m.x = pyo.Var()
>>> m.x.domain = 5
Traceback (most recent call last):
   ...
TypeError: Cannot create a Set from data that does not support __contains__...
ERROR (E2001): 5 is not a valid domain. Variable domains must be an instance
    of a Pyomo Set or convertible to a Pyomo Set.
    See also https://pyomo.readthedocs.io/en/stable/errors.html#e2001
```

4.5 Related Packages

The following is a list of software packages[1] that utilize or build off of Pyomo. This is certainly not a comprehensive list. Please reach out to the Pyomo developers if you would like your project added here.

4.5.1 Modeling Extensions

Package Name	Link	Description
Coramin	https://github.com/coramin/coramin	A suite of tools for developing MINLP algorithms
PAO	https://github.com/or-fusion/pao	Formulation and solution of multilevel optimization problems
OMLT	https://github.com/cog-imperial/OMLT	Represent machine learning models within an optimization formulation

[1] Please note that the Pyomo team does not evaluate nor necessarily endorse the listed packages.

4.5.2 Solvers and Solution Strategies

Package Name	Link	Description
Galini	https://github.com/cog-imperial/galini	An extensible, Python-based MIQCQP Solver
mpi-sppy	https://github.com/pyomo/mpi-sppy	Parallel solution of stochastic programming problems
Parapint	https://github.com/parapint/parapint	Parallel solution of structured NLPs.
Suspect	https://github.com/cog-imperial/suspect	FBBT and convexity detection

4.5.3 Domain-Specific Applications

Package Name	Link	Description
Chama	https://github.com/sandialabs/chama	Sensor placement optimization
Egret	https://github.com/grid-parity-exchange/egret	Formulation and solution of unit commitment and optimal power flow problems
IDAES	https://github.com/idaes/idaes-pse	Institute for the Design of Advanced Energy Systems
Prescient	https://github.com/grid-parity-exchange/prescient	Production Cost Model for power systems simulation and analysis
PrOMMiS	https://github.com/prommis/prommis	Process Optimization and Modeling for Minerals Sustainability
WaterTAP	https://github.com/watertap-org/watertap	Water treatment process modeling

4.6 Publications

These publications describe various Pyomo capabilities or subpackages:

4.7 Bibliography

CHAPTER

FIVE

PYOMO RESOURCES

Pyomo development is hosted at GitHub:

- https://github.com/Pyomo/pyomo

See the Pyomo Forum for online discussions of Pyomo or to ask a question:

- http://groups.google.com/group/pyomo-forum/

Ask a question on StackOverflow using the `#pyomo` tag:

- https://stackoverflow.com/questions/ask?tags=pyomo

Additional Pyomo tutorials and examples can be found at the following links:

- Pyomo — Optimization Modeling in Python ([PyomoBookIII])
- Pyomo Workshop Slides and Exercises
- Prof. Jeffrey Kantor's Pyomo Cookbook
- The companion notebooks for *Hands-On Mathematical Optimization with Python*
- Pyomo Gallery

CHAPTER

SIX

CONTRIBUTING TO PYOMO

Interested in contributing code or documentation to the project? Check out our *Contribution Guide*

CHAPTER

SEVEN

RELATED PACKAGES

Pyomo is a key dependency for a number of other software packages for specific domains or customized solution strategies. A non-comprehensive list of Pyomo-related packages may be found *here*.

CHAPTER EIGHT

CITING PYOMO

If you use Pyomo in your work, please cite:

> Bynum, Michael L., Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. Pyomo - Optimization Modeling in Python, 3rd Edition. Springer, 2021.

Additionally, several Pyomo capabilities and subpackages are described in further detail in separate *Publications*.

BIBLIOGRAPHY

[Pyomo-paper] William E. Hart, Jean-Paul Watson, David L. Woodruff. “Pyomo: modeling and solving mathematical programs in Python,” Mathematical Programming Computation, 3(3), August 2011.

[PyomoBookI] William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff. Pyomo – Optimization Modeling in Python, Springer Optimization and Its Applications, Vol 67. Springer. 2012.

[PyomoBookII] William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, John D. Siirola. Pyomo - Optimization Modeling in Python, 2nd Edition. Springer Optimization and Its Applications, Vol 67. Springer. 2017.

[PyomoBookIII] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. Pyomo - Optimization Modeling in Python, 3rd Edition. Vol. 67. Springer. 2021. DOI 10.1007/978-3-030-68928-5

[PyomoDAE-paper] Bethany Nicholson, John D. Siirola, Jean-Paul Watson, Victor M. Zavala, and Lorenz T. Biegler. “pyomo.dae: a modeling and automatic discretization framework for optimization with differential and algebraic equations”, *Mathematical Programming Computation*, 10(2), 187-223. 2018.

[PyomoDOE-paper] Wang, Jialu, and Alexander W. Dowling. “Pyomo.DOE: An open-source package for model-based design of experiments in Python”, *AIChE Journal*, 68(12), e17813. 2022. DOI 10.1002/aic.17813

[Parmest-paper] Katherine A. Klise, Bethany L. Nicholson, Andrea Staid, David L.Woodruff. “Parmest: Parameter Estimation Via Pyomo.” *Computer Aided Chemical Engineering*, 47, 41-46. 2019.

[PyomoGDP-paper] Qi Chen, Emma S. Johnson, David E. Bernal, Romeo Valentin, Sunjeev Kale, Johnny Bates, John D. Siirola, and Ignacio E. Grossmann. “Pyomo.GDP: an ecosystem for logic based modeling and optimization development.” *Optimization and Engineering*, 1-36. 2021. DOI 10.1007/s11081-021-09601-7

[PyomoGDP-proceedings] Qi Chen, Emma S. Johnson, John D. Siirola, and Ignacio E. Grossmann. “Pyomo.GDP: Disjunctive Models in Python.” In M. R. Eden, M. G. Ierapetritou, and G. P. Towler (Eds.), *Proceedings of the 13th International Symposium on Process Systems Engineering*, 889–894, 2018. DOI 10.1016/B978-0-444-64241-7.50143-9

[AIMMS] http://www.aimms.com/

[AM00] O. Abel and W. Marquardt, “Scenario-integrated modeling and optimization of dynamic systems”, *AIChE Journal*, 46(4). 2000.

[Bal85] E. Balas. “Disjunctive Programming and a Hierarchy of Relaxations for Discrete Optimization Problems”, *SIAM Journal on Algebraic Discrete Methods*, 6(3), 466–486, 1985. DOI 10.1137/0606047

[BJ72] E. Balas and R. Jeroslow. “Canonical Cuts on the Unit Hypercube”, *SIAM Journal on Applied Mathematics* 23(1), 61-19, 1972. DOI 10.1137/0123007

[BS04] D. Bertsimas and M. Sim. “The price of robustness”, *Operations research*, 52(1), 35-53, 2004. DOI 10.1287/opre.1030.0065.

[Dje20] H. Djelassi. “Discretization-based algorithms for the global solution of hierarchical programs”. Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, 2020. DOI 10.18154/RWTH-2020-09163.

[FGK02] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*, 2nd Edition, Duxbury Press, 2002.

[GAMS] http://www.gams.com

[GLM99] A. Grothey, S. Leyffer, and K. I. M. McKinnon. “A note on feasibility in Benders Decomposition”, Numerical Analysis Report NA/188, Dundee University. 1999.

[GT13] I. E. Grossmann and F. Trespalacios. “Systematic modeling of discrete-continuous optimization models through generalized disjunctive programming”, *AIChE Journal*, 59(9), 3276–3295. 2013. DOI 10.1002/aic.14088

[HG83] K. P. Halemane and I. E. Grossmann. “Optimal process design under uncertainty”, *AIChE Journal*, 29(3), 425–433. 1983. DOI 10.1002/aic.690290312

[IAE+21] N. M. Isenberg, P. Akula, J. C. Eslick, D. Bhattacharyya, D. C. Miller, and C. E. Gounaris. “A generalized cutting-set approach for nonlinear robust optimization in process systems engineering”, *AIChE Journal*, 67:e17175. 2021. DOI 10.1002/aic.17175

[KMM+23] B. Knueven, D. Mildebrath, C. Muir, J. D. Siirola, J.-P. Watson, and D. L. Woodruff. “A Parallel Hub-and-Spoke System for Large-Scale Scenario-Based Optimization Under Uncertainty”, *Math Programming Computation*, 15, 591-619. 2023. DOI 10.1007/s12532-023-00247-3

[KMT21] J. Kronqvist, R. Misener, and C. Tsay. “Between Steps: Intermediate Relaxations between big-M and Convex Hull Reformulations”. 2021. https://arxiv.org/abs/2101.12708

[MHJ+25] M. Mundt, W. E. Hart, E. S. Johnson, B. Nicholson, and J. D. Siirola. “Pyomo: Accidentally outrunning the bear”, *Patterns*, 6(7), 101311. 2025. ISSN 2666-3899. DOI 10.1016/j.patter.2025.101311

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*, New York: Wiley. 1988.

[RB01] W. C. Rooney and L. T. Biegler. “Design for model parameter uncertainty using nonlinear confidence regions”, *AIChE Journal*, 47(8). 2001. DOI 10.1002/aic.690470811

[RG94] R. Raman and I. E. Grossmann. “Modelling and computational techniques for logic based integer programming”, *Computers and Chemical Engineering*, 18(7), 563–578. 1994. DOI 10.1016/0098-1354(93)E0010-7

[SG03] N. W. Sawaya and I. E. Grossmann. “A cutting plane method for solving linear generalized disjunctive programming problems”, *Computer Aided Chemical Engineering*, 15(C), 1032–1037. 2003. DOI 10.1016/S1570-7946(03)80444-3

[TG15] F. Trespalacios and I. E. Grossmann. “Improved Big-M reformulation for generalized disjunctive programs”, *Computers and Chemical Engineering*, 76, 98–103. 2015. DOI 10.1016/j.compchemeng.2015.02.013

[VAN10] J. P. Vielma, S. Ahmed, and G. Nemhauser. “Mixed-Integer Models for Non-separable Piecewise Linear Optimization: Unifying framework and Extensions”, *Operations Research* 58(2), 303-315. 2010.

[Vie15] J. P. Vielma. “Mixed Integer Linear Programming Formluation Techniques”, *SIAM Review*, 57(1), 3-57. 2015. DOI 10.1137/130915303

[YLH18] Y. Yuan, Z. Li, and B. Huang. “Nonlinear robust optimization for process design”, *AIChE Journal*, 64(2), 481–494. 2018. DOI 10.1002/aic.15950

[NW99] Nocedal, Jorge, and Stephen J. Wright, eds. Numerical optimization. New York, NY: Springer New York, 1999.