Pyomo Online Documentation 4.1

COLLABORATORS			
	TITLE : Pyomo Online Documentation 4.1		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	William E. Hart and David L. Woodruff	June 6, 2016	

Contents

Preface

This book provides a quick introduction to Pyomo, which includes a collection of Python software packages that supports a diverse set of optimization capabilities for formulating and analyzing optimization models. A central component of Pyomo is Pyomo, which supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with algebraic modeling languages (AMLs), which support the description and analysis of mathematical models with a high-level language. Although most AMLs are implemented in custom modeling languages, Pyomo's modeling objects are embedded within Python, a full-featured high-level programming language that contains a rich set of supporting libraries.

Pyomo has also proven an effective framework for developing high-level optimization and analysis tools. For example, the PySP package provides generic solvers for stochastic programming. PySP leverages the fact that Pyomo's modeling objects are embedded within a full-featured high-level programming language, which allows for transparent parallelization of subproblems using Python parallel communication libraries.

Goals of the Book

This book provides a broad overview of different components of the Pyomo software. There are roughly two main goals for this book:

1. Help users get started with different Pyomo capabilities. Our goal is not to provide a comprehensive reference, but rather to provide a tutorial with simple and illustrative examples. Also, we aim to provide explanations behind the design and philosophy of Pyomo.
2. Provide preliminary documentation of new features and capabilities. We know that a new feature or capability probably will not be used unless it is documented. As Pyomo evolves, we plan to use this book to document these features. This provides users some context concerning the focus of Pyomo development, and it also provides an opportunity to get early feedback on new features before they are documented in other contexts.

Who Should Read This Book

This book is intended to be a reference for students, academic researchers and practitioners. Pyomo has been effectively used in the classroom with undergraduate and graduate students. However, we assume that the reader is generally familiar with optimization and mathematical modeling. Although this book does not contain a glossary, we recommend the Mathematical Programming Glossary [MPG] as a reference for the reader. We also assume that the reader is generally familiar with the Python programming language. There are a variety of books describing Python, as well as excellent documentation of the Python language and the software packages bundled with Python distributions.

Comments and Questions

The Pyomo home page provides resources for Pyomo users:

- http://pyomo.org

Pyomo development is hosted by Sandia National Laboratories and COIN-OR:

- https://software.sandia.gov/pyomo
- https://projects.coin-or.org/Pyomo

See the Pyomo Forum for online discussions of Pyomo:

- http://groups.google.com/group/pyomo-forum/

We welcome feedback on typos and errors in our examples, as well as comments on the presentation of this material.

Good Luck!

Chapter 1

Citing Pyomo

1.1 Pyomo

Hart, William E., Jean-Paul Watson, and David L. Woodruff. "Pyomo: modeling and solving mathematical programs in Python." Mathematical Programming Computation 3, no. 3 (2011): 219-260.

Hart, William E., Carl Laird, Jean-Paul Watson, and David L. Woodruff. Pyomo – Optimization Modeling in Python. Vol. 67. Springer, 2012.

1.2 PySP

Watson, Jean-Paul, David L. Woodruff, and William E. Hart. "PySP: modeling and solving stochastic programs in Python." Mathematical Programming Computation 4, no. 2 (2012): 109-149.

Chapter 2

Release Notes

2.1 Highlights

The following are highlights of the Pyomo 4.1 release:

- Modeling
  - API changes for model transformations
  - Revised API for SOSConstraint, Suffix and Block components
  - Optimization results are now loaded into models
  - Removed explicit specification of model preprocessing
- Solvers
  - Resolved many issues with writing and solving MPECs
  - Changes to MPEC meta-solver names
  - The solution output for runph has been changed to
- Other
  - Pyomo subcommands can now use configuration files (e.g. pyomo solve config.json)
  - New JSON/YAML format for parameter data
  - Added a script to install pyomo.extras

2.2 Determining Your Version of Pyomo

To determine your current version of Pyomo, use the command

```
pyomo --version
```

Chapter 3

Pyomo Overview

3.1 Mathematical Modeling

This chapter provides an introduction to Pyomo: Python Optimization Modeling Objects. A more complete description is contained in the Pyomo book. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with algebraic modeling languages (AMLs) such as AMPL [AMPL] AIMMS [AIMMS] and GAMS [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 mathematical 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. Hart et al. Pyomo, [PyomoJournal] 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.

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

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:

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

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 some contexts a mathematical model can be directly defined with the data values supplied at the time of the model definition and built into the model. 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.

3.4 A Simple Abstract Pyomo Model

We repeat the abstract model already given:

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

One way to implement this in Pyomo is as follows:

```
from __future__ import division
from pyomo.environ import *

model = AbstractModel()

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

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

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

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

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

model.OBJ = 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 = 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 used to ensure that `int` or `long` division arguments are converted to floating point values before division is performed.

```
from __future__ import division
```

In Python versions before 3.0, division returns the floor of the mathematical result of division if arguments are `int` or `long`. This import line avoids unexpected behavior when developing mathematical models with integer values.

The next import line that is required in every Pyomo model. Its purpose is to make the symbols used by Pyomo known to Python.

```
from pyomo.environ import *
```

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` most of the time in this book. In this example, we are declaring that it will be an abstract model.

```
model = AbstractModel()
```

We declare the parameters *m* and *n* using the Pyomo `Param` function. This function 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 = Param(within=NonNegativeIntegers)
model.n = Param(within=NonNegativeIntegers)
```

Although not required, it is convenient to define index sets. In this example we use the `RangeSet` function 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 = RangeSet(1, model.m)
model.J = 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` function, they indicate that the set will index the parameter.

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

Note

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

The next line interpreted by Python as part of the model declares the variable x. The first argument to the `Var` function 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 = Var(model.J, domain=NonNegativeReals)
```

In abstract models, Pyomo expressions are usually provided to objective function 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 function 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 `model.c, model.x` it returns an internal representation of the expression $\sum_{j=1}^{n} c_j x_j$.

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

To declare an objective function, the Pyomo function called `Objective` is used. The `rule` argument gives the name of a function that returns the expression to be used. The default *sense* is minimization. For maximization, the `sense=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 = Objective(rule=obj_expression)
```

Declaration of constraints is similar. A function is declared to deliver 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 $\sum_{j=1}^{n} a_{ij} x_j \geq b_i \ \forall i = 1 \ldots m$, 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(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]
```

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` function 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 function. 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 = 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.

```
# 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 ;
```

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.

```
# abstract2.py

from __future__ import division
from pyomo.environ import *

model = AbstractModel()

model.I = Set()
```

```
model.J = Set()

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

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

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

model.OBJ = 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 = 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
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 ;
```

3.5 A Simple Concrete Pyomo Model

It is possible to get nearly 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.

We repeat the concrete model already given:

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

This is implemented as a concrete model as follows:

```
from __future__ import division
from pyomo.environ import *

model = ConcreteModel()

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

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

model.Constraint1 = 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.

3.6 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
[    0.39] Applying Pyomo postprocessing actions
[    0.39] Pyomo Finished
```

The numbers is 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 4

Sets

4.1 Declaration

Sets can be declared using the `Set` and `RangeSet` functions or by assigning set expressions. The simplest set declaration creates a set and postpones creation of its members:

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

The `Set` function takes optional arguments such as:

- doc = String describing the set
- dimen = Dimension of the members of the set
- filter = A boolean function used during construction to indicate if a potential new member should be assigned to the set
- initialize = A function that returns the members to initialize the set. ordered = A boolean indicator that the set is ordered; the default is `False`
- validate = A boolean function that validates new member data
- virtual = A boolean indicator that the set will never have elements; it is unusual for a modeler to create a virtual set; they are typically used as domains for sets, parameters and variables
- within = Set used for validation; it is a super-set of the set being declared.

One way to create a set whose members will be two dimensional is to use the `dimen` argument:

```
model.B = 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 = 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 refer to a Python set, which can be returned by a function or given directly as in

```
model.D = Set(initialize=['red', 'green', 'blue'])
```

The `initialize` option can also specify a function that is applied sequentially to generate set members. Consider the case of a simple set. In this case, the initialization function accepts a set element number and model and returns the set element associated with that number:

```
def Z_init(model, i):
    if i > 10:
        return Set.End
    return 2*i+1
model.Z = Set(initialize=Z_init)
```

The `Set.End` return value terminates input to the set. Additional information about iterators for set initialization is in the Pyomo book.

Note
Data specified in an input file 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 = Set(model.A)
```

Arguments can be combined. For example, to create an array of sets with three dimensional members indexed by set `model.A`, use

```
model.F = Set(model.A, dimen=3)
```

The `initialize` option can be used to create a set that contains a sequence of numbers, but the `RangeSet` function provides a concise mechanism for simple sequences. This function 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 = RangeSet(1.5, 10, 3.5)
```

4.2 Operations

Sets may also be created by assigning other Pyomo sets as in these examples that also illustrate the set operators union, intersection, difference, and exclusive-or:

```
model.H = model.A
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
```

The cross-product operator is the asterisk (*). For example, to assign a set the cross product of two other sets, one could use

```
model.K = model.B * model.c
```

or to indicate the the members of a set are restricted to be in the cross product of two other sets, one could use

```
model.K = Set(within=model.B * model.C)
```

The cross-product operator is the asterisk (*). For example, to create a set that contains the cross-product of sets A and B, use

```
model.C = Set(model.A * model.B)
```

to instead create a set that can contain a subset of the members of this cross-product, use

```
model.C = Set(within=model.A * model.B)
```

4.3 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: same as boolean

For example, if the set `model.M` 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.M = Set(within=NegativeIntegers)
```

4.4 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 condusive 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 = 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:

$$\begin{array}{lll} \text{minimize} & \sum_{a\in\mathscr{A}} c_a x_a & \\ \text{subject to:} & S_n + \sum_{(i,n)\in\mathscr{A}} x_{(i,n)} & \\ & -D_n - \sum_{(n,j)\in\mathscr{A}} x_{(n,j)} & n \in \mathscr{N} \\ & x_a \geq 0, & a \in \mathscr{A} \end{array}$$

where

- Set: Nodes $\equiv \mathscr{N}$
- Set: Arcs $\equiv \mathscr{A} \subseteq \mathscr{N} \times \mathscr{N}$
- Var: Flow on arc (i, j): $\equiv x_{i,j},\ (i, j) \in \mathscr{A}$
- Param: Flow Cost on arc (i, j): $\equiv c_{i,j},\ (i, j) \in \mathscr{A}$
- Param: Demand at node i: $\equiv D_i,\ i \in \mathscr{N}$
- Param: Supply at node i: $\equiv S_i,\ i \in \mathscr{N}$

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 = Set(within=model.Nodes*model.Nodes)
```

or

```
model.Arcs = Set(dimen=2)
```

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(model, node):
    return model.Supply[node] \
     + sum(model.Flow[i, node] for i in model.Nodes if (i,node) in model.Arcs) \
     - model.Demand[node] \
     - sum(model.Flow[node, j] for j in model.Nodes if (j,node) in model.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(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
```

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(model, node):
    retval = []
    for (i,j) in model.Arcs:
        if j == node:
            retval.append(i)
    return retval
model.NodesIn = 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 :

```
# Isinglecomm.py
# NodesIn and NodesOut are intialized using the Arcs
from pyomo.environ import *

model = AbstractModel()

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

def NodesOut_init(model, node):
    retval = []
    for (i,j) in model.Arcs:
        if i == node:
            retval.append(j)
    return retval
model.NodesOut = Set(model.Nodes, initialize=NodesOut_init)

def NodesIn_init(model, node):
    retval = []
    for (i,j) in model.Arcs:
        if j == node:
            retval.append(i)
    return retval
model.NodesIn = Set(model.Nodes, initialize=NodesIn_init)

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

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

def Obj_rule(model):
    return summation(model.FlowCost, model.Flow)
model.Obj = Objective(rule=Obj_rule, sense=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 = Constraint(model.Nodes, rule=FlowBalance_rule)
```

for this model, a toy data file would be:

```
# Isinglecomm.dat: data for Isinglecomm.py

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 :=
CityA 0
CityB 1
CityC 1
;

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

This can be done somewhat more efficiently, and perhaps more clearly, using a build action as shown in Isinglebuild.py.

4.4.1 Sparse Index Sets Example

One may want to have a constraint that holds

```
for i in model.I, k in model.K, v in model.V[k]
```

There are many ways to accomplish this, but one good way is to create a set of tuples composed of all of `model.k, model.V[k]` pairs. This can be done as follows:

```
def kv_init(model):
    return ((k,v) for k in model.K for v in model.V[k])
model.KV=Set(dimen=2, initialize=kv_init)
```

So then if there was a constraint defining rule such as

```
def MyC_rule(model, i, k, v):
   return ...
```

Then a constraint could be declared using

```
model.MyConstraint = Constraint(model.I,model.KV,rule=c1Rule)
```

Here is the first few lines of a model that illustrates this:

```
from pyomo.environ import *

model = AbstractModel()

model.I=Set()
model.K=Set()
model.V=Set(model.K)

def kv_init(model):
    return ((k,v) for k in model.K for v in model.V[k])
model.KV=Set(dimen=2, initialize=kv_init)
```

```
model.a = Param(model.I, model.K)

model.y = Var(model.I)
model.x = Var(model.I, model.KV)

#include a constraint
#x[i,k,v] <= a[i,k]*y[i], for i in model.I, k in model.K, v in model.V[k]

def c1Rule(model,i,k,v):
   return model.x[i,k,v] <= model.a[i,k]*model.y[i]
model.c1 = Constraint(model.I,model.KV,rule=c1Rule)
```

Chapter 5

Parameters

The word "parameters" is used in many settings. When discussing a Pyomo model, we use the word to refer to data that must be provided in order to find an optimal (or good) assignment of values to the decision variables. Parameters are declared with the `Param` function, which takes arguments that are somewhat similar to the `Set` function. For example, the following code snippet declares sets `model.A`, `model.B` and then a parameter array `model.P` that is indexed by `model.A`:

```
model.A = Set()
model.B = Set()
model.P = Param(model.A, model.B)
```

In addition to sets that serve as indexes, the `Param` function takes the following command options:

- default = The value absent any other specification.
- doc = String describing the parameter
- initialize = A function (or Python object) that returns the members to initialize the parameter values.
- validate = A boolean function with arguments that are the prospective parameter value, the parameter indices and the model.
- within = Set used for validation; it specifies the domain of the parameter values.

These options perform in the same way as they do for `Set`. For example, suppose that `Model.A =RangeSet(1,3)`, then there are many ways to create a parameter that is a square matrix with 9, 16, 25 on the main diagonal zeros elsewhere, here are two ways to do it. First using a Python object to initialize:

```
v={}
v[1,1] = 9
v[2,2] = 16
v[3,3] = 25
model.S = Param(model.A, model.A, initialize=v, default=0)
```

And now using an initialization function that is automatically called once for each index tuple (remember that we are assuming that `model.A` contains 1,2,3)

```
def s_init(model, i, j):
    if i == j:
        return i*i
    else:
        return 0.0
model.S = Param(model.A, model.A, initialize=s_init)
```

In this example, the index set contained integers, but index sets need not be numeric. It is very common to use strings.

Note
Data specified in an input file will override the data specified by the initialize options.

Parameter values can be checked by a validation function. In the following example, the parameter S indexed by `model.A` and checked to be greater than 3.14159. If value is provided that is less than that, the model instantation would be terminated and an error message issued. The function used to validate should be written so as to return `True` if the data is valid and `False` otherwise.

```
def s_validate(model, v, i):
   return v > 3.14159
model.S = Param(model.A, validate=s_validate)
```

Chapter 6

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 = Var(within=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 = 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 = Var(model.A, domain=PositiveInteger, 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.

Chapter 7

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.g = 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 parameters and that `model.x` has been declared with the same index set, while `model.y` has been declared as a singleton:

```
def profrul(model):
   return summation(model.p, model.x) + model.y
model.Obj = Objective(rule=ObjRule, sense=maximize)
```

This example uses the `sense` option to specify maximization. The default sense is `minimize`.

Chapter 8

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 for them to be exactly three:

```
def teaOKrule(model):
    return(model.x['butter'] + model.x['scones'] == 3)
model.TeaConst = 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 = Var()

def aRule(model):
   return model.x >= 2
Boundx = Constraint(rule=aRule)

def bRule(model):
   return (2, model.x, None)
Boundx = Constraint(rule=bRule)
```

For this simple example, it would also be possible to declare `model.x` with a `bound` 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 i^{th} item to buy where the cost per item is given by the parameter `model.a`:

```
model.A = RangeSet(1,10)
model.a = Param(model.A, within=PostiveReals)
model.ToBuy = Var(model.A)
def bud_rule(model, i):
    return model.a[i]*model.ToBuy[i] <= i
aBudget = Constraint(model.A, rule=bud_rule)
```

Note
Python and Pyomo are case sensitive so `model.a` is not the same as `model.A`.

Chapter 9

Disjunctions

This is an advanced topic.

A disjunction is a set of collections of variables, parameters, and constraints that are linked by an OR (really exclusive or) constraint. The simplest case is a 2-term disjunction:

D1 V D2

That is, either the constraints in the collection D1 are enforced, OR the constraints in the collection D2 are enforced.

In pyomo, we model each collection using a special type of block called a `Disjunct`. Each `Disjunct` is a block that contains an implicitly declared binary variable, "indicator_var" that is 1 when the constraints in that `Disjunct` is enforced and 0 otherwise.

9.1 Declaration

The following condensed code snippet illustrates a `Disjunct` and a `Disjunction`:

```
# Two conditions
def _d(disjunct, flag):
    model = disjunct.model()
    if flag:
        # x == 0
        disjunct.c = Constraint(expr=model.x == 0)
    else:
        # y == 0
        disjunct.c = 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)
```

Model.d is an indexed `Disjunct` that is indexed over an implicit set with members 0 and 1. Since it is an indexed thing, each member is initialized using a call to a rule, passing in the index value (just like any other pyomo component). However, just defining disjuncts is not sufficient to define disjunctions, as pyomo has no way of knowing which disjuncts should be bundled into which disjunctions. To define a disjunction, you use a `Disjunction` component. The disjunction takes either a rule or an expression that returns a list of disjuncts over which it should form the disjunction. This is what `_c` function in the example returns.

Note

There is no requirement that disjuncts be indexed and also no requirement that they be defined using a shared rule. It was done in this case to create a condensed example.

9.2 Transformation

In order to use the solvers currently avaialbe, one must convert the disjunctive model to a standard MIP/MINLP model. The easiest way to do that is using the (included) BigM or Convex Hull transformations. From the Pyomo command line, include the option `--transform pyomo.gdp.bigm` or `--transform pyomo.gdp.chull`

9.3 Notes

Some notes:

- all variables that appear in disjuncts need upper and lower bounds
- for linear models, the BigM transform can estimate reasonably tight M values for you
- for all other models, you will need to provide the M values through a "BigM" Suffix.
- the convex hull reformulation is only valid for linear and convex nonlinear problems. Nonconvex problems are not supported (and are not checked for).

When you declare a Disjunct, it (at declaration time) will automatically have a variable "indicator_var" defined and attached to it. After that, it is just a Var like any other Var.

Chapter 10

Expressions

In this chapter, we use the word “expression” is two ways: first in the general sense of the word and second to desribe a class of Pyomo objects that have the name `expression` as described in the subsection on expression objects.

10.1 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 expression 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 Pyton data in the form of a Python variable called `switch`:

```
switch = 3

model.A = RangeSet(1, 10)
model.c = Param(model.A)
model.d = Param()
model.x = Var(model.A, domain=Boolean)

def pi_rule(model)
    accexpr = summation(model.c, model.x)
    if switch >= 2:
        accexpr = accexpr - model.d
    return accexpr >= 0.5
PieSlice = 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.

Caution
Because model elements result in expressions, not values, the following does not work as expected in an abstract model!

```
model.A = RangeSet(1, 10)
model.c = Param(model.A)
model.d = Param()
model.x = Var(model.A, domain=Boolean)
```

```
def pi_rule(model)
    accexpr = summation(model.c, model.x)
    if model.d >= 2:  # NOT in an abstract model!!
        accexpr = accexpr - model.d
    return accexpr >= 0.5
PieSlice = 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`

10.2 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 [Vielma_et_al].

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 (keys are 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 full 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: return exp(x) + value(model.p)
                        (model.p is a Pyomo Param)

                    # 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 piecwise 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':maximize,'force_pw':True}
```

Here is a simple example based on the abstract2.py example given early. 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.

```
# abstract2piece.py
# Similar to abstract2.py, but the objective is now c times x to the fourth power

from pyomo.environ import *

model = AbstractModel()

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

Topx = 6.1 # range of x variables

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

# the next line declares a variable indexed by the set J
```

```
model.x = Var(model.J, domain=NonNegativeReals, bounds=(0, Topx))
model.y = Var(model.J, domain=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

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

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

model.OBJ = 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 = Constraint(model.I, rule=ax_constraint_rule)
```

A more advanced example is provided as abstract2piecebuild.py.

10.3 `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 = AbstractModel()
model.x = Var(initialize=1.0)
def _e(m,i):
    return m.x*i
model.e = Expression([1,2,3],initialize=_e)

instance = model.create_instance()

print 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

# print the value of the expression given the loaded optimal solution
print 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 = ConcreteModel()
model.x = 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.

Chapter 11

Data Input

Pyomo can initialize models in two general ways. When executing the `pyomo` command, one or more data command files can be specified to declare data and load data from other data sources (e.g. spreadsheets and CSV files). When initializing a model within a Python script, a `DataPortal` object can be used to load data from one or more data sources.

11.1 Data Command Files

The following commands can be used in data command files:

- `set` declares set data,
- `param` declares a table of parameter data, which can also include the declaration of the set data used to index parameter data,
- `load` loads set and parameter data from an external data source such as ASCII table files, CSV files, ranges in spreadsheets, and database tables,
- `table` loads set and parameter data from a table,
- `include` specifies a data command file that is to be processed immediately,
- the `data` and `end` commands do not perform any actions, but they provide compatibility with AMPL scripts that define data commands, and
- `namespace` defines groupings of data commands.

The syntax of the `set` and `param` data commands are adapted from AMPL's data commands. However, other Pyomo data commands do not directly correspond to AMPL data commands. In particular, Pyomo's `table` command was introduced to work around semantic ambiguities in the `param` command. Pyomo's `table` command does not correspond to AMPL's `table` command. Instead, the `load` command mimics AMPL's `table` command with a simplified syntax.

Warning
The data command file was initially developed to provide compatability in data formats between Pyomo and AMPL. However, these data formats continue to diverge in their syntax and semantics. Simple examples using `set` and `param` data commands are likely to work for both AMPL and Pyomo, particularly with abstract Pyomo models. But in general a user should expect to need to adapt their AMPL data command files for use with Pyomo.

See the Pyomo book for detailed descriptions of these commands. The following sections provide additional details, particularly for new data commands that are not described in the Pyomo book: `table`.

11.1.1 table

The `table` data command was developed to provide a more flexible and complete data declaration than is possible with the `param` declaration. This command has a similar syntax to the `load` command, but it includes a complete specification of the table data.

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`. The column labels are provided after the colon and before the `:=`. Subsequently, the table data is provided. Note that the syntax is not sensitive to whitespace. Thus, 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
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

This example declares data for the `M` and `N` parameters. As this example illustrates, these parameters may 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 examples 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
;
```

Finally, singleton parameter values can be specified with a simple `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 file table0.dat 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 unlabeled tables. 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 A={1} Z={1,2} M(1) N(1,2) :=
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 A={1} Z={1,2} M(A) N(Z) :=
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.
- 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.
- Both commands can be used to declare a singleton parameter.
- 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 a `DataPortal` object. 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). See the discussion of `DataPortal` objects below for an example.

11.1.2 namespace

The `namespace` command allows data commands to be organized into named groups that can be enabled from the `pyomo` command line. For example, consider again the abstract2.py example. Suppose that the cost data shown in abstract2.dat were valid only under certain circumstances that we will label as "TerryG" and that there would be different cost data under circumstances that we will label "JohnD." This could be represented using the following data file:

```
# abs2nspace.dat AMPL format with namespaces

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

param a :=
TV  Graham 3
TV John 4.4
TV Carol 4.9
Film Graham 1
Film John 2.4
Film Carol 1.1
;

namespace TerryG {
   param c := [*]
     Graham 2.2
     John 3.1416
     Carol 3
   ;
}

namespace JohnD {
   param c := [*]
     Graham 2.7
     John 3
     Carol 2.1
   ;
}

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

To use this data file with abstract2.py, a namespace must be indicated on the command line. To select the "TerryG" data specification, `--namespace TerryG` would be added to the command line. For example:

```
pyomo solve abstract2.py abs2nspace.dat --namespace TerryG --solver=cplex
```

If the `--namespace` option is omitted, then no data will be given for `model.c` (and no default was given for `model.c`). In other words, there is no default namespace selection.

The option `-ns` (with one dash) is an alias for `--namespace` (which needs two dashes) Multiple namespaces can be selected by giving multiple `--namespace` or `-ns` arguments on the Pyomo command line.

11.2 DataPortal Objects

The `load` and `store` Pyomo data commands can be used to load set and table data from a variety of data sources. Pyomo's `DataPortal` object provides this same functionality for users who work with Python scripts. A `DataPortal` object manages the process of loading data from different data sources, and it is used to construct model instances in a standard manner. Similarly, a `DataPortal` object can be used to store model data externally in a standard manner.

11.2.1 Loading Data

The `load` method can be used to load data into Pyomo models from a variety of sources and formats. The most common format is a table representation of set and parameter data. For example, consider the file `A.tab`, which defines a simple set:

```
A
A1
A2
```

```
A3
```

The following example illustrates how a `DataPortal` object can be used to load this data into a model:

```
model = AbstractModel()
model.A = Set()

data = DataPortal()
data.load(filename='tab/A.tab', set=model.A)
instance = model.create(data)
```

The `load` method opens the data file, processes it, and loads the data in a format that is then 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.

Note
Subsequent examples omit the model declaration and instance creation.

In the previous example, the `set` option is used to define the model component that is loaded with the set data. If the data source defines a table of data, then this option is used to specify data for a multi-dimensional set. For example, 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
```

If a two-dimensional set is declared, then it can be loaded with the same syntax:

```
model.A = Set(dimen=2)

data.load(filename='tab/C.tab', set=model.A)
```

This example also illustrates that the column titles do not directly impact the process of loading data. The set `model.A` is declared to have two dimensions, so the first element of the set will be `(A1, 1)` and so on. Column titles are only used to select columns that are included in the table that is loaded (see the `select` option below.)

The `param` option is used to define the a parameter component that is loaded with data. The simplest parameter is a singleton. For example, consider the file `Z.tab` that contains just one number:

```
1.01
```

This data is loaded with the following syntax:

```
model.z = Param()

data.load(filename='tab/Z.tab', param=model.z)
```

Indexed parameters can be defined from table data. For example, consider the file `Y.tab`:

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

The parameter `y` is loaded with the following syntax:

```
model.A = Set(initialize=['A1','A2','A3','A4'])
model.y = Param(model.A, default=0.0)

data.load(filename='tab/Y.tab', param=model.y)
```

Pyomo assumes that the parameter values are defined on the rightmost column; the column names are not used to specify the index and parameter data (see the `select` option below). In this file, the `A` column contains the index values, and the `Y` column contains the parameter values. Note that no value is given in the file for index A4, so for this index the default value will be used (unless a previous `load` put in a different value).

Multiple parameters can be initialized at once by specifying a list (or tuple) of component parameters. For example, consider the file `XW.tab`:

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

The parameters `x` and `w` are loaded with the following syntax:

```
model.A = Set(initialize=['A1','A2','A3','A4'])
model.x = Param(model.A)
model.w = Param(model.A)

data.load(filename='tab/XW.tab', param=(model.x,model.w))
```

Note that the data for set `A` is predefined in this example. The index set can be loaded along with the parameter data using the `index` option as shown in the next example:

```
model.A = Set()
model.x = Param(model.A)
model.w = Param(model.A)

data.load(filename='tab/XW.tab', param=(model.x,model.w), index=model.A)
```

We have previously noted that the column names are not used to define the set and parameter data in the examples given so far. The `select` option can be used to define the columns in the table that are used to load 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.A = Set()
model.w = Param(model.A)

data.load(filename='tab/XW.tab', select=('A','W'), param=model.w, index=model.A)
```

The columns `A` and `W` are selected from the file `XW.tab`, and a the data for a single parameter is loaded.

Note

The `load` method allows for a variety of other options that are supported by the `add` method for `ModelData` objects. See the Pyomo book for a detailed description of these options.

Chapter 12

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 peform the action. For example,

```
model.BuildBpts = 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 abstract2.py and abstract2piece.py examples, is

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

from pyomo.environ import *

model = AbstractModel()

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

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

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

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

```
# the only goal is to trigger the action at build time
model.BuildBpts = 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 = Piecewise(model.J, model.y, model.x, pw_pts=model.bpts,  ↩
    pw_constr_type='EQ', f_rule=f4)

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

model.OBJ = 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 = Constraint(model.I, rule=ax_constraint_rule)
```

This example uses the build action to create a model component with breakpoints for a piecewise 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 functionality in Pyomo, we use a `BuildAction`.

Another application of `BuildAction` can be intialization of Pyomo model data from Python data structures, or efficient initialization of Pyomo model data from other Pyomo model data. Consider the Isinglecomm.py 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 :

```
# Isinglecomm.py
# NodesIn and NodesOut are created by a build action using the Arcs
from pyomo.environ import *

model = AbstractModel()

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

model.NodesOut = Set(model.Nodes, within=model.Nodes, initialize=[])
model.NodesIn = 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 = BuildAction(rule = Populate_In_and_Out)

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

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

def Obj_rule(model):
```

```
    return summation(model.FlowCost, model.Flow)
model.Obj = Objective(rule=Obj_rule, sense=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 = Constraint(model.Nodes, rule=FlowBalance_rule)
```

for this model, the same data file can be used as for Isinglecomm.py such as the toy data file:

```
# Isinglecomm.dat: data for Isinglecomm.py

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 :=
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 prefered 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`.

Chapter 13

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.

13.1 Passing Options to a Solver

To pass arguments to a solver when using the pyomo solve command, appned 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 Solver Options for a discusion of passing options in a script.

13.2 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 Solver Display 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=",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 = 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.

13.3 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 `nl`
- os: generate an OSiL format XML file.
- python: use the direct Python interface.

Note that not all solvers support all interfaces.

Chapter 14

PySP Overview

This chapter describes PySP: (Pyomo Stochastic Programming), where parameters are allowed to be uncertain.

14.1 Overview of Modeling Components and Processes

The sequence of activities is typically the following:

- Create a deterministic model and declare components
- Develop base-case data for the deterministic model
- Test, verify and validate the deterministic model
- Model the stochastic processes
- Develop a way to generate scenarios (in the form of a tree if there are more than two stages)
- Create the data files need to describe the stochastics
- Use PySP to solve stochastic problem

When viewed from the standpoint of file creation, the process is

- Create an abstract model for the deterministic problem in a file called `ReferenceModel.py`
- Specify the stochastics in a file called `ScenarioStructure.dat`
- Specify scenario data

14.2 Birge and Louveaux's Farmer Problem

Birge and Louveaux [BirgeLouveauxBook] make use of the example of a farmer who has 500 acres that can be planted in wheat, corn or sugar beets, at a per acre cost of 150, 230 and 260 (Euros, presumably), respectively. The farmer needs to have at least 200 tons of wheat and 240 tons of corn to use as feed, but if enough is not grown, those crops can be purchased for 238 and 210, respectively. Corn and wheat grown in excess of the feed requirements can be sold for 170 and 150, respectively. A price of 36 per ton is guaranteed for the first 6000 tons grown by any farmer, but beets in excess of that are sold for 10 per ton. The yield is 2.5, 3, and 20 tons per acre for wheat, corn and sugar beets, respectively.

14.2.1 ReferenceModel.py

So far, this is a deterministic problem because we are assuming that we know all the data. The Pyomo model for this problem shown here is in the file `ReferenceModel.py` in the sub-directory `examples/pysp/farmer/models` that is distributed with Pyomo.

```
# Farmer: rent out version has a scalar root node var
# note: this will minimize
#
# Imports
#

from __future__ import division
from pyomo.environ import *

#
# Model
#

model = AbstractModel()

#
# Parameters
#

model.CROPS = Set()

model.TOTAL_ACREAGE = Param(within=PositiveReals)

model.PriceQuota = Param(model.CROPS, within=PositiveReals)

model.SubQuotaSellingPrice = Param(model.CROPS, within=PositiveReals)

def super_quota_selling_price_validate (model, value, i):
    return model.SubQuotaSellingPrice[i] >= model.SuperQuotaSellingPrice[i]

model.SuperQuotaSellingPrice = Param(model.CROPS, validate= ↩
    super_quota_selling_price_validate)

model.CattleFeedRequirement = Param(model.CROPS, within=NonNegativeReals)

model.PurchasePrice = Param(model.CROPS, within=PositiveReals)

model.PlantingCostPerAcre = Param(model.CROPS, within=PositiveReals)

model.Yield = Param(model.CROPS, within=NonNegativeReals)

#
# Variables
#

model.DevotedAcreage = Var(model.CROPS, bounds=(0.0, model.TOTAL_ACREAGE))

model.QuantitySubQuotaSold = Var(model.CROPS, bounds=(0.0, None))
model.QuantitySuperQuotaSold = Var(model.CROPS, bounds=(0.0, None))

model.QuantityPurchased = Var(model.CROPS, bounds=(0.0, None))

model.FirstStageCost = Var()
model.SecondStageCost = Var()

#
```

```
# Constraints
#

def ConstrainTotalAcreage_rule(model):
    return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = Constraint(rule=ConstrainTotalAcreage_rule)

def EnforceCattleFeedRequirement_rule(model, i):
    return model.CattleFeedRequirement[i] <= (model.Yield[i] * model.DevotedAcreage[i]) +  ↩
        model.QuantityPurchased[i] - model.QuantitySubQuotaSold[i] - model. ↩
        QuantitySuperQuotaSold[i]

model.EnforceCattleFeedRequirement = Constraint(model.CROPS, rule= ↩
    EnforceCattleFeedRequirement_rule)

def LimitAmountSold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + model.QuantitySuperQuotaSold[i] - (model.Yield[i ↩
        ] * model.DevotedAcreage[i]) <= 0.0

model.LimitAmountSold = Constraint(model.CROPS, rule=LimitAmountSold_rule)

def EnforceQuotas_rule(model, i):
    return (0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])

model.EnforceQuotas = Constraint(model.CROPS, rule=EnforceQuotas_rule)

#
# Stage-specific cost computations
#

def ComputeFirstStageCost_rule(model):
    return model.FirstStageCost - summation(model.PlantingCostPerAcre, model.DevotedAcreage ↩
        ) == 0.0

model.ComputeFirstStageCost = Constraint(rule=ComputeFirstStageCost_rule)

def ComputeSecondStageCost_rule(model):
    expr = summation(model.PurchasePrice, model.QuantityPurchased)
    expr -= summation(model.SubQuotaSellingPrice, model.QuantitySubQuotaSold)
    expr -= summation(model.SuperQuotaSellingPrice, model.QuantitySuperQuotaSold)
    return (model.SecondStageCost - expr) == 0.0

model.ComputeSecondStageCost = Constraint(rule=ComputeSecondStageCost_rule)

#
# Objective
#

def Total_Cost_Objective_rule(model):
    return model.FirstStageCost + model.SecondStageCost

model.Total_Cost_Objective = Objective(sense=minimize, rule=Total_Cost_Objective_rule)
```

14.2.2 Example Data

The data introduced here are in the file ReferenceModel.dat in the sub-directory examples/pysp/farmer/scenariodata that is distributed with Pyomo. These data are given for illustration. The file ReferenceModel.dat is not required by PySP.

```
set CROPS := WHEAT CORN SUGAR_BEETS ;
```

```
param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota :=
            WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice :=
            WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice :=
            WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement :=
            WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no need, as cattle don't eat them)
param PurchasePrice :=
            WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre :=
            WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 3.0 CORN 3.6 SUGAR_BEETS 24 ;
```

Any of these data could be modeled as uncertain, but we will consider only the possibility that the yield per acre could be higher or lower than expected. Assume that there is a probability of 1/3 that the yields will be the average values that were given (i.e., wheat 2.5; corn 3; and beets 20). Assume that there is a 1/3 probability that they will be lower (2, 2.4, 16) and 1/3 probability they will be higher (3, 3.6, 24). We refer to each full set of data as a *scenario* and collectively we call them a *scenario tree*. In this case the scenario tree is very simple: there is a root node and three leaf nodes: one corresponding to each scenario. The acreage-to-plant decisions are root node decisions because they must be made without knowing what the yield will be. The other variables are so-called *second stage* decisions, because they will depend on which scenario is realized.

14.2.3 ScenarioStructure.dat

PySP requires that users describe the scenario tree using specific constructs in a file named `ScenarioStructure.dat`; for the farmer problem, this file can be found in the pyomo sub-directory `examples/pysp/farmer/scenariodata` that is distributed with Pyomo.

```
# IMPORTANT - THE STAGES ARE ASSUMED TO BE IN TIME-ORDER.

set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;

param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;

param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
```

```
                                AboveAverageNode  0.33333333 ;

set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;

param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;

set StageVariables[FirstStage] :=  DevotedAcreage[*] ;
set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*] ;

param StageCost := FirstStage  FirstStageCost
                   SecondStage SecondStageCost ;
```

This data file is verbose and somewhat redundant, but in most applications it is generated by software rather than by a person, so this is not an issue. Generally, the left-most part of each expression (e.g. “set Stages :=”) is required and uses reserved words (e.g., `Stages`) and the other names are supplied by the user (e.g., “FirstStage” could be any name). Every assignment is terminated with a semi-colon. We will now consider the assignments in this file one at a time.

The first assignments provides names for the stages and the words "set Stages" are required, as are the := symbols. Any names can be used. In this example, we used "FirstStage" and "SecondStage" but we could have used "EtapPrimero" and "ZweiteEtage" if we had wanted to. Whatever names are given here will continue to be used to refer to the stages in the rest of the file. The order of the names is important. A simple way to think of it is that generally, the names must be in time order (technically, they need to be in order of information discovery, but that is usually time-order). Stages refers to decision stages, which may, or may not, correspond directly with time stages. In the farmer example, decisions about how much to plant are made in the first stage and "decisions" (which are pretty obvious, but which are decision variables nonetheless) about how much to sell at each price and how much needs to be bought are second stage decisions because they are made after the yield is known.

```
set Stages := FirstStage SecondStage ;
```

Node names are constructed next. The words "set Nodes" are required, but any names may be assigned to the nodes. In two stage stochastic problems there is a root node, which we chose to name "RootNode" and then there is a node for each scenario.

```
set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;
```

Nodes are associated with time stages with an assignment beginning with the required words "param Nodestage." The assignments must make use of previously defined node and stage names. Every node must be assigned a stage.

```
param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;
```

The structure of the scenario tree is defined using assignment of children to each node that has them. Since this is a two stage problem, only the root node has children. The words "param Children" are required for every node that has children and the name of the node is in square brackets before the colon-equals assignment symbols. A list of children is assigned.

```
set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;
```

The probability for each node, conditional on observing the parent node is given in an assignment that begins with the required words "param ConditionalProbability." The root node always has a conditional probability of 1, but it must always be given anyway. In this example, the second stage nodes are equally likely.

```
param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
                                AboveAverageNode  0.33333333 ;
```

Scenario names are given in an assignment that begins with the required words "set Scenarios" and provides a list of the names of the scenarios. Any names may be given. In many applications they are given unimaginative names generated by software such as "Scen1" and the like. In this example, there are three scenarios and the names reflect the relative values of the yields.

```
set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;
```

Leaf nodes, which are nodes with no children, are associated with scenarios. This assignment must be one-to-one and it is initiated with the words "param ScenarioLeafNode" followed by the colon-equals assignment characters.

```
param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;
```

Variables are associated with stages using an assignment that begins with the required words "set StageVariables" and the name of a stage in square brackets followed by the colon-equals assignment characters. Variable names that have been defined in the file ReferenceModel.py can be assigned to stages. Any variables that are not assigned are assumed to be in the last stage. Variable indexes can be given explicitly and/or wildcards can be used. Note that the variable names appear without the prefix "model." In the farmer example, DevotedAcreage is the only first stage variable.

```
set StageVariables[FirstStage] :=  DevotedAcreage[*] ;
set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*] ;
```

Note
Variable names appear without the prefix "model."

Note
Wildcards can be used, but fully general Python slicing is not supported.

For reporting purposes, it is useful to define auxiliary variables in `ReferenceModel.py` that will be assigned the cost associated with each stage. This variables do not impact algorithms, but the values are output by some software during execution as well as upon completion. The names of the variables are assigned to stages using the "param StageCost" assignment. The stages are previously defined in `ScenarioStructure.dat` and the variables are previously defined in `ReferenceModel.py`.

```
param StageCost := FirstStage  FirstStageCost
                   SecondStage SecondStageCost ;
```

14.2.4 Scenario data specification

So far, we have given a model in the file named `ReferenceModel.py`, a set of deterministic data in the file named `ReferenceModel.py`, and a description of the stochastics in the file named `ScenarioStructure.dat`. All that remains is to give the data for each scenario. There are two ways to do that in PySP: *scenario-based* and *node-based*. The default is scenario-based so we will describe that first.

For scenario-based data, the full data for each scenario is given in a `.dat` file with the root name that is the name of the scenario. So, for example, the file named `AverageScenario.dat` must contain all the data for the model for the scenario named "AvererageScenario." It turns out that this file can be created by simply copying the file `ReferenceModel.dat` as shown above because it contains a full set of data for the "AverageScenario" scenario. The files `BelowAverageScenario.dat` and `AboveAverageScenario.dat` will differ from this file and from each other only in their last line, where the yield is specified. These three files are distributed with Pyomo and are in the pyomo sub-directory `examples/pysp/farmer/scenariodata` along with `ScenarioStructure.dat` and `ReferenceModel.dat`.

Scenario-based data wastes resources by specifying the same thing over and over again. In many cases, that does not matter and it is convenient to have full scenario data files available (for one thing, the scenarios can easily be run independently using the `pyomo` command). However, in many other settings, it is better to use a node-based specification where the data that is unique to each node is specified in a .dat file with a root name that matches the node name. In the farmer example, the file `RootNode.dat` will be the same as `ReferenceModel.dat` except that it will lack the last line that specifies the yield. The files `BelowAverageNode.dat`, `AverageNode.dat`, and `AboveAverageNode.dat` will contain only one line each to specify the yield. If node-based data is to be used, then the `ScenarioStructure.dat` file must contain the following line:

```
param ScenarioBasedData := False ;
```

An entire set of files for node-based data for the farmer problem are distributed with Pyomo in the sub-directory `examples/pysp/farmer/nodedata`

14.3 Finding Solutions for Stochastic Models

PySP provides a variety of tools for finding solutions to stochastic programs.

14.3.1 runef

The `runef` command puts together the so-called *extensive form* version of the model. It creates a large model that has constraints to ensure that variables at a node have the same value. For example, in the farmer problem, all of the `DevotedAcres` variables must have the same value regardless of which scenario is ultimately realized. The objective can be the expected value of the objective function, or the CVaR, or a weighted combination of the two. Expected value is the default. A full set of options for `runef` can be obtained using the command:

```
runef --help
```

The pyomo distribution contains the files need to run the farmer example in the sub-directories to the sub-directory `examples/pysp/farmer` so if this is the current directory and if CPLEX is installed, the following command will cause formation of the EF and its solution using CPLEX.

```
runef -m models -i nodedata --solver=cplex --solve
```

The option `-m models` has one dash and is short-hand for the option `--model-directory=models` and note that the full option uses two dashes. The `-i` is equivalent to `--instance-directory=` in the same fashion. The default solver is CPLEX, so the solver option is not really needed. With the `--solve` option, runef would simply write an .lp data file that could be passed to a solver.

14.3.2 runph

The runph command executes an implementation of Progressive Hedging (PH) that is intended to support scripting and extension.

The pyomo distribution contains the files need to run the farmer example in the sub-directories to the sub-directory examples/pysp/farmer so if this is the current directory and if CPLEX is installed, the following command will cause PH to execute using the default sub-problem solver, which is CPLEX.

```
runph -m models -i nodedata
```

The option `-m models` has one dash and is short-hand for the option `--model-directory=models` and note that the full option uses two dashes. The `-i` is equivalent to `--instance-directory=` in the same fashion.

After about 33 iterations, the algorithm will achieve the default level of convergence and terminate. A lot of output is generated and among the output is the following solution information:

```
Variable=DevotedAcreage
        Index: [CORN]            (Scenarios:  BelowAverageScenario   AverageScenario    ↩
            AboveAverageScenario   )
                Values:       79.9844      80.0000      79.9768     Max-Min=      0.0232  ↩
                        Avg=     79.9871
        Index: [SUGAR_BEETS]             (Scenarios:  BelowAverageScenario    ↩
            AverageScenario   AboveAverageScenario   )
                Values:      249.9848     249.9770     250.0000     Max-Min=      0.0230  ↩
                        Avg=    249.9873
        Index: [WHEAT]           (Scenarios:  BelowAverageScenario   AverageScenario    ↩
            AboveAverageScenario   )
                Values:      170.0308     170.0230     170.0232     Max-Min=      0.0078  ↩
                        Avg=    170.0256
Cost Variable=FirstStageCost
        Tree Node=RootNode               (Scenarios:  BelowAverageScenario    ↩
            AverageScenario   AboveAverageScenario   )
        Values:   108897.0836  108897.4725  108898.1476     Max-Min=      1.0640     Avg=  ↩
            108897.5679
```

For problems with no, or few, integer variables, the default level of convergence leaves root-node variables almost converged. Since the acreage to be planted cannot depend on the scenario that will be realized in the future, the average, which is labeled "Avg" in this output, would be used. A farmer would probably interpret acreages of 79.9871, 249.9873, and 170.0256 to be 80, 250, and 170. In real-world applications, PH is embedded in scripts that produce output in a format desired by a decision maker.

But in real-world applications, the default settings for PH seldom work well enough. In addition to post-processing the output, a number of parameters need to be adjusted and sometimes scripting to extend or augment the algorithm is needed to improve convergence rates. A full set of options can be obtained with the command:

```
runph --help
```

Note that there are two dashes before `help`.

By default, PH uses quadratic objective functions after iteration zero; in some settings it may be desirable to linearize the quadratic terms. This is required to use a solver such as glpk for MIPs because it does not support quadratic MIPs. The directive `--linearize-nonbinary-penalty-terms=n` causes linearization of the penalty terms using n pieces. For example, to use glpk on the farmer, assuming glpk is installed and the command is given when the current directory is the `examples/pysp/farmer`, the following command will use default settings for most parameters and four pieces to approximate quadratic terms in sub-problems:

```
runph -i nodedata -m models --solver=glpk --linearize-nonbinary-penalty-terms=4
```

Use of the `linearize-nonbinary-penalty-terms` option requires that all variables not in the final stage have bounds.

14.3.3 Final Solution

At each iteration, PH computes an average for each variable over the nodes of the scenario tree. We refer to this as X-bar. For many problems, particularly those with integer restrictions, X-bar might not be feasible for every scenario unless PH happens to be fully converged (in the primal variables). Consequently, the software computes a solution system X-hat that is more likely to be feasible for every scenario and will be equivalent to X-bar under full convergence. This solution is reported upon completion of PH and its expected value is report if it is feasible for all scenarios.

Methods for computing X-hat are controlled by the `--xhat-method` command-line option. For example

```
--xhat-method=closest-scenario
```

causes X-hat to be set to the scenario that is closest to X-bar (in a z-score sense). Other options, such as `voting` and `rounding`, assign values of X-bar to X-hat except for binary and general integer variables, where the values are set by probability weighted voting by scenarios and rounding from X-bar, respectively.

14.3.4 Solution Output Control

To get the full solution, including leaf node solution values, use the `runph --output-scenario-tree-solution` option.

In both `runph` and `runef` the solution can be written in csv format using the `--solution-writer=pyomo.pysp.plugins.csvsolutionwriter` option.

14.4 Summary of PySP File Names

PySP scripts such as `runef` and `runph` require files that specify the model and data using files with specific names. All files can be in the current directory, but typically, the file `ReferenceModel.py` is in a directory that is specified using `--model-directory=` option (the short version of this option is `-i +) and the data files are in a directory specified in the +--instance-directory=` option (the short version of this option is +-m +).

Note
A file name other than `ReferenceModel.py` can be used if the file name is given in addition to the directory name as an argument to the `--instance-directory` option. For example, on a Windows machine `--instance-directory=models\MyModel.py` would specify the file `MyModel.py` in the local directory `models`.

- `ReferenceModel.py`: A full Pyomo model for a singe scenario. There should be no scenario indexes in this model because they are implicit.
- `ScenarioStructure.dat`: Specifies the nature of the stochastics. It also specifies whether the rest of the data is node-based or scenario-based. It is scenario-based unless `ScenarioStructure.dat` contains the line

```
param ScenarioBasedData := False ;
```

If scenario-based, then there is a data file for each scenario that specifies a full set of data for the scenario. The name of the file is the name of the scenario with `.dat` appended. The names of the scenarios are given in the `ScenarioStructure.dat` file.

If node-based, then there is a file with data for each node that specifies only that data that is unique for the node. The name of the file is the name of the node with `.dat` appended. The names of the nodes are given in the `ScenarioStructure.dat` file.

14.5 Solving Sub-problems in Parallel and/or Remotely

The Python package called Pyro provides capabilities that are used to enable PH to make use of multiple solver processes for sub-problems and allows both `runef` and `runph` to make use remote solvers. We will focus on PH in our discussion here.

There are two solver management systems available for `runph`, one is based on a `pyro_mip_server` and the other is based on a `phsolverserver`. Regardless of which is used, a name server and a dispatch server must be running and accessible to the `runph` process. The name server is launched using the command `pyomo_ns` and then the dispatch server is launched with `dispatch_srvr`. Note that both commands contain an underscore. Both programs keep running until terminated by an external signal, so it is common to pipe their output to a file.

Solvers are controlled by solver servers. The pyro mip solver server is launched with the command `pyro_mip_server`. This command may be repeated to launch as many solvers as are desired. The `runph` then needs a `--solver-manager=pyro` option to signal that `runph` should not launch its own solver, but should send subproblems to be dispatched to parallel solvers. To summarize the commands:

- Once: `pyomo_ns`
- Once: `dispatch_srvr`
- Multiple times: `pyro_mip_server`
- Once: `runph ... --solver-manager=pyro ...`

Note
The `runph` option `--shutdown-pyro` will cause a shutdown signal to be sent to `pyomo_ns`, `dispatch_srvr` and all `pyro_mip_server` programs upon termination of `runph`.

Instead of using `pyro_mip_server`, one can use `phsolverserver` in its place. You can get a list of arguments using `pyrosolverserver --help`, which does not launch a solver server (it just displays help and terminates). If you use the phsolverserver, then use `--solver-manager=phpyro` as an argument to runph rather than `--solver-manager=pyro`.

Warning
Unlike the normal `pyro_mip_server`, there must be one `phsolverserver` for each sub-problem. One can use fewer phsolverservers than there are scenarios by adding the command-line option "--phpyro-required-workers=X". This will partition the jobs among the available workers,

---

14.6 Generating SMPS Input Files From PySP Models

This document explains how to convert a PySP model to the SMPS file format for stochastic linear programs. Converting a PySP model to a set of SMPS files is performed by the `pysp2smps` command. This command gets installed with Pyomo starting at version 4.2.

SMPS is a standard for expressing stochastic mathematical programs that is based on the ancient MPS format for linear programs, which is matrix-based. Modern algebraic modeling languages such as Pyomo offer a lot of flexibility so it is a challenge to take models expressed in Pyomo/PySP and force them into SMPS format. The conversions can be inefficient and error prone because Pyomo allows flexible expressions and model construction so the resulting matrix may not be the same for each set of input data. We provide tools for conversion to SMPS because some researchers have tools that read SMPS and exploit its limitations on problem structure; however, the user should be aware that the conversion is not always possible.

Currently, these routines only support two-stage stochastic programs. Support for models with more than two time stages will be considered in the future as this tool matures.

14.6.1 Additional Requirements for SMPS Conversion

To enable proper conversion of a PySP model to a set of SMPS files, the following additional requirements must be met:

1. The reference Pyomo model must include annotations that identify stochastic data locations in the second-stage problem.
2. All model variables must be declared in the ScenarioStructure.dat file.
3. The set of constraints and variables, and the overall sparsity structure of the objective and constraint matrix must not change across scenarios.

The bulk of this section discusses in-depth the annotations mentioned in the first point. The second point may come as a surprise to users that are not aware of the ability to *not* declare variables in the ScenarioStructure.dat file. Indeed, for most of the code in PySP, it is only critical that the variables for which non-anticipativity must be enforced need to be declared. That is, for a two-stage stochastic program, all second-stage variables can be left out of the ScenarioStructure.dat file when using commands such as `runef` and `runph`. However, conversion to SMPS format requires all variables to be properly assigned a decision stage by the user.

Note
Variables can be declared as *primary* by assigning them to a stage using the `StageVariables` assignment, or declared as *auxiliary* variables, which are assigned to a stage using `StageDerivedVariables` assignment. For algorithms such as PH, the distinction is meaningful and those variables that are fully determined by primary variables and the data should generally be assigned to `StageDerivedVariables` for their stage.

The third point may also come as a surprise, but the ability to handle a non-uniform problem structure in most PySP tools falls directly from the fact that the non-anticipativity conditions are all that is required in many cases. However, the conversion to SMPS format is based on a matrix representation of the problem where the stochastic coefficients are provided as a set of sparse matrix coordinates. This subsequently requires that the row and column dimensions as well as the sparsity structure of the problem does not change across scenarios.

14.6.2 Annotating Models for SMPS File Generation

Annotations are necessary for alerting the SMPS conversion routines of the locations of data that needs to be updated when changing from one scenario to another. Knowing these sparse locations allows decomposition algorithms to employ efficient methods for solving a stochastic program. In order to use the SMPS conversion tool, at least one of the following annotations must be declared on the reference Pyomo model:

- **PySP_StochasticRHSAnnotation**: indicates the existence of stochastic constraint right-hand-sides (or bounds) in second-stage constraints
- **PySP_StochasticMatrixAnnotation**: indicates the existence of stochastic variable coefficients in second-stage constraints
- **PySP_StochasticObjectiveAnnotation**: indicates the existence stochastic cost coefficients in the second-stage cost function

These will be discussed in further detail in the remaining sections. The following code snippet demonstrates how to import these annotations and declare them on a model.

```
from pyomo.pysp.annotations import *
model.stoch_rhs = PySP_StochasticRHSAnnotation()
model.stoch_matrix = PySP_StochasticMatrixAnnotation()
model.stoch_objective = PySP_StochasticObjectiveAnnotation()
```

Populating these annotations with entries is optional, and simply declaring them on the reference Pyomo model will alert the SMPS conversion routines that all coefficients appearing on the second-stage model should be assumed stochastic. That is, adding the lines in the previous code snippet alone implies that: (i) all **second-stage constraints** have stochastic bounds, (ii) all **first- and second-stage variables** appearing in **second-stage constraints** have stochastic coefficients, and (iii) all **first- and second-stage variables** appearing in the objective have stochastic coefficients.

PySP can attempt to determine the *stage*-ness of a constraint by examining the set of variables that appear in the constraint expression. E.g., a first-stage constraint is characterized as having only first-stage variables appearing in its expression. A second-stage constraint has at least one second-stage variable appearing in its expression. The stage of a variable is declared in the scenario tree provided to PySP. This method of constraint stage classification is not perfect. That is, one can very easily define a model with a constraint that uses only first-stage variables in an expression involving stochastic data. This constraint would be incorrectly identified as first-stage by the method above, even though the existence of stochastic data necessarily implies it is second-stage. To deal with cases such as this, an additional annotation is made available that is named **PySP_ConstraintStageAnnotation**. This annotation will be discussed further in a later section.

It is often the case that relatively few coefficients on a stochastic program change across scenarios. In these situations, adding explicit declarations within these annotations will allow for a more sparse representation of the problem and, consequently, more efficient solution by particular decomposition methods. Adding declarations to these annotations is performed by calling the `declare` method, passing some component as the initial argument. Any remaining argument requirements for this method are specific to each annotation. Valid types for the component argument typically include:

- `Constraint`: includes single constraint objects as well as constraint containers

- `Objective`: includes single objective objects as well as objective containers
- `Block`: includes Pyomo models as well single block objects and block containers

Any remaining details for adding declarations to the annotations mentioned thus far will be discussed in later sections. The remainder of this section discusses the semantics of these declarations based on the type for the component argument.

When the `declare` method is called with a component such as an indexed `Constraint` or a `Block` (model), the SMPS conversion routines will interpret this as meaning all constraints found within that indexed `Constraint` or on that `Block` (that have not been deactivated) should be considered. As an example, we consider the following partially declared concrete Pyomo model:

```
model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# variables declared as second-stage on the
# PySP scenario tree
model.z = Var()
model.y = Var()

# indexed constraint
model.r_index = Set(initialize=['a', 'b', 'c'])
def r_rule(model, i):
    return expr= p + i <= 1 *model.z + model.y * 5 <= 10 + q + i
model.r = Constraint(model.r_index, rule=r_rule)

# singleton constraint
model.c = Constraint(expr= p * model.z >= 1)

# a sub-block with a singleton constraint
model.b = Block()
model.b.c = Constraint(expr= q * model.y >= 1)
```

Here the local Python variables `p` and `q` serve as placeholders for data that changes with each scenario.

The following are equivalent annotations of the model, each declaring all of the constraints shown above as having stochastic right-hand-side data:

- Implicit form

```
model.stoch_rhs = PySP_StochasticRHSAnnotation()
```

- Implicit form for `Block` (model) assignment

```
model.stoch_rhs = PySP_StochasticRHSAnnotation()
model.stoch_rhs.declare(model)
```

- Explicit form for singleton constraint with implicit form for indexed constraint and sub-block

```
model.stoch_rhs = PySP_StochasticRHSAnnotation()
model.stoch_rhs.declare(model.r)
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b)
```

- Explicit form for singleton constraints at the model and sub-block level with implicit form for indexed constraint

```
model.stoch_rhs = PySP_StochasticRHSAnnotation()
model.stoch_rhs.declare(model.r)
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b.c)
```

- Fully explicit form for singleton constraints as well as all indices of indexed constraint

```
model.stoch_rhs = PySP_StochasticRHSAnnotation()
model.stoch_rhs.declare(model.r['a'])
model.stoch_rhs.declare(model.r['b'])
model.stoch_rhs.declare(model.r['c'])
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b.c)
```

Note that the equivalence of the first three bullet forms to the last two bullet forms relies on the following conditions being met: (1) `model.z` and `model.y` are declared on the second stage of the PySP scenario tree and (2) at least one of these second-stage variables appears in each of the constraint expressions above. Together, these two conditions cause each of the constraints above to be categorized as second-stage; thus, causing them to be considered by the SMPS conversion routines in the implicit declarations used by the first three bullet forms.

Warning
Pyomo simplifies product expressions such that terms with `0` coefficients are removed from the final expression. This can sometimes create issues with determining the correct stage classification of a constraint as well as result in different sparsity patterns across scenarios. This issue is discussed further in the later section entitled Edge Cases.

When it comes to catching errors in model annotations, there is a minor difference between the first bullet form from above (empty annotation) and the others. In the empty case, PySP will use exactly the set of second-stage constraints it is aware of. This set will either be determined through inspection of the constraint expressions or through the user-provided constraint-stage classifications declared using the **PySP_ConstraintStageAnnotation** annotation type. In the case where the stochastic annotation is not empty, PySP will verify that all constraints declared within it belong to the set of second-stage constraints it is aware of. If this verification fails, an error will be reported. This behavior is meant to aid users in debugging problems associated with non-uniform sparsity structure across scenarios that are, for example, caused by `0` coefficients in product expressions.

14.6.2.1 Annotations on AbstractModel Objects

Pyomo models defined using the `AbstractModel` object require the modeler to take further steps when making these annotations. In the `AbstractModel` setting, these assignments must take place within a `BuildAction`, which is executed only after the model has been constructed with data. As an example, the last bullet form from the previous section could be written in the following way to allow execution with either an `AbstractModel` or a `ConcreteModel`:

```
def annotate_rule(m):
    m.stoch_rhs = PySP_StochasticRHSAnnotation()
    m.stoch_rhs.declare(m.r['a'])
    m.stoch_rhs.declare(m.r['b'])
    m.stoch_rhs.declare(m.r['c'])
    m.stoch_rhs.declare(m.c)
    m.stoch_rhs.declare(m.b.c)
model.annotate = BuildAction(rule=annotate_rule)
```

Note that the use of `m` rather than `model` in the `annotate_rule` function is meant to draw attention to the fact that the model object being passed into the function as the first argument may not be the same object as the model outside of the function. This is in fact the case in the `AbstractModel` setting, whereas for the `ConcreteModel` setting they are the same object. We often use `model` in both places to avoid errors caused by forgetting to use the correct object inside the function (Python scoping rules handle the rest). Also note that a `BuildAction` must be declared on the model after the declaration of any components being accessed inside its rule function.

14.6.2.2 Stochastic Constraint Bounds (RHS)

If stochastic elements appear on the right-hand-side of constraints (or as constants in the body of constraint expressions), these locations should be declared using the **PySP_StochasticRHSAnnotation** annotation type. When components are declared with this annotation, there are no additional required arguments for the `declare` method. However, to allow for more flexibility when dealing with double-sided inequality constraints, the `declare` method can be called with at most one of the keywords `lb` or `ub` set to `False` to signify that one of the bounds is not stochastic. The following code snippet shows example declarations with this annotation for various constraint types.

```
from pyomo.pysp.annotations import PySP_StochasticRHSAnnotation

model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# a second-stage variable
model.y = Var()

# declare the annotation
model.stoch_rhs = PySP_StochasticRHSAnnotation()

# equality constraint
model.c = Constraint(expr= model.y == q)
model.stoch_rhs.declare(model.c)

# double-sided inequality constraint with
# stochastic upper bound
model.r = Constraint(expr= 0 <= model.y <= p)
model.stoch_rhs.declare(model.r, lb=False)

# indexed constraint using a BuildAction
model.C_index = RangeSet(1,3)
def C_rule(model, i):
    if i == 1:
        return model.y >= i * q
    else:
        return Constraint.Skip
model.C = Constraint(model.C_index, rule=C_rule)
def C_annotate_rule(model, i):
    if i == 1:
        model.stoch_rhs.declare(model.C[i])
    else:
        pass
model.C_annotate = BuildAction(model.C_index, rule=C_annotate_rule)
```

Note that simply declaring the **PySP_StochasticRHSAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all constraints identified as second-stage should be treated as having stochastic right-hand-side data. Calling the `declare` method on at least one component implies that the set of constraints considered should be limited to what is declared within the annotation.

14.6.2.3 Stochastic Constraint Matrix

If coefficients of variables change in the second-stage constraint matrix, these locations should be declared using the **PySP_StochasticMa** annotation type. When components are declared with this annotation, there are no additional required arguments for the `declare` method. Calling the `declare` method with the single component argument signifies that all variables encountered in the constraint expression (including first- and second-stage variables) should be treated as having stochastic coefficients. This can be limited to a specific subset of variables by calling the `declare` method with the `variables` keyword set to an explicit list of variable objects. The following code snippet shows example declarations with this annotation for various constraint types.

```
from pyomo.pysp.annotations import PySP_StochasticMatrixAnnotation

model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# a first-stage variable
model.x = Var()

# a second-stage variable
model.y = Var()

# declare the annotation
model.stoch_matrix = PySP_StochasticMatrixAnnotation()

# a singleton constraint with stochastic coefficients
# both the first- and second-stage variable
model.c = Constraint(expr= p * model.x + q * model.y == 1)
model.stoch_matrix.declare(model.c)
# an assignment that is equivalent to the previous one
model.stoch_matrix.declare(model.c, variables=[model.x, model.y])

# a singleton range constraint with a stochastic coefficient
# for the first-stage variable only
model.r = Constraint(expr= 0 <= p * model.x - 2.0 * model.y <= 10)
model.stoch_matrix.declare(model.r, variables=[model.x])
```

As is the case with the **PySP_StochasticRHSAnnotation** annotation type, simply declaring the **PySP_StochasticMatrixAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all constraints identified as second-stage should be considered, and, additionally, that all variables encountered in these constraints should be considered to have stochastic coefficients. Calling the `declare` method on at least one component implies that the set of constraints considered should be limited to what is declared within the annotation.

14.6.2.4 Stochastic Objective Elements

If the cost coefficients of any variables are stochastic in the second-stage cost expression, this should be noted using the **PySP_StochasticObjectiveAnnotation** annotation type. This annotation uses the same semantics for the `declare` method as the **PySP_StochasticMatrixAnnotation** annotation type, but with one additional consideration regarding any constants in the objective expression. Constants in the objective are treated as stochastic and automatically handled by the SMPS code. If the objective expression does not contain any constant terms or these constant terms do not change across scenarios, this behavior can be disabled by setting the keyword `include_constant` to `False` in a call to the `declare` method.

```
from pyomo.pysp.annotations import PySP_StochasticObjectiveAnnotation

model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# a first-stage variable
model.x = Var()

# a second-stage variable
model.y = Var()

# declare the annotation
```

```
model.stoch_objective = PySP_StochasticObjectiveAnnotation()

model.FirstStageCost = Expression(expr= 5.0 * model.x)
model.SecondStageCost = Expression(expr= p * model.x + q * model.y)
model.TotalCost = Objective(expr= model.FirstStageCost + model.SecondStageCost)

# each of these declarations is equivalent for this model
model.stoch_objective.declare(model.TotalCost)
model.stoch_objective.declare(model.TotalCost, variables=[model.x, model.y])
```

Similar to the previous annotation type, simply declaring the **PySP_StochasticObjectiveAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all variables appearing in the single active model objective expression should be considered to have stochastic coefficients.

14.6.2.5 Annotating Constraint Stages

Annotating the model with constraint stages is sometimes necessary to identify to the SMPS routines that certain constraints belong in the second time-stage even though they lack references to any second-stage variables. Annotation of constraint stages is achieved using the **PySP_ConstraintStageAnnotation** annotation type. If this annotation is added to the model, it is assumed that it will be fully populated with explicit stage assignments for every constraint in the model. The `declare` method should be called giving a `Constraint` or `Block` as the first argument and a positive integer as the second argument (1 signifies the first time stage). Example:

```
from pyomo.pysp.annotations import PySP_ConstraintStageAnnotation()

# declare the annotation
model.constraint_stage = PySP_ConstraintStageAnnotation()

# all constraints on this Block are first-stage
model.B = Block()
...
model.constraint_stage.declare(model.B, 1)

# all indices of this indexed constraint are first-stage
model.C1 = Constraint(..., rule=...)
model.constraint_stage.declare(model.C1, 1)

# all but one index in this indexed constraint are second-stage
model.C2 = Constraint(..., rule=...)
for index in model.C2:
    if index == 'a':
        model.constraint_stage.declare(model.C2[index], 1)
    else:
        model.constraint_stage.declare(model.C2[index], 2)
```

14.6.2.6 Edge Cases

The section discusses various points that may give users some trouble, and it attempts to provide more details about the common pitfalls associated with translating a PySP model to SMPS format.

- *Moving a Stochastic Objective to the Constraint Matrix*

It is often the case that decomposition algorithms theoretically support stochastic cost coefficients but the software implementation has not yet added support for them. This situation is easy to work around in PySP. One can simply augment the model with

an additional constraint and variable that *computes* the objective, and then use this variable in the objective rather than directly using the second-stage cost expression. Consider the following reference Pyomo model that has stochastic cost coefficients for both a first-stage and a second-stage variable in the second-stage cost expression:

```
from pyomo.pysp.annotations import PySP_StochasticObjectiveAnnotation

model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# first-stage variable
model.x = Var()
# second-stage variable
model.y = Var()

# first-stage cost expression
model.FirstStageCost = Expression(expr= 5.0 * model.x)
# second-stage cost expression
model.SecondStageCost = Expression(expr= p * model.x + q * model.y)

# define the objective as the sum of the
# stage-cost expressions
model.TotalCost = Objective(expr= model.FirstStageCost + model.SecondStageCost)

# declare that model.x and model.y have stochastic cost
# coefficients in the second stage
model.stoch_objective = PySP_StochasticObjectiveAnnotation()
model.stoch_objective.declare(model.TotalCost, variables=[model.x, model.y])
```

The code snippet below re-expresses this model using an objective consisting of the original first-stage cost expression plus a second-stage variable `SecondStageCostVar` that represents the second-stage cost. This is enforced by restricting the variable to be equal to the second-stage cost expression using an additional equality constraint named `ComputeSecondStageCost`. Additionally, the **PySP_StochasticObjectiveAnnotation** annotation type is replaced with the **PySP_StochasticMatrixAnnotation** annotation type.

```
from pyomo.pysp.annotations import PySP_StochasticMatrixAnnotation

model = ConcreteModel()

# data that is initialized on a per-scenario basis
p = ...
q = ...

# first-stage variable
model.x = Var()
# second-stage variables
model.y = Var()
model.SecondStageCostVar = Var()

# first-stage cost expression
model.FirstStageCost = Expression(expr= 5.0 * model.x)
# second-stage cost expression
model.SecondStageCost = Expression(expr= p * model.x + q * model.y)

# define the objective using SecondStageCostVar
# in place of SecondStageCost
model.TotalCost = Objective(expr= model.FirstStageCost + model.SecondStageCostVar)

# set the variable SecondStageCostVar equal to the
# expression SecondStageCost using an equality constraint
```

```
model.ComputeSecondStageCost = Constraint(expr= model.SecondStageCostVar == model. ↩
    SecondStageCost)

# declare that model.x and model.y have stochastic constraint matrix
# coefficients in the ComputeSecondStageCost constraint
model.stoch_matrix = PySP_StochasticMatrixAnnotation()
model.stoch_matrix.declare(model.ComputeSecondStageCost, variables=[model.x, model.y])
```

- *Stochastic Constant Terms*

The standard description of a linear program does not allow for a constant term in the objective function because this has no weight on the problem solution. Additionally, constant terms appearing in a constraint expression must be lumped into the right-hand-side vector. However, when modeling with an AML such as Pyomo, constant terms very naturally fall out of objective and constraint expressions.

If a constant terms falls out of a constraint expression and this term changes across scenarios, it is critical that this is accounted for by including the constraint in the **PySP_StochasticRHSAnnotation** annotation type. Otherwise, this would lead to an incorrect representation of the stochastic program in SMPS format. As an example, consider the following:

```
model = AbstractModel()

# a first-stage variable
model.x = Var()

# a second-stage variable
model.y = Var()

# a param initialized with scenario-specific data
model.p = Param()

# a second-stage constraint with a stochastic upper bound
# hidden in the left-hand-side expression
def c_rule(m):
    return (m.x - m.p) + m.y <= 10
model.c = Constraint(rule=c_rule)
```

Note that in the expression for constraint `c`, there is a fixed parameter `p` involved in the variable expression on the left-hand-side of the inequality. When an expression is written this way, it can be easy to forget that the value of this parameter will be pushed to the bound of the constraint when it is converted into linear canonical form. Remember to declare these constraints within the **PySP_StochasticRHSAnnotation** annotation type.

A constant term appearing in the objective expression presents a similar issue. Whether or not this term is stochastic, it must be dealt with when certain outputs expect the problem to be expressed as a linear program. The SMPS code in PySP will deal with this situation for you by implicitly adding a new second-stage variable to the problem in the final output file that uses the constant term as its coefficient in the objective and that is fixed to a value of 1.0 using a trivial equality constraint. The default behavior when declaring the **PySP_StochasticObjectiveAnnotation** annotation type will be to assume this constant term in the objective is stochastic. This helps ensure that the relative scenario costs reported by algorithms using the SMPS files will match that of the PySP model for a given solution. When moving a stochastic objective into the constraint matrix using the method discussed in the previous subsection, it is important to be aware of this behavior. A stochastic constant term in the objective would necessarily translate into a stochastic constraint right-hand-side when moved to the constraint matrix.

- *Stochastic Variable Bounds*

Although not directly supported, stochastic variable bounds can be expressed using explicit constraints along with the **PySP_StochasticR** annotation type to achieve the same effect.

- *Problems Caused by Zero Coefficients*

Expressions that involve products with some terms having `0` coefficients can be problematic when the zeros can become nonzero in certain scenarios. This can cause the sparsity structure of the LP to change across scenarios because Pyomo simplifies these expressions when they are created such that terms with a `0` coefficient are dropped. This can result in an invalid SMPS conversion. Of course, this issue is not limited to explicit product expressions, but can arise when the user implicitly assigns a variable a zero coefficient by outright excluding it from an expression. For example, both constraints in the following code snippet suffer from this same underlying issue, which is that the variable `model.y` will be excluded from the constraint expressions in a subset of scenarios (depending on the value of `q`) either directly due to a `0` coefficient in a product expressions or indirectly due to user-defined logic that is based off of the values of stochastic data.

```
model = ConcreteModel()

# data that is initialized on a per-scenario basis
# with q set to zero for this particular scenario
p = ...
q = 0

model.x = Var()
model.y = Var()

model.c1 = Constraint(expr= p * model.x + q * model.y == 1)

def c2_rule(model):
    expr = p * model.x
    if q != 0:
       expr += model.y
    return expr >= 0
model.c2 = Constraint(rule=c2_rule)
```

The SMPS conversion routines will attempt some limited checking to help prevent this kind of situation from silently turning the SMPS representation to garbage, but it must ultimately be up to the user to ensure this is not an issue. This is in fact the most challenging aspect of converting PySP's AML-based problem representation to the structure-preserving LP representation used in the SMPS format.

One way to deal with the `0` coefficient issue, which works for both cases discussed in the example above, is to create a *zero* `Expression` object. E.g.,

```
model.zero = Expression(expr=0)
```

This component can be used to add variables to a linear expression so that the resulting expression retains a reference to them. This behavior can be verified by examining the output from the following example:

```
from pyomo.environ import *
model = ConcreteModel()
model.x = Var()
model.y = Var()
model.zero = Expression(expr=0)

# an expression that does NOT
# retain model.y
print((model.x + 0 * model.y).to_string())              # -> x

# an equivalent expression that DOES
# retain model.y
```

```
print((model.x + model.zero * model.y).to_string())       # -> x + 0.0 * y

# an equivalent expression that does NOT
# retain model.y (so beware)
print((model.x + 0 * model.zero * model.y).to_string())   # -> x
```

14.6.3 Generating SMPS Input Files

As discussed at the start of this section, the `pysp2smps` command is used to execute the conversion to SMPS format. A detailed description of the command-line options available with this command can be obtained by executing the command `pysp2smps --help` in your shell. Here we discuss some of the basic inputs to this command.

Consider the baa99 example inside the `pysp/baa99` subdirectory that is distributed with the Pyomo examples (pyomo_examples.zip). Both the reference model and the scenario tree structure are defined in the file `baa99.py` using PySP callback functions. This model has been annotated to enable conversion to the SMPS format. Assuming one is in this example's directory, SMPS files can be generated for the model by executing the following shell command:

```
$\$$ pysp2smps -m baa99.py --basename baa99 \
     --output-directory sdinput/baa99
```

Assuming successful execution, this would result in the following files being created:

- `sdinput/baa99/baa99.mps`
- `sdinput/baa99/baa99.tim`
- `sdinput/baa99/baa99.sto`

The first file is the core problem file written in MPS file. It is written using an arbitrary scenario instances from the scenario tree as a reference. The second file indicates at which row and column the first and second time stages begin. The third file contains the location and values of stochastic data in the problem for each scenario. This file is generated by merging the individual output for each scenario in the scenario tree into separate BLOCK sections.

To ensure that the problem structure is the same and that all locations of stochastic data have been annotated properly, the script creates additional auxiliary files that are compared across scenarios. The command-line option `--keep-auxiliary-files` can be used to retain the auxiliary files that were generated for the template scenario used to write the core file. When this option is used with the above example, the following additional files will appear in the output directory:

- `sdinput/baa99/baa99.mps.det`
- `sdinput/baa99/baa99.sto.struct`
- `sdinput/baa99/baa99.row`
- `sdinput/baa99/baa99.col`

The `.mps.det` file is simply the core file for the reference scenario with the values for all stochastic coefficients set to zero. If this does not match for every scenario, then there are places in the model that still need to be declared on one or more of the stochastic data annotations. The `.row` and the `.col` files indicate the ordering of constraints and variables, respectively, that was used to write the core file. The `.sto.struct` file lists the nonzero locations of the stochastic data in terms of their row and column location in the core file. These files are created for each scenario instance in the scenario tree and placed inside of a subdirectory named `scenario_files` within the output directory. These files will be removed removed unless validation fails or the `--keep-scenario-files` option is used.

The `pysp2smps` command also supports parallel execution. This can significantly reduce the overall time required to produce the SMPS files when there are many scenarios. Parallel execution using PySP's Pyro-based tools can be performed using the steps below. Note that each of these commands can be launched in the background inside the same shell or in their own separate shells.

1. Start the Pyro name server:

```
$\$$ pyomo_ns -n localhost
```

1. Start the Pyro dispatch server:

```
$\$$ dispatch_srvr -n localhost
```

1. Start 8 ScenarioTree Servers (for the 625 baa99 scenarios)

```
$\$$ mpirun -np 8 scenariotreeserver --pyro-host=localhost
```

1. Run `pysp2smps` using the Pyro ScenarioTree Manager

```
$\$$ pysp2smps -m baa99.py --basename baa99 \
     --output-directory sdinput/baa99 \
     --pyro-required-scenariotreeservers=8 \
     --pyro-host=localhost --scenario-tree-manager=pyro
```

An annotated version of the farmer example is also provided. The model file can be found in the `pysp/farmer/smps_model` examples subdirectory. Note that the scenario tree for this model is defined in a separate file. When launching the `pysp2smps` command, a scenario tree structure file can be provided via the `--scenario-tree-location (-s)` command-line option. For example, assuming one is in the `pysp/farmer` subdirectory, the farmer model can be converted to SMPS files using the command:

```
$\$$ pysp2smps -m smps_model/ReferenceModel.py \
     -s scenariodata/ScenarioStructure.dat --basename farmer \
     --output-directory sdinput/farmer
```

Note that, by default, the files created by the `pysp2smps` command use shortened symbols that do not match the names of the variables and constraints declared on the Pyomo model. This is for efficiency reasons, as using fully qualified component names can result in significantly larger files. However, it can be useful in debugging situations to generate the SMPS files using the original component names. To do this, simply add the command-line option `--symbolic-solver-labels` to the command string.

The `pysp2smps` supports other formats for the core problem file (e.g., the LP format). The command-line option `--core-format` can be used to control this setting. Refer to the command-line help string for more information about the list of available format.

Chapter 15

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.

15.1 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 [AMPL]. 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.

15.2 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:

```
from pyomo.environ import *

model = ConcreteModel()

# Export integer data
model.priority = Suffix(direction=Suffix.EXPORT, datatype=Suffix.INT)

# Export and import floating point data
model.dual = Suffix(direction=Suffix.IMPORT_EXPORT)

# Store floating point data
model.junk = 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.

15.3 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.

```
from pyomo.environ import *

model = ConcreteModel()
model.x = Var()
model.y = Var([1,2,3])
model.foo = 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.setValue(model.x, 1.0)

# Same as above with dict interface
model.foo[model.x] = 1.0


# 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.setValue(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.setValue(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])              # -> raise KeyError
```

The non-dict method `clearValue` 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.clearValue(model.y)

print(model.foo[model.y[1]])          # -> raise KeyError

del model.foo[model.y[1]]             # -> raise KeyError

model.foo.clearValue(model.y[1])      # -> does nothing
```

A summary non-dict Suffix methods is provided here:

```
|  clearAllValues()
|      Clears all suffix data.
|
|  clearValue(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.
```

15.4 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.

```
from pyomo.environ import *
```

```
model = ConcreteModel()
model.dual = Suffix(direction=Suffix.IMPORT)
model.x = Var()
model.obj = Objective(expr=model.x)
model.con = 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.

```
print(instance.dual[instance.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.

15.5 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.

```
from pyomo.environ import *

model = ConcreteModel()
model.y = Var([1,2,3],within=NonNegativeReals)

model.sosno = Suffix(direction=Suffix.EXPORT)
model.ref = Suffix(direction=Suffix.EXPORT)

# Add entry for each index of model.y
model.sosno.setValue(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 interface, 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.

```
from pyomo.environ import *
from pyomo.opt import SolverFactory

### Create the ipopt solver plugin using the ASL interface
```

```
solver = 'ipopt'
solver_io = 'nl'
stream_solver = True      # True prints solver output to screen
keepfiles =     False     # True prints intermediate file names (.nl,.sol,...)
opt = SolverFactory(solver,solver_io=solver_io)

if opt is None:
    print("")
    print("ERROR: Unable to create solver plugin for %s "\
          "using the %s interface" % (solver, solver_io))
    print("")
    exit(1)
###

### Create the example model
model = ConcreteModel()
model.x1 = Var(bounds=(1,5),initialize=1.0)
model.x2 = Var(bounds=(1,5),initialize=5.0)
model.x3 = Var(bounds=(1,5),initialize=5.0)
model.x4 = Var(bounds=(1,5),initialize=1.0)
model.obj = Objective(expr=model.x1*model.x4*(model.x1+model.x2+model.x3) + model.x3)
model.inequality = Constraint(expr=model.x1*model.x2*model.x3*model.x4 >= 25.0)
model.equality = 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 = Suffix(direction=Suffix.IMPORT)
model.ipopt_zU_out = Suffix(direction=Suffix.IMPORT)
# Ipopt bound multipliers (sent to solver)
model.ipopt_zL_in = Suffix(direction=Suffix.EXPORT)
model.ipopt_zU_in = Suffix(direction=Suffix.EXPORT)
# Obtain dual solutions from first solve and send to warm start
model.dual = Suffix(direction=Suffix.IMPORT_EXPORT)
###

### Send the model to ipopt and collect the solution
print("")
print("INITIAL SOLVE")
results = opt.solve(model,keepfiles=keepfiles,tee=stream_solver)
# load the results (including any values for previously declared
# IMPORT / IMPORT_EXPORT Suffix components)
model.solutions.load_from(results)
###

### 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)
opt.options['warm_start_init_point'] = 'yes'
opt.options['warm_start_bound_push'] = 1e-6
opt.options['warm_start_mult_bound_push'] = 1e-6
opt.options['mu_init'] = 1e-6
###

### Send the model and suffix information to ipopt and collect the solution
print("")
print("WARM-STARTED SOLVE")
```

```
# The solver plugin will scan the model for all active suffixes
# valid for importing, which it will store into the results object
results = opt.solve(model,keepfiles=keepfiles,tee=stream_solver)
# load the results (including any values for previously declared
# IMPORT / IMPORT_EXPORT Suffix components)
model.solutions.load_from(results)
###
```

The difference in performance can be seen by examining Ipopt's iteration log with and without warm starting:

- Without Warmstart:

```
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 3.60e-02 5.05e-01  -1.0 1.61e-01    -  1.00e+00 1.00e+00h  1
   3  1.6849424e+01 2.78e-01 6.68e-02  -1.7 2.85e-01    -  7.94e-01 1.00e+00h  1
   4  1.7051199e+01 4.71e-03 2.78e-03  -1.7 6.06e-02    -  1.00e+00 1.00e+00h  1
   5  1.7011979e+01 7.19e-03 8.50e-03  -3.8 3.66e-02    -  9.45e-01 9.98e-01h  1
   6  1.7014271e+01 1.74e-05 9.78e-06  -3.8 3.33e-03    -  1.00e+00 1.00e+00h  1
   7  1.7014021e+01 1.23e-07 1.82e-07  -5.7 2.69e-04    -  1.00e+00 1.00e+00h  1
   8  1.7014017e+01 1.77e-11 2.52e-11  -8.6 3.32e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 8
```

- With Warmstart:

```
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 4.48e-12 6.43e-12  -9.0 1.92e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 2
```

15.6 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.

```
from pyomo.environ import *

model = AbstractModel()
model.x = Var()
model.c = Constraint(expr= model.x >= 1)

def foo_rule(m):
   return ((m.x, 2.0), (m.c, 3.0))
model.foo = Suffix(rule=foo_rule)

# Instantiate the model
inst = model.create()
print(inst.foo[model.x]) # -> raise KeyError
print(inst.foo[inst.x])  # -> 2.0
print(inst.foo[inst.c])  # -> 3.0
```

The next example shows an abstract model where suffixes are attached only to the variables:

```
from pyomo.environ import *

model = AbstractModel()
model.I = RangeSet(1,4)
model.x = Var(model.I)
def c_rule(m, i):
    return m.x[i] >= i
model.c = Constraint(model.I, rule=c_rule)

def foo_rule(m):
    return ((m.x[i], 3.0*i) for i in m.I)
model.foo = Suffix(rule=foo_rule)

# instantiate the model
inst = model.create_instance()
for i in inst.I:
    print (i, inst.foo[inst.x[i]])
```

Chapter 16

DAE Toolbox

The DAE toolbox allows users to incorporate differential equations in a Pyomo model. The modeling components in this toolbox 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. The toolbox also includes model transformations which use a simultaneous discretization approach for transforming a DAE model into an algebraic model.

16.1 DAE Modeling Components

The DAE toolbox introduces three new modeling components to Pyomo:

ContinuousSet
: Used to represent bounded continuous domains

DerivativeVar
: Defines how a `Var` will be differentiated or the derivatives to be included in the model

Integral
: Defines an integral over a continous domain

As will be shown later, differential equations can be declared using using these new DAE modeling components along with the standard Pyomo `Var` and `Constraint` components.

16.1.1 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. 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.

The following code snippet shows examples of declaring a `ContinuousSet` component on a concrete Pyomo model:

```
# Required imports
from pyomo.environ import *
from pyomo.dae import *

model = ConcreteModel()

# declare by providing bounds
model.t = ContinuousSet(bounds=(0,5))
```

```
# declare by initializing with desired discretization points
model.x = ContinuousSet(initialize=[0,1,2,5])
```

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
from pyomo.environ import *
from pyomo.dae import *

model = AbstractModel()

# The ContinuousSet below will be initialized using the points
# in the data file when a model instance is created.
model.t = ContinuousSet()
```

Note
A `ContinuousSet` may not be constructed unless two numeric bounding points are provided.

Note
If a separate data file is used to initialize a `ContinuousSet`, it is done using the *set* command and not *continuousset*

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.

SUMMARY OF CONTINUOUSSET METHODS

get_finite_elements()
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 over each finite element. Otherwise this method returns a list of all the discretization points in the ContinuousSet.

get_discretization_info()
Returns a dictionary containing information on the discretization scheme that has been applied to the ContinuousSet.

get_changed()
Returns "True" if additional points were added to the ContinousSet while applying a discretization scheme

get_upper_element_boundary(value)
Returns the first finite element point that is greater than or equal to the value sent to the function.

get_lower_element_boundary(value)
Returns the first finite element point that is less than or equal to the value sent to the function.

16.1.2 DerivativeVar

The `DerivativeVar` component is used to declare a derivative of a `Var`. 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.

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
from pyomo.environ import *
from pyomo.dae import *

model = ConcreteModel()
model.s = Set(initialize=['a','b'])
model.t = ContinuousSet(bounds=(0,5))
model.l = ContinuousSet(bounds=(-10,10))

model.x = Var(model.t)
model.y = Var(model.s,model.t)
model.z = 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`.

Another way to use derivatives without explicitly declaring `DerivativeVar` components is to use the .derivative() method on a variable within an expression or constraint. For example:

```
# Required imports
from pyomo.environ import *
from pyomo.dae import *

model = ConcreteModel()
model.t = ContinuousSet(bounds=(0,5))
model.x = Var(model.t)

# Create the first derivative of model.x with respect to model.t
# within a constraint rule.
def _diffeq_rule(m,i):
    return m.x[i].derivative(m.t) == m.x[i]**2
model.diffeq = Constraint(model.t,rule=_diffeq_rule)
```

In the above example a DerivatveVar component representing the desired derivative will automatically be added to the Pyomo model when the constraint is constructed. The .derivative() method accepts positional arguments representing what the derivative is being taken with respect to.

Note
If a variable is indexed by a single ContinuousSet then the .derivative() method with no positional arguments may be used to specify the first derivative of that variable with respect to the ContinuousSet.

16.2 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
from pyomo.environ import *
from pyomo.dae import *

model = ConcreteModel()
model.s = Set(initialize=['a','b'])
model.t = ContinuousSet(bounds=(0,5))
model.l = ContinuousSet(bounds=(-10,10))

model.x = Var(model.s,model.t)
model.y = Var(model.t,model.l)
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,i,j):
    if j == 0:
        return Constraint.Skip
    return m.x[i].derivative(m.t) == m.x[i]**2
model.ode = Constraint(model.s,model.t,rule=_ode_rule)

# A partial differential equation
def _pde_rule(m,i,j):
    if i == 0 or j == -10 or j == 10:
        return Constraint.Skip
    return m.dydt[i,j] == m.dydl2[i,j]
model.pde = Constraint(model.t,model.l,rule=_pde_rule)
```

Note
Often a modeler does not want to apply a differential equation at one or both boundaries of a continuous domain. This must be addressed explicitly in the Constraint declaration using *Constraint.Skip* as shown above. By default, a Constraint declared over a ContinuousSet will be applied at every discretization point contained in the set.

16.3 Declaring Integrals

The Integral component is still under development but some basic functionality is available in the current Pyomo release. Integrals must be 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 releases of this tool will include more sophisticated numerical integration methods.

Declaring an Integral component is similar to declaring an Expression component. A simple example is shown below:

```
def _intX(m,i):
    return (m.X[i]-m.X_desired)**2
model.intX = Integral(model.time,wrt=model.time,rule=_intX)

def _obj(m):
    return m.scale*m.intX
model.obj = 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 `ContinousSet` 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)

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. The reason for this is to keep track of the order of the indexing sets. This logic should become more clear with the following example showing a double integral over the `ContinuousSet` components *t1* and *t2*. In addition, the expression is also indexed by the `Set` *s*.

```
def _intX1(m,i,j,s):
    return (m.X[i,j,s]-m.X_desired[j,s])**2
model.intX1 = Integral(model.t1,model.t2,model.s,wrt=model.t1,rule=_intX1)

def _intX2(m,j,s):
    return (m.intX1[j,s]-m.X_desired[s])**2
model.intX2 = Integral(model.t2,model.s,wrt=model.t2,rule=_intX2)

def _obj(m):
    return sum(model.intX2[k] for k in m.s)
model.obj = Objective(rule=_obj)
```

16.4 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 this tool 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, 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.

16.4.1 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:

$$\begin{aligned}
&\text{Given} dx/dt = f(t,x) \text{and} x(t0) = x_0 \\
&\text{discretize} t \text{and} x \text{such that} \\
&x(t0+kh) = x_k \\
&x_{k+1} = x_k + h * f(t_{k+1}, x_{k+1}) \\
&t_{k+1} = t_k + h
\end{aligned}$$

where h is the step size between discretization points or the size of each finite element. These equations are generated automatically as `Constraint` components 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 all ContinuousSets.

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 `ContinousSet` 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.

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample import model

# Discretize model using Backward Difference method
discretizer = 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 = Constraint(rule=_sum_limit)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

16.4.2 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.

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample2 import model

# Discretize model using Radau Collocation
discretizer = 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 = Objective(rule=obj_rule)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

16.4.3 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 = 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 = 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 = TransformationFactory('dae.finite_difference')
disc_fe.apply_to(model,wrt=model.t1,nfe=10)
disc_col = 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.

16.4.4 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 trasnformation 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 Finite_Difference_Transformation 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.

Chapter 17

Scripts

There are two main ways to add scripting for Pyomo models: using Python scripts and using callbacks for the `pyomo` command that alter or supplement its workflow.

Note
The examples are written to conform with the Python version 3 `print` function. If executed with Python version 2, the output from `print` statements may not look as nice.

17.1 Python Scripts

17.1.1 Iterative Example

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.

Note
The built-in code for printing solutions prints only non-zero variable values. So if you run this code, no variable values will be output for the first solution found because all of the variables are zero. However, other information about the solution, such as the objective value, will be displayed.

```
# iterative1.py

from pyomo.environ import *
from pyomo.opt import SolverFactory

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

#
# A simple model with binary variables and
# an empty constraint list.
#
model = AbstractModel()
model.n = Param(default=4)
model.x = Var(RangeSet(model.n), within=Binary)
def o_rule(model):
    return summation(model.x)
model.o = Objective(rule=o_rule)
model.c = 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):
    instance.solutions.load_from(results)

    expr = 0
    for j in instance.x:
        if instance.x[j].value == 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:

```
# iterative1.py
from pyomo.environ import *
from pyomo.opt import SolverFactory
```

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

```
# Create a solver
opt = 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 = AbstractModel()
```

```
model.n = Param(default=4)
model.x = Var(RangeSet(model.n), within=Binary)
def o_rule(model):
    return summation(model.x)
model.o = 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 = 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):
```

The next line associates the results obtained with the instance. This then enables direct queries of solution values in subsequent lines using variable names contained in the instance:

```
    instance.solutions.load_from(results)
```

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 instance.x[j].value == 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` method to get it.

The next line adds to the constaint 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 exerise for the reader.

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

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

17.2 Changing the Model or Data and Re-solving

The `iterative1.py` example 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 `instance` object rather than the `model` object so that we do not have to create a new `model` object. Here is the basic idea:

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 someting in `instance`
5. Call presolve
6. Solve `instance` again

If `instance` has a parameter whose name is in `ParamName` with an index that is in `idx`, the the value in `NewVal` can be assigned to it using

```
    getattr(instance, ParamName)[idx] = NewVal
```

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

```
    getattr(instance, ParamName)[None] = NewVal
```

or else

```
    getattr(instance, ParamName).set_value(NewVal)
```

The function `getattr` is provided by Python. For more information about access to Pyomo parameters, see the section in this document on Param Access. Note that for concrete models, the model is the instance.

17.3 Fixing Variables and Re-solving

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

```
# iterative2.py

from pyomo.environ import *
from pyomo.opt import SolverFactory

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

#
# A simple model with binary variables and
# an empty constraint list.
#
```

```
model = AbstractModel()
model.n = Param(default=4)
model.x = Var(RangeSet(model.n), within=Binary)
def o_rule(model):
    return summation(model.x)
model.o = Objective(rule=o_rule)
model.c = 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
instance.solutions.load_from(results)

if instance.x[2] == 0:
    instance.x[2] = 1
else:
    instance.x[2] = 0
instance.x[2].fixed = True

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:

```
instance.solutions.load_from(results)

if instance.x[2] == 0:
    instance.x[2] = 1
else:
    instance.x[2] = 0
instance.x[2].fixed = True
results = opt.solve(instance)
```

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

```
instance.solutions.load_from(results)

if instance.x[2] == 0:
    instance.x[2].setlb(1)
    instance.x[2].setub(1)
else:
    instance.x[2].setlb(0)
    instance.x[2].setub(0)
results = opt.solve(instance)
```

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 Variable Access.

Note that `instance.x.fix(2)` is equivalent to

```
instance.x.value = 2
instance.x.fixed = True
```

and `instance.x.fix()` is equivalent to `instance.x.fixed =True`

17.4 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 using

```
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.

17.5 Pyomo Callbacks

Pyomo enables altering or extending its workflow through the use of callbacks that are defined in the model file. Taken together, the callbacks allow for consruction of a rich set of workflows. However, many users might be interesting in making use of only one or two of the callbacks. They are executable Python functions with pre-defined names:

- `pyomo_preprocess`: Preprocessing before model construction
- `pyomo_create_model`: Constructs and returns the model object
- `pyomo_create_modeldata`: Constructs and returns a ModelData object
- `pyomo_print_model`: Display model information
- `pyomo_modify_instance`: Modify the model instance
- `pyomo_print_instance`: Display instance information
- `pyomo_save_instance`: Write the model instance to a file
- `pyomo_print_results`: Display the results of optimization
- `pyomo_save_results`: Store the optimization results
- `pyomo_postprocess`: Postprocessing after optimization

Many of these functions have arguments, which must be declared when the functions are declared. This can be done either by listing the arguments, as we will show below, or by providing a dictionary for arbitrary keyword arguments in the form `**kwds`. If the abritrary keywords are used, then the arguments are access using the get method. For example the `pyomo_preprocess` function takes one argument (as will be described below) so the following two function will produce the same output:

```
def pyomo_preprocess(options=None):
   if options == None:
      print "No command line options were given."
   else:
      print "Command line arguments were: %s" % options
```

```
def pyomo_preprocess(**kwds):
   options = kwds.get('options',None)
   if options == None:
      print "No command line options were given."
   else:
      print "Command line arguments were: %s" % options
```

To access the various arguments using the `**kwds` argument, use the following strings:

- *`options`* for the command line arguments dictionary

- *model-options* for the --model-options dictionary
- *model* for a model object
- *instance* for an instance object
- *results* for a results object

17.5.1 pyomo_preprocess

This function has one argument, which is an enhanced Python dictionary containing the command line options given to launch Pyomo. It is called before model construction so it augments the workflow. It is defined in the model file as follows:

```
def pyomo_preprocess(options=None):
```

17.5.2 pyomo_create_model

This function is for experts who want to replace the model creation functionality provided by the pyomo script with their own. It takes two arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo and a dictionary with the options given in the --model-options argument to the pyomo command. The function must return the model object that has been created.

17.5.3 pyomo_create_modeldata

Users who employ ModelData objects may want to give their own method for populating the object. This function returns returns a ModelData object that will be used to instantiate the model to form an instance. It takes two arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo and a model object.

17.5.4 pyomo_print_model

This callback is executed between model creation and instance creation. It takes two arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo and a model object.

17.5.5 pyomo_modify_instance

This callback is executed after instance creation. It takes three arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo, a model object, and an instance object.

17.5.6 pyomo_print_instance

This callback is executed after instance creation (and after the pyomo_modify_instance callback). It takes two arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo and an instance object.

17.5.7 pyomo_save_instance

This callback also takes place after instance creation and takes It takes two arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo and an instance object.

17.5.8 pyomo_print_results

This callback is executed after optimization. It takes three arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo, an instance object, and a results object. Note that the --print-results option provides a way to print results; this callback is intended for users who want to customize the display.

17.5.9 pyomo_save_results

This callback is executed after optimization. It takes three arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo, an instance object, and a results object. Note that the `--save-results` option provides a way to store results; this callback is intended for users who want to customize the format or contents.

17.5.10 pyomo_postprocess

This callback is also executed after optimization. It also takes three arguments: an enhanced Python dictionary containing the command line options given to launch Pyomo, an instance object, and a results object.

17.6 Accessing Variable Values

17.6.1 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.

17.6.2 One Variable from a Python Script

Assuming the model has been instantiated and solved and the results have been loded 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 `instance.quant.value`. 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 very simple model and there are no parameter values to be read from a data file, so the `model.create_instance()` call does not specify a file name. In this example, the value of `x[2]` is accessed.

```
# noiteration1.py

from pyomo.environ import *
from pyomo.opt import SolverFactory

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

#
# A simple model with binary variables and
# an empty constraint list.
#
model = AbstractModel()
model.n = Param(default=4)
model.x = Var(RangeSet(model.n), within=Binary)
def o_rule(model):
    return summation(model.x)
model.o = Objective(rule=o_rule)
model.c = ConstraintList()

# Create a model instance and optimize
instance = model.create_instance()
results = opt.solve(instance)
instance.solutions.load_from(results)
```

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

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.

17.6.3 All Variables from a Python Script

As with one variable, we assume that the model has been instantiated and solved and the results have been loded back into the instance object using `instance.solutions.load_from(results)` and the code includes the line `from pyomo.core import Var`, 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. Assuming the instance object has the name `instance`, the following code snippet displays all variables and their values:

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

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 loded back into the instance object using `instance.solutions.load_from(results)` and that the code includes the line `from pyomo.core import Var` here is a code snippet for fixing all integers at their current value:

```
for v in instance.component_objects(Var, active=True):
    varobject = getattr(instance, v)
    if isinstance(varobject.domain, IntegerSet) or isinstance(varobject.domain, BooleanSet) ↩
        :
        print ("fixing "+str(v))
        for index in varobject:
            varobject[index].fixed = True # fix the current value
```

Another way to access all of the variables (particularly if there are blocks) is as follows:

```
for v in model.component_objects(Var):
    print("FOUND VAR:" + v.cname(True))
    v.pprint()

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

The use of `True` as an argument to `cname` indicates that the full name is desired.

17.6.4 All Variables from Workflow Callbacks

The `pyomo_print_results`, `pyomo_save_results`, and `pyomo_postprocess` callbacks from the `pyomo` script take the instance as one of their arguments and the instance has the solver results at the time of the callback so the body of the callback matches the code snipped given for a Python script.

For example, if the following defintion were included in the model file, then the `pyomo` command would output all variables and their values (including those variables with a value of zero):

```
def pyomo_print_results(options, instance, results):
    from pyomo.core import Var
    for v in instance.component_objects(Var, active=True):
        print ("Variable "+str(v))
        varobject = getattr(instance, v)
        for index in varobject:
            print ("   ",index, varobject[index].value)
```

17.7 Accessing Parameter Values

Access to paramaters is completely analgous to access to variables. For example, here is a code snippet to print the name and value of every Parameter:

```
from pyomo.core import Param
for p in instance.component_objects(Param, active=True):
    print ("Parameter "+str(p))
    parmobject = getattr(instance, p)
    for index in parmobject:
        print ("   ",index, parmobject[index].value)
```

NOTE:The value of a Param can be returned as None+ if no data was specified for it. This will be true even if a default value was given. To inspect the default value of a Param, replace `.value` with `.default()` but note that the default might be a function.

17.8 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`.

17.8.1 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 = Suffix(direction=Suffix.IMPORT)
```

See the section on [?simpara] 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"
from pyomo.core import Constraint
for c in instance.component_objects(Constraint, active=True):
    print ("   Constraint "+str(c))
    cobject = getattr(instance, str(c))
    for index in cobject:
        print ("      ", index, instance.dual[cobject[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 (display, this case) 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`.

```
# driveabs2.py
from __future__ import division
from pyomo.environ import *
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 = Suffix(direction=Suffix.IMPORT)

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

# display all duals
print ("Duals")
from pyomo.core import Constraint
for c in instance.component_objects(Constraint, active=True):
    print ("   Constraint",c)
    cobject = getattr(instance, str(c))
    for index in cobject:
        print ("      ", index, instance.dual[cobject[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.

```
# driveconc1.py
from __future__ import division
from pyomo.environ import *
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 = Suffix(direction=Suffix.IMPORT)

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

# display all duals
print ("Duals")
from pyomo.core import Constraint
for c in model.component_objects(Constraint, active=True):
    print ("   Constraint",c)
    cobject = getattr(model, str(c))
    for index in cobject:
        print ("      ", index, model.dual[cobject[index]])
```

17.8.2 All Duals from Workflow Callbacks

The `pyomo` script needs to be instructed to obtain duals, either by using a command line option such as `--solver-suffixes=dual` or by adding code in the `pyomo_preprocess` callback to add `solver-suffixes` to the list of command line arguments if it is not there and to add *`dual`* to its list of arguments if it is there, but *`dual`* is not. If a suffix with the name dual has been declared on the model the use of the command line option or `pyomo_preprocess` callback is not required.

The `pyomo_print_results`, `pyomo_save_results`, and `pyomo_postprocess` callbacks from the `pyomo` script take the instance as one of their arguments and the instance has the solver results at the time of the callback so the body of the callback matches the code snipped given for a Python script.

For example, if the following definition were included in the model file, then the `pyomo` command would output all constraints and their duals.

```
def pyomo_print_results(options, instance, results):
    # display all duals
    print ("Duals")
    from pyomo.core import Constraint
    for c in instance.component_objects(Constraint, active=True):
        print ("   Constraint",c)
        cobject = getattr(instance, c)
        for index in cobject:
            print ("      ", index, instance.dual[cobject[index]])
```

Note

If the `--solver-suffixes` command line option is used to request constraint duals, an IMPORT style Suffix component will be added to the model by the `pyomo` command.

17.9 Accessing Slacks

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

17.10 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:

```
instance = model.create()
results = opt.solve(instance)
print ("The solver returned a status of:"+str(results.Solution.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):
     # this is feasible and optimal
elif results.solver.termination_condition == TerminationCondition.infeasible:
     # do something about it? or exit?
else:
     # something else is wrong
     print (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? ...
```

17.11 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.

17.12 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 = SolverFactory('cbc')
optimizer.options["threads"] = 4
```

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

Sometime 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.

17.13 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 = 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).

17.14 Solving Multiple Instances in Parallel

Use of parallel solvers for PySP is discussed in the section on Parallel PySP.

Solvers are controlled by solver servers. The pyro mip solver server is launched with the command `pyro_mip_server`. This command may be repeated to launch as many solvers as are desired. A name server and a dispatch server must be running and accessible to the process that runs the script that will use the mip servers as well as to the mip servers. The name server is launched using the command `pyomo_ns` and then the dispatch server is launched with `dispatch_srvr`. Note that both commands contain an underscore. Both programs keep running until terminated by an external signal, so it is common to pipe their output to a file. The commands are:

- Once: `pyomo_ns`
- Once: `dispatch_srvr`

- Multiple times: `pyro_mip_server`

This example demonstrates how to use these services to solve two instances in parallel.

```
# parallel.py
from __future__ import division
from pyomo.environ import *
from pyomo.opt import SolverFactory
from pyomo.opt.parallel import SolverManagerFactory
import sys

action_handle_map = {} # maps action handles to instances

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

# create a solver manager
# 'pyro' could be replaced with 'serial'
solver_manager = SolverManagerFactory('pyro')
if solver_manager is None:
    print "Failed to create solver manager."
    sys.exit(1)

#
# A simple model with binary variables and
# an empty constraint list.
#
model = AbstractModel()
model.n = Param(default=4)
model.x = Var(RangeSet(model.n), within=Binary)
def o_rule(model):
    return summation(model.x)
model.o = Objective(rule=o_rule)
model.c = ConstraintList()

# Create two model instances
instance1 = model.create()

instance2 = model.create()
instance2.x[1] = 1
instance2.x[1].fixed = True

# send them to the solver(s)
action_handle = solver_manager.queue(instance1, opt=optsolver, warmstart=False, tee=True,  ↩
    verbose=False)
action_handle_map[action_handle] = "Original"
action_handle = solver_manager.queue(instance2, opt=optsolver, warmstart=False, tee=True,  ↩
    verbose=False)
action_handle_map[action_handle] = "One Var Fixed"

# retrieve the solutions
for i in range(2): # we know there are two instances
    this_action_handle = solver_manager.wait_any()
    solved_name = action_handle_map[this_action_handle]
    results = solver_manager.get_results(this_action_handle)
    print "Results for",solved_name
    print results
```

This example creates two instances that are very similar and then sends them to be dispatched to solvers. If there are two solvers, then these problems could be solved in parallel (we say "could" because for such trivial problems to be actually solved in parallel, the solvers would have to be very, very slow). This example is non-sensical; the goal is simply to show `solver_manager.queue` to submit jobs to a name server for dispatch to solver servers and `solver_manager.wait_any` to recover the results.

The `wait_all` function is similar, but it takes a list of action handles (returned by `queue`) as an argument and returns all of the results at once.

17.15 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 pyutilib.services import TempFileManager
TempfileManager.tempdir = YourDirectoryNameGoesHere
```

Chapter 18

Pyomo Solver Interfaces

This chapter describes how Pyomo interfaces with different solvers.

Chapter 19

Using Black-Box Optimizers with Pyomo.Opt

Many optimization software packages contain *black-box* optimizers, which perform optimization without using detailed knowledge of the structure of an optimization problem. Thus, black-box optimizers require a generic interface for optimization problems that defines key features of problems, like objectives and constraints.

The `pyomo.opt` package contains the `pyomo.opt.blackbox` subpackage, which provides facilities for (a) integrating Pyomo solvers with blackbox optimization applications and (b) wrapping Pyomo models for use by external blackbox optimizers. We illustrate these capabilities in this chapter with simple examples that illustrate the use of `pyomo.opt.blackbox`.

19.1 Defining and Optimizing Simple Black-Box Applications

Many black-box optimizers interact with an optimization problem by executing a separate process that computes properties of the optimization problem. This process typically reads an input file that defines the requested properties and writes an output file that contains the computed values. Unfortunately, no standards have emerged for black-box optimizers that interact with problems in this manner. Thus, different file formats are used by different optimizer software packages.

19.1.1 Defining an Optimization Problem

The `pyomo.opt.blackbox` package provides several Python classes for optimization problems that coordinates file I/O for the user and simplifies the definition of simple black-box problems. The `RealOptProblem` class provides a generic interface for continuous optimization problems (i.e. with real variables). The following example defines a simple continuous optimization problem:

```
class RealProblem1(RealOptProblem):

    def __init__(self):
        RealOptProblem.__init__(self)
        self.lower=[0.0, -1.0, 1.0, None]
        self.upper=[None, 0.0, 2.0, -1.0]
        self.nvars=4

    def function_value(self, point):
        self.validate(point)
        return point.vars[0] - point.vars[1] + (point.vars[2]-1.5)**2 + (point.vars[3]+2) ↩
            **4
```

This problem is equivalent to the following problem definition:

$$\begin{array}{ll} \min & x_0 - x_1 + (x_2 - 1.5)^2 + (x_3 + 2)^4 \\ \text{s.t.} & 0 \leq x_0 \\ & -1 \leq x_1 \leq 0 \\ & 0 \leq x_2 \leq 2 \\ & x_3 \leq -1 \end{array}$$

Note that the problem class does *not* specify the sense of the optimization problem. These problem classes are not a complete specification of an optimization problem. Rather, an instance of a problem class can compute information about the problem that is used during optimization.

Similarly, the `MixedIntOptProblem` class provides a generic interface for mixed-integer optimization problems, which may contain real variables, integer variables and binary variables. The following example defines a simple mixed-integer optimization problem:

```
class MixedIntProblem1(MixedIntOptProblem):

    def __init__(self):
        MixedIntOptProblem.__init__(self)
        self.real_lower=[0.0]*4
        self.real_upper=[2.0]*4
        self.int_lower=[-2]*3
        self.int_upper=[0]*3
        self.nreal=4
        self.nint=3
        self.nbinary=2

    def function_value(self, point):
        self.validate(point)
        return sum((x-1)**2 for x in self.reals) + \
               sum((y+1)**2 for y in self.ints) + \
               sum(b for b in self.bits)
```

This problem is equivalent to the following problem definition:

$$\begin{array}{ll} \min & \sum_{i=1}^{4}(x_i-1)^2+\sum_{i=1}^{3}(y_i+1)^2+\sum_{i=1}^{2}z_i \\ \text{s.t.} & 0 \leq x_i \leq 2 \\ & -2 \leq y_i \leq 0 \\ & z_i \in \{0,1\} \end{array}$$

19.1.2 Optimizating with Coliny Solvers

The `Coliny` software library supports interfaces to a variety of black-box optimizers <Coliny>. The `coliny` executable reads an XML specification of the optimization problem and solver, as well as a specification of how the optimizer is applied. Consider the following XML specification:

```
<!--- RealProblem1.xml

    This Coliny XML specification illustrates the execution of the
    colin:ls solver on the RealProblem1 problem.
--->

<ColinInput>

  <Problem type="MINLP0">
     <Domain>
        <RealVars num="4">
          <Lower index="1" value="0.0"/>
          <Lower index="2" value="-1.0"/>
          <Lower index="3" value="1.0"/>

          <Upper index="2" value="0.0"/>
          <Upper index="3" value="2.0"/>
          <Upper index="4" value="-1.0"/>
        </RealVars>
     </Domain>

     <Driver>
```

```
        <Command>RealProblem1.py</Command>
     </Driver>
  </Problem>

  <Solver type="colin:ls">
     <InitialPoint>
       0.0 2.0 -1.0 10.0
     </InitialPoint>
     <Options>
        <Option name="min_function_value">-1.0</Option>
     </Options>
  </Solver>
</ColinInput>
```

This XML specification defines a `MINLP0` problem, which indicated that this is a mixed-integer problem that supports zero-order derivatives (i.e. no derivatives). This problem has four real variables with lower and upper bounds specified. The problem values are computed with the `RealProblem1.py` command-line, which defines and uses the `RealProblem1` class defined above:

```
#!/usr/bin/env python
#
# RealProblem1.py

import sys
from pyomo.opt.blackbox import RealOptProblem

class RealProblem1(RealOptProblem):

    def __init__(self):
        RealOptProblem.__init__(self)
        self.lower=[0.0, -1.0, 1.0, None]
        self.upper=[None, 0.0, 2.0, -1.0]
        self.nvars=4

    def function_value(self, point):
        self.validate(point)
        return point.vars[0] - point.vars[1] + (point.vars[2]-1.5)**2 + (point.vars[3]+2) ↩
            **4

problem = RealProblem1()
problem.main(sys.argv)
```

Note that this command is a Python script that includes the shebang character sequence on the first line. On Linux and Unix systems, this line indicates that this is a script that is executed using the `python` command that is found in the user environment. Thus, this example assumes that the `python` command has `pyomo.opt` installed. Since multiple versions of Python can be installed on a single computer, the XML `Command` element may need to be defined with an explicitly Python version. For example, if Python 2.6 is installed in `/usr/local` with `pyomo.opt`, then the `Command` element would look like:

```
   <Command>/usr/local/bin/python26 RealProblem1.py</Command>
```

Additionally, the duplication of bounds information between `RealProblem1.py` and `RealProblem1.xml` is not strictly necessary in this example. The bounds information in `RealProblem1.py` is used in the `validate` method to verify that the point being evaluated is consistent with the bounds information. We can generally assume that the Coliny solver will only evaluate feasible points, so a simpler problem definition can be used:

```
#!/usr/bin/env python
#
# RealProblem2.py

import sys
from pyomo.opt.blackbox import RealOptProblem
```

```
class RealProblem2(RealOptProblem):

    def __init__(self):
        RealOptProblem.__init__(self)
        self.nvars=4

    def function_value(self, point):
        return point.vars[0] - point.vars[1] + (point.vars[2]-1.5)**2 + (point.vars[3]+2) ↩
            **4

problem = RealProblem2()
problem.main(sys.argv)
```

The last two lines of `RealProblem1.py` create a problem instance and then call the `main` method to parse the command-line arguments. This script has the following command-line syntax:

```
  RealProblem1.py <input-file> <output-file>
```

The first argument is the name of an XML input file, and the second argument is the name of an XML output file. The optimization problem class manages the parsing of the input and generation of the output file. For example, consider the following input file:

```
<ColinRequest>
  <Parameters>
    <Real size="4"> 0.1e-1 -0.1 1.1 -1.9</Real>
  </Parameters>
  <Requests>
    <FunctionValue/>
  </Requests>
</ColinRequest>
```

The `RealProblem1.py` script creates the following output file:

```
<?xml version="1.0" encoding="UTF-8"?>
 <ColinResponse>
  <FunctionValue>
   0.2701
  </FunctionValue>
 </ColinResponse>
```

19.2 Diving Deeper

The previous section provided an overview of the how the `pyomo.opt.blackbox` package supports the definition of optimization problems that are solved with black-box optimizers. In this section we provide more detail about how the Python problem class can be customized, as well as details about the XML file format used to communicate with Coliny optimizers. The Dakota User Manual <Dakota> provides documentation of the file format of the input and output files used with Dakota optimizers.

Table Table 19.1 summarizes the methods of the `OptProblem` class that a user is likely to either use or redefine when declaring a subclass. The `MixedIntOptProblem` class is a convenient base class for the problems solved by most black-box optimizers, and this class provides the definition of the `main`, `create_point` and `validate` methods. However, any of the remaining methods may need to be defined, depending on the problem.

Table 19.1: Methods in the `OptProblem` class.

Method	Description
*_init_*	The constructor, which may be redefined to specify problem properties.

Table 19.1: (continued)

main	Method that processes command-line options to create a results file from an input file.
create_point	Create an instances of the class that defines a point in the search domain.
function_value	Returns the value of the objective function.
function_values	Returns a list of objective function values.
gradient	Returns a list that represents the gradient vector at the given point.
hessian	Returns a Hessian matrix.
nonlinear_constraint_values	Returns a list of values for the constraint functions.
jacobian	Returns a Jacobian matrix.
validate	Returns `True` if the given point is feasible, and `False` otherwise.

The following detailed example illustrates the use of all of these methods in a simple application:

```
class RealProblem3(RealOptProblem):

    def __init__(self):
        RealOptProblem.__init__(self)
        self.nvars=4
        self.ncons=4
        self.response_types = [response_enum.FunctionValue,
                               response_enum.Gradient,
                               response_enum.Hessian,
                               response_enum.NonlinearConstraintValues,
                               response_enum.Jacobian]

    def function_value(self, point):
        return point.vars[0] - point.vars[1] + (point.vars[2]-1.5)**2 + (point.vars[3]+2) ↩
            **4

    def gradient(self, point):
        return [1, -1, 2*(point.vars[2]-1.5), 4*(point.vars[3]+2)**3]

    def hessian(self, point):
        H = []
        H.append( (2,2,2) )
        H.append( (3,3,12*(point.vars[3]+2)**2) )
        return H

    def nonlinear_constraint_values(self, point):
        C = []
        C.append( sum(point.vars) )
        C.append( sum(x**2 for x in point.vars) )
        return C

    def jacobian(self, point):
        J = []
        for j in range(self.nvars):
            J.append( (0,j,1) )
        for j in range(self.nvars):
            J.append( (1,j,2*point.vars[j]) )
        return J
```

The `response_types` attribute defined in the constructor specifies the type of information that this class can compute. For example, consider the following input XML file:

```
<ColinRequest>
  <Parameters>
    <Real size="4"> 0.1e-1 -0.1 1.1 -1.9</Real>
  </Parameters>
  <Requests>
    <FunctionValue/>
    <Gradient/>
    <Hessian/>
    <NonlinearConstraintValues/>
    <Jacobian/>
  </Requests>
</ColinRequest>
```

This input file requests that the class compute all of the response values, and thus the following output is generated:

```
<?xml version="1.0" encoding="UTF-8"?>
 <ColinResponse>
  <Gradient>
   1 -1 -0.79999999999999982 0.0040000000000000105
  </Gradient>
  <NonlinearConstraintValues>
   -0.8899999999999999 4.8300999999999998
  </NonlinearConstraintValues>
  <FunctionValue>
   0.2701
  </FunctionValue>
  <Hessian>
   (2, 2, 2) (3, 3, 0.12000000000000022)
  </Hessian>
  <Jacobian>
   (0, 0, 1) (0, 1, 1) (0, 2, 1) (0, 3, 1) (1, 0, 0.02) (1, 1, -0.20000000000000001) (1, 2, ↩
       2.2000000000000002) (1, 3, -3.7999999999999998)
  </Jacobian>
 </ColinResponse>
```

Note that the values for Jacobian and Hessian matrices are represented in a sparse manner. Currently, these are represented with a list of tuple values, though a sparse matrix representation might be supported in the future.

Chapter 20

Distributed Optimization with Pyro

Pyomo supports distributed computing via the Python "PYRO" package. PYRO stands for PYthon Remote Objects. Full documentation of PYRO is available from: http://pyro.sourceforge.net/.

The following describes a "quick-start" process for creating and using a client and multiple solvers on a single, presumably multi-core compute server. For example, an institution may have an 8-core workstation with numerous CPLEX licenses. With distributed solves under PYRO, Pyomo algorithms can take advantage of the full set of resources on a machine.

The following example assumes a unix/linux platform. The steps for Windows are qualitatively identical - the sole difference is that you can't (or at least we haven't figured out how to) put processes in the background on Windows. The work-around is simply (albeit painfully) to launch the various processes in distinct shells.

20.1 Step 1: Starting a Name Server

All PYRO objects communicate via a name server, which provides a well-defined point of contact through which distributed objects can interact. You can think of the name server as a phone directory.

To start the name server, type:

```
pyomo-ns
```

In general, we suggest that the output be redirected to a file, with the entire process being placed in the background:

```
pyomo-ns >& ns.out &
```

20.2 Step 2: Starting a Dispatch Server

With the name server up and running, the next step is to create a dispatch server. The function of the dispatch server is to route work from clients to servers - both of the latter will be established in the immediately following steps. We again assume the process is executed in the background, with the output redirected:

```
dispatch_srvr >& dispatch_srvr.out &
```

20.3 Step 3: Starting a MIP server

With the work dispatcher up, the next step is to create servers to do real work! Pyomo ships with a `pyro_mip_server` script, which launches a server capable of solving a single MIP at a time. This server can be invoked as follows:

```
pyro_mip_server >& pyro_mip_server1.out &
```

We can also create multiple instances of the `pyro_mip_server`, e.g., to take advantage of multiple solver licenses:

```
pyro_mip_server >& pyro_mip_server2.out &
```

With this configuration, the dispatch server "sees" two mip servers, and can route work to both.

20.4 Step 4: Running a Client

To take advantage of the distributed MIP servers, a Pyomo user only needs to change the type of the solver manager supplied to the various solver scripts.

For example, one can run pyomo as follows, considering the PySP example found in: `pyomo/examples/pysp/farmer`:

```
pyomo solve --solver=cplex --solver-manager=pyro farmer_lp.py farmer_lp.dat
```

This will execute the LP solve using one of the two mip servers established in Step 3, which might be useful if they are on remote servers.

To take advantage of parallelism, we can solve the farmer example using progressive hedging, as follows:

```
runph --solver=cplex --solver-manager=pyro --model-directory=models --instance-directory= ↩
    scenariodata
```

20.5 Moving from Multi-Core to Distributed Computation

Truly distributed computation, i.e., with the client and server components on different hosts, is only incrementally more difficult than what is outlined above. If multiple hosts are involved in the computation, the only real issue is making sure the various hosts can all find a common nameserver. After starting `pyomo-ns` on some host (presumably a server-class machine), the other components (`dispatch_srvr` and the `pyro_mip_server`) can be pointed to the nameserver by simply setting the environment variable PYRO_NS_HOSTNAME to the name (or IP address) of the host running the nameserver. The same process should be followed on the client prior to executing either `pyomo`, `runph`, or some other client solver script.

We have tested this on linux clusters with success. The only issues encountered involve overly aggressive firewalls on the host running the nameserver, which was easily corrected. In theory, Pyro should also work on Windows clusters, and linux-Window hybrid clusters via the same mechanism.

20.6 Cleaning Up After Yourself

It is important to remember that the name server, the dispatch server, and the mip server processes are persistent, and need to be terminated when a user has completed computational experiments. Actually, that is not entirely correct - the server processes can live forever, and continue to receive work. The only issue is when multiple users are attempting to use the same compute platform, are running their own servers, etc. While this may work, we have not tested it fully yet!

Bibliography

- [AIMMS] http://www.aimms.com/
- [AMPL] http://www.ampl.com
- [GAMS] http://www.gams.com
- [Coliny] Hart and Siirola. Coliny Technical Report. Sandia National Labs.
- [Dakota] Eldred et al. Dakota Technical Report. Sandia National Labs.
- [MPG] Mathematical Programming Glossary, http://glossary.computing.society.informs.org/, Ed. Allen Holder, 2012.
- [] Hart, W.E., Laird, C., Watson, J.-P., Woodruff, D.L., Pyomo – Optimization Modeling in Python, Springer, 2012.
- [PyomoJournal] William E. Hart, Jean-Paul Watson, David L. Woodruff, "Pyomo: modeling and solving mathematical programs in Python," Mathematical Programming Computation, Volume 3, Number 3, August 2011
- [BirgeLouveauxBook] Introduction to Stochastic Programming; J.R. Birge and F. Louveaux; Springer Series in Operations Research; New York: Springer, 1997.
- [Vielma_et_al] Vielma, J.P, Ahmed, S., Nemhauser, G., "Mixed-Integer Models for Non-separable Piecewise Linear Optimization: Unifying framework and Extensions," Operations Research 58, 2010. pp. 303-315.

Colophon

This book was created using `asciidoc` software.

The Sandia National Laboratories Laboratory-Directed Research and Development Program and the U.S. Department of Energy Office of Science Advanced Scientific Computing Research Program funded portions of this work.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND 2012-1572P.