Coopr User Manual:
Getting Started with the Pyomo Modeling Language

William E. Hart[1] Jean-Paul Watson[2] David L. Woodruff[3]

November 7, 2009

[1]Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; wehart@sandia.gov

[2]Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; jwatson@sandia.gov

[3]Graduate School of Management, University of California at Davis, Davis, CA 95616-8609; dlwoodruff@ucdavis.edu

Contents

Chapter 1

Getting Started with Pyomo

1.1 Introduction

The Python Optimization Modeling Objects (Pyomo) software package supports the definition and solution of optimization applications using the Python scripting language. Python is a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. Pyomo includes Python classes for sparse sets, parameters, and variables, which can be used to formulate algebraic expressions that define objectives and constraints. Thus, Pyomo can be used to concisely represent mixed-integer linear programming (MILP) models for large-scale, real-world problems that involve thousands of constraints and variables. Further, Pyomo includes a flexible framework for applying optimizers to analyze these models.

The design of Pyomo is motivated by a variety of factors that have impacted applications at Sandia National Laboratories. Sandia's discrete mathematics group has successfully used AMPL [1, 3] to model and solve large-scale integer programs for many years. This application experience has highlighted the value of Algebraic Modeling Languages (AMLs) for solving real-world applications, and AMLs are now an integral part of operations research solutions at Sandia.

Pyomo was developed to provide an alternative platform for developing math programming models that leverages Python's rich programming environment to facilitate the application and deployment of optimization capabilities. Pyomo provides a set of Python classes and functions that define a modeling capability that is similar to AML's like AMPL. Further, Pyomo leverages a flexible plugin framework to provide a highly extensible and flexible modeling framework. Pyomo is integrated into Coopr, a COmmon Optimization Python Repository. Coopr packages provide optimization components that can be applied to optimize Pyomo models in a flexible manner.

This chapter discusses how to install Coopr and verify that Pyomo can be run. The rest of this document introduces the user to Pyomo and describes the details of the Pyomo's modeling objects. This presentation is principally intended for Pyomo end-users. Readers may also find the following references useful when diving deeper into Coopr and Pyomo:

- W. E. Hart, J. Siirola, and J.-P. Watson, "Coopr User Manual: Customizing Coopr with Plugins", Sandia National Laboratories, 2009.
- W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Coopr User Manual: Pyomo Modeling Language and Extension Packages", Sandia National Laboratories, 2009.
- W. E. Hart, J.-P. Watson, and D. L. Woodruff, "PYthon Optimization Modeling Objects (Pyomo)", 2009, (in preparation).

1.2 Installing Coopr

There are several different ways that Coopr can be installed:

easy_install Coopr releases can be directly installed using the Python `easy_install` command.

source Coopr can be installed from source.

coopr_install The `coopr_install` command provides a one-step installation of Coopr and the Python packages that Coopr depends on.

The first two options are the techniques that Python developers typically used. The `easy_install` command is the *de facto* standard python installation technique. For example, the following command will download Coopr and the Python packages that it depends on, and install them in Python's site-packages directory:

```
easy_install Coopr
```

In most cases, end-users will want to use the `coopr_install` script to install Coopr and other packages that Coopr depends on. This is a Python script that creates a directory that contains a *virtual* Python installation, related Coor scripts, examples and related documentation. This installation does not require administrator privileges, and the user can view the Coopr documentation and examples in the installation directory.

The `coopr_install` script does not rely on non-standard Python packages, so it can be run as follows:

```
coopr_install coopr
```

On MS Windows, the `python` command needs to be run explicitly:

```
python coopr_install coopr
```

This creates the `coopr` directory, which has the following directory structure:

```
admin       Administrative data for maintaining this distribution
bin         Scripts and executables
```

```
doc         Coopr documentation and tutorials
examples    Coopr examples
lib         Python libraries and installed packages
include     Python header files
src         Python packages whose source files can be
            modified and used directly within this virtual Python
            installation.
Scripts     Python bin directory (used on MS Windows)
util        Coopr utility scripts (including coopr_install)
```

If the `bin` directory is put in user's `PATH` environment, then the `bin/python` command can be used to employ Coopr and associated packages without further configuration. Further, Coopr's Python scripts are installed in the `bin` directory such that they reference this virtual Python installation directly.

If `coopr_install` is executed with no installation directory, then the script will search the user's `PATH` environment for the `pyomo` command. If found, the path of this command will be used to identify the Coopr installation that is being updated or replaced. If not found, then a default installation path is used: `C:`
`coopr` on Windows and `./coopr` on Linux.

By default, `coopr_install` installs the latest release of Coopr. The current development trunk can be installed using the `--trunk` option:

```
coopr_install --trunk coopr
```

Also, Coopr has a stable branch, which is updated as major software revisions are finalized. This can be installed with the `--stable` option:

```
coopr_install --trunk coopr
```

Users can reinstall Coopr using the `--clear` option:

```
coopr_install --clear coopr
```

Note that this option is also needed to switch between the trunk, stable, and release installations, since that involves a reinstallation of Coopr. A Coopr installation can also be updated with the `--update` option:

```
coopr_install --update coopr
```

This updates Coopr to the latest release, or the latest revision of trunk and stable installations.

The `coopr_install` script installs a variety of Python packages that Coopr uses. This script also has options for using third-party Coopr extensions that are available on the Coopr Forum software repository **??**. The Coopr Forum repository facilitates community involvement in Coopr by allowing people to contribute code extensions and plugins without

going directly through the Coopr software repository. For example, the `coopr.plugins.neos` package provides a simple example of how Coopr can be extended with plugins to enable optimization on the NEOS optimization server [2]. This plugin package can be installed with Coopr using the `--forum-pkg` option:

```
coopr_install --forum-pkg=neos coopr
```

Multiple packages can be separated with a comma-separated list of package names.

The Python `setuptools` package is the *de facto* standard for deploying Python software. This package extends Python's `distutils` functionality. A key element of this extension is the `easy_install` command, which allows the installation of Python software from remote repositories. In particular, the Python Package Index (PyPI) provides a convenient repository for hosting Python packages. The `easy_install` command can easily upload and download packages from PyPI, thereby simplifying the distribution of Packages like Coopr, which depends on a variety of freely available packages.

Finally, here are some notes about `coopr_install`:

- This script installs packages by downloading files from the internet. If you are running this from within a firewall, you may need to set the `HTTP_PROXY` environment variable to a value like `http://<proxyhost>:<port>`.
- By default, the virtual Python installation used with Coopr exposes the packages that are installed with your Python installation. Occasionally, this can cause conflicts between different package version. The `--no-site-packages` option isolates the Coopr installation from the Python packages that have been installed with the Python interpreter.

Chapter 2

Introducing Pyomo

2.1 Pyomo Overview

Pyomo can be used to define abstract problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo can generate problem instances and apply optimization solvers with a fully expressive programming language. Python's clean syntax allows Pyomo to express mathematical concepts with a reasonably intuitive syntax. Further, Pyomo can be used within an interactive Python shell, thereby allowing a user to interactively interrogate Pyomo-based models. Thus, Pyomo has many of the advantages of both AML interfaces and modeling libraries.

2.1.1 A Simple Example

In this section we illustrate Pyomo's syntax and capabilities by demonstrating how a simple AMPL example can be replicated with Pyomo Python code. Consider the AMPL model, `prod.mod`:

```
set P;

param a {j in P};
param b;
param c {j in P};
param u {j in P};

var X {j in P};

maximize Total_Profit: sum {j in P} c[j] * X[j];

subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;

subject to Limit {j in P}: 0 <= X[j] <= u[j];
```

To translate this into Pyomo, the user must first import the Pyomo module and create a Pyomo **Model** object:

```
# Imports
from coopr.pyomo import *

# Create the model object
model = Model()
```

This import assumes that Pyomo is available on the users's Python path (see Python documentation for further details about the PYTHONPATH environment variable). Next, we create the sets and parameters that correspond to the data used in the AMPL model. This can be done very intuitively using the **Set** and **Param** classes.

```
# Sets
model.P = Set()

# Parameters
model.a = Param(model.P)
model.b = Param()
model.c = Param(model.P)
model.u = Param(model.P)
```

Note that parameter b is a scalar, while parameters a, c and u are arrays indexed by the set P.

Next, we define the decision variables in this model.

```
# Variables
model.X = Var(model.P)
```

Decision variables and model parameters are used to define the objectives and constraints in the model. Parameters define constants and the variables are the values that are optimized. Parameter values are typically defined by a data file that is processed by Pyomo.

Objectives and constraints are explicitly defined expressions in Pyomo. The **Objective** and **Constraint** classes require a **rule** option that specifies how these expressions are constructed. This is a function that takes one or more arguments: the first arguments are indices into a set that defines the set of objectives or constraints that are being defined, and the last argument is the model that is used to define the expression.

```
# Objective
def Objective_rule(model):
    return sum([model.c[j]*model.X[j] for j in model.P])
model.Total_Profit = Objective(rule=Objective_rule, sense=maximize)
```

```
# Time Constraint
def Time_rule(model):
    return summation(model.X, denom=model.a) < model.b
model.Time = Constraint(rule=Time_rule)

# Limit Constraint
def Limit_rule(j, model):
    return (0, model.X[j], model.u[j])
model.Limit = Constraint(model.P, rule=Limit_rule)
```

The rules used to construct these objects use standard Python functions. The **Objective_rule** function returns an algebraic expression that defines the objective; this expression is generated using Python's list comprehension syntax, which is used to create a list of terms that are added together with the **sum()** function. The **Time_rule** function returns a $<$ expression that defines an upper bound on the constraint body. The constraint body is created with Python's **summation()** function; in this example the summation is $\sum_i X_i/a_i$. The **Limit_rule** function illustrates another convention that is supported by Pyomo; a rule can return a tuple that defines the lower bound, body and upper bound for a constraint. The value 'None' can be returned for one of the limit values if a bound is not enforced.

Once an abstract model has been created, it can be printed as follows:

```
model.pprint()
```

This summarize the information in the Pyomo model, but it does not print out explicit expressions. This is due to the fact that an abstract model needs to be instanted with data to generate the model objectives and constraints:

```
instance = model.create("prod.dat")
instance.pprint()
```

Once a model instance has been constructed, an optimizer can be applied to it to find an optimal solution. For example, the PICO integer programming solver can be used within Pyomo as follows:

```
opt = solvers.SolverFactory("pico")
opt.keepFiles=True
results = opt.solve(instance)
```

This creates an optimizer object for the PICO executable, and it indicates that temporary files should be kept. The Pyomo model instance is optimized, and the optimizer returns an object that contains the solutions generated during optimization.

2.1.2 Putting It All Together

2.2 A Complete Pyomo Example

```
# Imports
from coopr.pyomo import *

# Create the model object
model = Model()

# Sets
model.P = Set()

# Parameters
model.a = Param(model.P)
model.b = Param()
model.c = Param(model.P)
model.u = Param(model.P)

# Variables
model.X = Var(model.P)

# Objective
def Objective_rule(model):
    return sum([model.c[j]*model.X[j] for j in model.P])
model.Total_Profit = Objective(rule=Objective_rule, sense=maximize)

# Time Constraint
def Time_rule(model):
    return summation(model.X, denom=model.a) < model.b
model.Time = Constraint(rule=Time_rule)

# Limit Constraint
def Limit_rule(j, model):
    return (0, model.X[j], model.u[j])
model.Limit = Constraint(model.P, rule=Limit_rule)
```

2.2.1 Pyomo Commandline Script

Appendix 2.2 provides a complete Python script for the model described in the previous section. Although this Python script can be executed directly, Coopr includes a `pyomo` script that can construct this model, apply an optimizer and summarize the results. For example, the following command line executes Pyomo using a data file in a format consistent with AMPL:

```
pyomo prod.py prod.dat
```

This script executes the following steps:

- create abstract model
- read data
- generate instance
- presolve
- apply solver
- load results into instance

The `pyomo` script has a variety of command line options to provide information about the optimization process. Options can control how debugging information is printed, including logging information generated by the optimizer and a summary of the model generated by Pyomo. Further, Pyomo can be configured to keep all intermediate files used during optimization, which can support debugging of the model construction process.

Chapter 3

Declaring Pyomo Models

This chapter and the next provide a reference for the Pyomo modeling language. This modeling language consists of a set of Python objects and utility functions that

3.1 Sets

A set is any collection of data that relates to a model. Pyomo set objects either contain concrete data, or they are "virtual" sets that do not contain data, but which support operations like set iteration and/or set membership validation. Several different classes can be used to define sets in Pyomo models:

- `Set`
  A generic set declaration class.
- `RangeSet`
  A set that describe a range of numbers.

3.1.1 Set Declarations

A simple instance of `Set` objects declares an unordered set of arbitrary objects:

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

A set array can also be specified by providing sets as options to the `Set` object. Multi-dimensional set arrays can be declared by simply including a list of sets as options to the `Set` object:

```
model.B = Set()
model.C = Set(model.A)
model.D = Set(model.A, model.B)
```

Set declarations can also use standard set operations to declare a set in a constructive fashion:

```
model.D = model.A | model.B
model.E = model.B & model.A
model.F = model.A - model.B
model.G = model.A ^ model.B
```

Also, set cross-products can be specified as `A*B`

```
model.H = model.A * model.B
```

Note that this is different from the following, which specifies that `Hsub` is a subset of this cross-product.

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

3.1.2 Set Initialization

By default, a set object refers to an abstract set in a model. However, a set can be initialized with data by using the `initialize` option, which is a function that accepts the set indices and model and returns the value of that set element:

```
def I_init(model):
    ans=[]
    for a in model.A:
      for b in model.B:
        ans.append( (a,b) )
    return ans
model.I = model.A*model.B
model.I.initialize = I_init
```

Note that the set `model.I` is not created when this set object is constructed. Instead, `I_init()` is called during the construction of a problem instance.

A set can also be explicitly constructed by add set elements:

```
model.J = Set()
model.J.add(1,4,9)
```

The `initialize` option can also be used to specify the values in a set. These default values may be overriden by later construction steps, or by data in an input file:

```
model.K = Set(initialize=[1,4,9])
model.K_2 = Set(initialize=[(1,4),(9,16)],dimen=2)
```

A set array can be constructed with the `initialize` option, which is a function that accepts the set indices and model and returns the set for that array index:

```
def P_init(i, j, model):
    return range(0,i*j)
model.P = Set(model.B,model.B)
model.P.initialize = P_init
```

The `initialize` option can also be used to specify the values in a set array. These default values are defined in a dictionary, which specifies how each array element is initialized:

```
R_init={}
R_init[2] = [1,3,5]
R_init[3] = [2,4,6]
R_init[4] = [3,5,7]
model.R = Set(model.B,initialize=R_init)
```

Note that a set array *cannot* be explicitly constructed by adding set elements to individual arrays. For example, the following is invalid:

```
model.Q = Set(model.B)
model.Q[2].add(4)
model.Q[4].add(16)
```

The reason is that the line

```
model.Q = Set(model.B)
```

declares set `Q` with an abstract index set `B`. However, `B` is not initialized until this model is instantiated with the `model.create()` call. We could, however, execute

```
model.Q[2].add(4)
model.Q[4].add(16)
```

after the execution of `model.create()`.

3.1.3 Data Validation

Validation of set data is supported in two different ways. First, a superset can be specified with the `within` option:

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

Validation of set data can also be performed with the `validate` option, which is a function that returns `True` if a data belongs in this set:

```
def M_validate(value,model):
    return value in model.A
```

```
model.M = Set(validate=M_validate)
```

Although the `within` option is convenient, it can force the creation of a temporary set. For example, consider the declaration

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

In this example, the cross-product of sets `A` and `B` is needed to validate the members of set `C`. Pyomo creates this set implicitly and uses it for validation. By contrast, a simple validation function could be used in this example, though with a less intuitive syntax:

```
def O_validate(value,model):
  return value[0] in model.A and value[1] in model.B
model.O = Set(validate=O_validate)
```

Validation of a set array is supported with the `within` option. The elements of all sets in the array must be in this set:

```
model.S = Set(model.B, within=model.A)
```

Validation of set arrays can also be performed with the `validate` option. This is applied to all sets in the array:

```
def T_validate(value,model):
    return value in model.A
model.T = Set(model.B, validate=M_validate)
```

3.1.4 Set Options

By default, sets are unordered. That is, the internal representation may place the set elements in any order. In some cases, we need to know the order in which set elements are declared. In such cases, we can declare a set to be ordered with an additional constructor option.

An ordered set can take an initialization function, using the `initialize` options, with an additional option that specifies the index into the ordered set. In this case, the function is called repeatedly to construct each element in the set:

```
def U_init(z, model):
    if z==5:
       return None
    if z==0:
       return 1
    else:
```

```
        return model.U[z-1]*(z+1)
model.U = Set(ordered=True, initialize=U_init)
```

This example can be generalized to array sets. Note that in this case we can use ordered sets to to index the array, thereby guaranteeing that data has been filled. The following example illustrates the use of the `RangeSet(a,b)` object, which generates an ordered set from `a` to `b` (inclusive).

```
def V_init(i, z, model):
    if z==5:
        return None
    if i==1:
        if z==0:
            return 1
        else:
            return (z+1)
    return model.V[i-1][z]+z
model.V = Set(RangeSet(1,4), initialize=V_init, ordered=True)
```

3.1.5 Class Attributes

Pyomo set objects have the following attributes:

- `name`
  The set name.
- validate
  A function that a user can specify to define set membership.
- ordered
  A boolean value that indicates whether this set is ordered.
- domain
  A super-set of this set, which is used to define set membership.
- dimen
  The "dimension" of the data in this set. Each set member is either a singleton, or a tuple with length 'dimen'.
- virtual
  A boolean value that indicates whether this set is virtual.
- doc
  A string describing this set.

3.1.6 Predefined Sets

A variety of virtual sets are declared in Pyomo, including:

- Any
  The set of all possible values.
- Reals
  The set of floating point values.
- PositiveReals
  The set of strictly positive floating point values.
- NonPositiveReals
  The set of non-positive floating point values.
- NegativeReals
  The set of strictly negative floating point values.
- NonNegativeReals
  The set of non-negative floating point values.
- PercentFraction
  The set of floating point values in the interval [0,1].
- Integers
  The set of integer values.
- PositiveIntegers
  The set of positive integer values.
- NonPositiveIntegers
  The set of non-positive integer values.
- NegativeIntegers
  The set of negative integer values.
- NonNegativeIntegers
  The set of non-negative integer values.
- Boolean
  The set of boolean values, which can be represented as False/True, 0/1, 'False'/'True' and 'F'/'T'.
- Binary
  The same as 'Boolean'.

3.2 Parameters

A parameter is a numerical value that is used to formulate constraints and objectives in a model. Pyomo parameters are managed with the `Param` class, which can denote a single, independent value, or an array of values.

3.2.1 Param Declarations

A simple instance of `Param` declares a single numerical value:

```
model.Z = Param()
```

A parameter array can also be specified by providing sets as options to the `Param` object. Multi-dimensional parameter arrays can be declared by simply including a list of sets as options to the `Param` object:

```
model.A = Set()
model.Y = Param(model.A)
model.B = Set()
model.X = Param(model.A, model.B)
```

3.2.2 Parameter Initialization

By default, a `Param` object refers to one or more abstract parameters in a model. However, a `Param` object can be initialized with data by using the `initialize` option, which is a function that accepts the parameter indices and model and returns the value of that parameter element:

```
def W_init(i,j,model):
    # Create the value of model.W[i,j]
    return i*j
model.W = Param(model.A, model.B, initialize=W_init)
```

Note that the parameter `model.W` is not created when this object is constructed. Instead, `W_init()` is called during the construction of a model instance.

The `initialize` option can also be used to specify the values in a parameter. These default values may be overriden by later construction steps, or by data in an input file:

```
V_init={}
V_init[1]=1
V_init[2]=2
V_init[3]=9
model.V = Param(model.A, initialize=V_init)
```

Note that parameter `V` is initialized with a dictionary, which maps tuples from parameter indices to parameter values. Simple, unindexed parameters can be initialized with a scalar value.

```
model.U = Param(initialize=9.9)
```

Pyomo assumes that parameter values are specified in a sparse manner. For example, the instance `Param(model.A,model.B)` declares a parameter indexed over sets `A` and `B`. However, not all of these values are necessarily declared in a model. The default value for all parameters not declared is zero. This default can be overriden with the `default` option.

The following example illustrates how a parameter can be declared where every parameter value is nonzero, but the parameter is stored with a sparse representation.

```
R_init={}
R_init[2,2]=1
R_init[2,4]=1
R_init[2,6]=1
R_init[2,8]=1
model.R = Param(model.A, model.B, default=99.0, initialize=R_init)
```

Note that the parameter default value can also be specified in an input file. See `data.dat` for an example.

Note that the explicit specification of a zero default changes Pyomo's behavior. For example, consider:

```
model.a = Param(model.A, default=0.0)
model.b = Param(model.A)
```

When `model.a[x]` is accessed and the index has not been explicitly initialized, the value zero is returned. This is true whether or not the parameter has been initialized with data. Thus, the specification of a default value makes the parameter seem to be densely initialized.

However, when `model.b[x]` is accessed and the index has not been initialized, an error occurs (and a Python exception is thrown). Since the user did not explicitly declare a default, Pyomo treats the reference to `model.b[x]` as an error.

3.2.3 Data Validation

Validation of parameter data is supported in two different ways. First, the domain of feasible parameter values can be specified with the `within` option:

```
model.T = Param(within=model.B)
```

Note that the default domain for parameters is `Reals`, the set of floating point values. Validation of parameter data can also be performed with the `validate` option, which is a

function that returns `True` if a parameter value is valid:

```
def S_validate(value,model):
    return value in model.A
model.S = Param(validate=S_validate)
```

3.2.4 Paramter Options

TBD

3.2.5 Class Attributes

3.3 Variable

A variable is a numerical value that is determined during optimization. Pyomo variables are managed with the `Var` class, which can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables can have initial values, and the value of variable can be retrieved and set.

3.3.1 Var Declarations

A simple instance of `Var` declares a single variable:

```
model.x = Var()
```

A variable array can also be specified by providing sets as options to the `Var` object. Multi-dimensional variable arrays can be declared by simply including a list of sets as options to the `Var` object:

```
model.A = Set()
model.Y = Var(model.A)
model.B = Set()
model.X = Var(model.A, model.B)
```

3.3.2 Variable Initialization

By default, a `Var` object refers to one or more variables in a model. Variable values are typically determined during optimization. However, variables can be initialized using the `initialize` option. This option can specify a numerical value used to initialize a variable or variable array:

```
model.x = Var(initialize=9)
```

```
model.x = Var(model.A, initialize={1:1, 2:4, 3:9})
model.x = Var(model.A, initialize=2)
```

Additionally, this option can use a function that accepts the variable indices and model and returns the value of that variable element:

```
def f(i,model):
    return 3*i
model.x = Var(model.A, initialize=f)
```

3.3.3 Variable Domain

The domain of a variable is specified with the `within` option:

```
model.x = Var(within=model.A)
```

This domain is used in various aspects of model construction. For example, binary variables define zero-one constraints in integer programs, as well as upper and lower bounds for linear programming relaxations.

3.3.4 Variable Options

Variable bounds can be explicitly specified with the `bounds` option:

```
model.x = Var(bounds=(0.0,1.0))
def f(i,model):
  return (model.A[i], model.B[i])
model.y = Var(bounds=f)
```

The `bounds` option can specify a 2-tuple with lower and upper values. Alternatively, it can specify a function that returns a 2-tuple for each variable index. Note that `None` can be used to specify that a bound is not enforced.

3.3.5 Working With Variables

Variable objects have a variety of helper functions and utility methods that facilitate the use of these objects. The `float` function can be used to coerce a `Var` object into a floating point value:

```
tmp = float(model.x)
tmp = float(model.x[i])
```

Similarly, the `value` function can be used to coerce a `Var` object into its natural numerical value:

```
tmp = value(model.x)
tmp = value(model.x[i])
```

Variable values can be set using the equality operator:

```
model.x = tmp
model.x[i] = tmp
```

Finally, the `len` function returns the number of variables in a variable array.

```
len(model.x)
```

3.3.6 Class Attributes

Methods

- `dim`
  Returns the number of dimensions of the variable index

- `keys`
  Returns the indices of the variable array

- `reset`
  Set the variable with the initial value. When a variable is constructed, its value is `None`

Options

- `value`
  The value of the variable.

- `initial`
  The initial value of the variable.

- `lb`
  The value of the variable lower bound.

- `ub`
  The value of the variable upper bound.

- `fixed`
  A boolean value that indicates whether this variable is fixed during optimization.

3.4 Objectives

An objective... variable is a numerical value that is determined during optimization. Pyomo variables are managed with the `Var` class, which can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables can have initial values, and the value of variable can be retrieved and set.

3.4.1 Var Declarations

A simple instance of `Var` declares a single variable:

```
model.x = Var()
```

A variable array can also be specified by providing sets as options to the `Var` object. Multi-dimensional variable arrays can be declared by simply including a list of sets as options to the `Var` object:

```
model.A = Set()
model.Y = Var(model.A)
model.B = Set()
model.X = Var(model.A, model.B)
```

3.4.2 Variable Initialization

By default, a `Var` object refers to one or more abstract variables in a model. However, a `Var` object can be initialized with data by using the `initialize` option, which is a function that accepts the variable indices and model and returns the value of that variable element:

```
def f(i,model):
    return 3*i
model.x = Var(model.A,initialize=f)
```

Additionally, the `initialize` option can specify a numerical value used to initialize a variable or variable array:

```
model.x = Var(initialize=9)
model.x = Var(model.A,initialize={1:1, 2:4, 3:9})
model.x = Var(model.A,initialize=2)
```

3.4.3 Data Validation

Validation of variable data is supported in two different ways. First, the domain of feasible variable values can be specified with the `within` option:

```
model.x = Var(within=model.A)
```

Note that the default domain for variables is `Reals`, the set of floating point values. Validation of variable data can also be performed with the `validate` option, which is a function that returns `True` if a variable value is valid:

```
def S_validate(value,model):
    return value in model.A
model.S = Var(validate=S_validate)
```

3.4.4 Variable Options

The option `bounds` specifies upper and lower bounds for variables. Simple bounds can be specified, or a function that defines bounds for different variables.

```
model.x = Var(bounds=(0.0,1.0))
def f(i,model):
  return (model.x_low[i], model._x_high[i])
model.x = Var(bounds=f)
```

3.4.5 Class Attributes

3.5 Constraints

A constraint is a numerical value that is determined during optimization. Pyomo variables are managed with the `Var` class, which can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables can have initial values, and the value of variable can be retrieved and set.

3.5.1 Var Declarations

A simple instance of `Var` declares a single variable:

```
model.x = Var()
```

A variable array can also be specified by providing sets as options to the `Var` object. Multi-dimensional variable arrays can be declared by simply including a list of sets as options to the `Var` object:

```
model.A = Set()
model.Y = Var(model.A)
model.B = Set()
```

```
model.X = Var(model.A, model.B)
```

3.5.2 Variable Initialization

By default, a `Var` object refers to one or more abstract variables in a model. However, a `Var` object can be initialized with data by using the `initialize` option, which is a function that accepts the variable indices and model and returns the value of that variable element:

```
def f(i,model):
    return 3*i
model.x = Var(model.A,initialize=f)
```

Additionally, the `initialize` option can specify a numerical value used to initialize a variable or variable array:

```
model.x = Var(initialize=9)
model.x = Var(model.A,initialize={1:1, 2:4, 3:9})
model.x = Var(model.A,initialize=2)
```

3.5.3 Data Validation

Validation of variable data is supported in two different ways. First, the domain of feasible variable values can be specified with the `within` option:

```
model.x = Var(within=model.A)
```

Note that the default domain for variables is `Reals`, the set of floating point values. Validation of variable data can also be performed with the `validate` option, which is a function that returns `True` if a variable value is valid:

```
def S_validate(value,model):
    return value in model.A
model.S = Var(validate=S_validate)
```

3.5.4 Variable Options

The option `bounds` specifies upper and lower bounds for variables. Simple bounds can be specified, or a function that defines bounds for different variables.

```
model.x = Var(bounds=(0.0,1.0))
def f(i,model):
  return (model.x_low[i], model._x_high[i])
```

```
model.x = Var(bounds=f)
```

3.5.5 Class Attributes

3.6 Miscellaneous Model Components

A variable is a numerical value that is determined during optimization. Pyomo variables are managed with the `Var` class, which can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables can have initial values, and the value of variable can be retrieved and set.

3.6.1 Var Declarations

A simple instance of `Var` declares a single variable:

```
model.x = Var()
```

A variable array can also be specified by providing sets as options to the `Var` object. Multi-dimensional variable arrays can be declared by simply including a list of sets as options to the `Var` object:

```
model.A = Set()
model.Y = Var(model.A)
model.B = Set()
model.X = Var(model.A, model.B)
```

3.6.2 Variable Initialization

By default, a `Var` object refers to one or more abstract variables in a model. However, a `Var` object can be initialized with data by using the `initialize` option, which is a function that accepts the variable indices and model and returns the value of that variable element:

```
def f(i,model):
    return 3*i
model.x = Var(model.A,initialize=f)
```

Additionally, the `initialize` option can specify a numerical value used to initialize a variable or variable array:

```
model.x = Var(initialize=9)
model.x = Var(model.A,initialize={1:1, 2:4, 3:9})
model.x = Var(model.A,initialize=2)
```

3.6.3 Data Validation

Validation of variable data is supported in two different ways. First, the domain of feasible variable values can be specified with the `within` option:

```
model.x = Var(within=model.A)
```

Note that the default domain for variables is `Reals`, the set of floating point values. Validation of variable data can also be performed with the `validate` option, which is a function that returns `True` if a variable value is valid:

```
def S_validate(value,model):
    return value in model.A
model.S = Var(validate=S_validate)
```

3.6.4 Variable Options

The option `bounds` specifies upper and lower bounds for variables. Simple bounds can be specified, or a function that defines bounds for different variables.

```
model.x = Var(bounds=(0.0,1.0))
def f(i,model):
  return (model.x_low[i], model._x_high[i])
model.x = Var(bounds=f)
```

3.6.5 Class Attributes

Chapter 4

Loading Data into Pyomo Models

4.1 AMPL Data Files

TODO

4.2 Tables

TODO

4.3 Excel Spreadsheets

TODO

Acknowledgements

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Bibliography

[1] *AMPL home page*. `http://www.ampl.com/`, 2008.

[2] E. D. Dolan, R. Fourer, J.-P. Goux, T. S. Munson, and J. Sarich, *Kestrel: An interface from optimization modeling systems to the NEOS server*, INFORMS Journal on Computing, 20 (2008), pp. 525–538.

[3] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming, 2nd Ed.*, Brooks/Cole–Thomson Learning, Pacific Grove, CA, 2003.