Coopr User Manual: Getting Started with the Pyomo Modeling Language

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

September 18, 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

Introduction

This is a preliminary draft at a Pyomo Reference manual. There are some big picture issues that need to be addressed:

- What is the scope of this document? Should it include Coopr Opt? How about PySP?
- Should this include all aspects of Coopr?
- What specific chapters should we include ... even if we stay focused on 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 [2, 7] 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 is not intended to perform modeling *better* than existing tools. Instead, it supports a different modeling approach for which the software is designed for flexibility, extensibility, portability, and maintainability.

TODO: Review our goals? 1) Fully embedded modeling language, 2) Open source, 3) generic solvers, 4) extensibility. We review our goals in detail in the next section, so it seems funny to summarize them here. NOTE: we don't mention generic solvers later. Do we really have space to include that in this paper???

TODO: mention Open Source here? The point would be to say that the goal of Pyomo is not just to provide a free framework, but instead provide a framework that can be customized and extended. I'm not sure how to blend that theme with the previous paragraph

TODO: introduce Coopr here

Pyomo is integrated into Coopr, a COmmon Optimization Python Repository. The Coopr Opt package supports the execution of models developed with Pyomo using standard MILP solvers.

1.2 Design Goals and Requirements

The following sections describe the design goals and requirements that have guided the development of Pyomo. The design of Pyomo has been driven by a two different types of projects at Sandia. First, Pyomo has been used by research projects that need a flexible framework for customizing the formulation and evaluation of math programming models. Second, projects with external users often require that math programming modeling techniques be deployed without commercial licenses.

1.2.1 Open Source

A key goal of Pyomo is to provide an open-source math programming modeling capability. Although open-source optimization solvers are widely available in packages like COIN-OR, surprisingly few open-source tools have been developed to model optimization applications. An open-source capability for Pyomo is motivated by several factors:

- **Transparency and Reliability**: When managed well, open-source projects facilitate transparency in the software design and implementation. Since any developer can study and modify the software, bugs and performance limitations can be identified and resolved by a wide range of developers with diverse software experience. Consequently, there is growing evidence that managing software as open-source can improve its reliability.

- **Flexible Licensing**: A variety of significant operations research applications at Sandia have required the use of a modeling tool with a non-commercial license. There have been many different reasons for this requirement, including the need to support open-source analysis tools, limitations for software deployment on classified computers, and licensing policies for commercial partners (e.g. that are motivated by minimizing the costs of deploying an application model internally within a large company). The Coopr software, which contains Pyomo, is licensed under the BSD.

Although the use of an open-source model is not a panacea; ensuring high reliability of the software requires careful software management and a commited developer community. However, there is increasing recognition that open source software provides many advantages beyond simple cost savings [4], including supporting open standards and avoiding being locked in to a single vendor.

1.2.2 Customizable Capability

A key limitation of commercial modeling tools is the ability to customize the modeling or optimization process. Pyomo's open-source project model allows a diverse range of developers to prototype new capabilities. Thus, developers can customize the software for specific applications, and they can prototype capabilites that are integrated into future

More generally, Pyomo is designed to support a "stone soup" development model where each developer "scratches their own itch". A key element of this design is the plugin framework that Pyomo uses to integrate components like optimizers, optimizater managers, and model format conversions. This framework manages the registration of components, and it automates the interaction of these components through well-defined interfaces. Thus, users can customize Pyomo in a modular manner without risk of destabilizing core functionality.

1.2.3 Solver Integration

Modeling tools can be roughly categorized into two classes based on how they integrate with optimization solvers: *tightly coupled* modeling tools directly link in optimization solver libraries (including dynamic linking), and *loosely coupled* modeling tools apply external optimization executables (e.g. through system calls). Of course, these options are not exclusive, and a goal of Pyomo is to support both types of solver interfaces.

This design goal has led to a distinction in Pyomo between model formulation and optimization execution. Pyomo uses a high level programming language to formulate a problem that can be solved by optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations.

1.2.4 Abstract Models

A requirement of Pyomo's design is that it support the definition of abstract models in a manner similar to the AMPL. AMPL separates the declaration of a model from the data that generates a model instance. This is supports an extremely flexible modeling capability, which has been leveraged extensively in applications at Sandia.

To mimic this capability, Pyomo uses a symbolic representation of data, variables, constraints, etc. Model instances are then generated from external data sets using construction routines that are provided by the user when defining sets, parameters, etc. Further, Pyomo is designed to use data sets in the AMPL format to facilitate translation of models between AMPL and Pyomo.

1.2.5 Flexible Modeling Language

Another goal of Pyomo is to directly use a modern programming language to support the definition of math programming models. In this manner, Pyomo is similar to tools like FlopC++ [6] and OptimJ [17], which support modeling in C++ and Java respectively. The use of an existing programming language has several advantages:

- **Extensibility and Robustness**: A well-used modern programming language provides a robust foundation for developing and applying models, because the language has been well-tested in a wide variety of contexts. Further, extensions typically do not require changes to the language but instead involve additional classes and modeling routines that can be used in the modeling process. Thus, support of the modeling language is not a long-term factor when managing the software.

- **Documentation**: Modern programming languages are typically well-documented, and there is often a large on-line community to provide feedback to new users.

- **Standard Libraries**: Languages like Java and Python have a rich set of libraries for tackling just about every programming task. For example, standard libraries can support capabilities like data integration (e.g. working with spreadsheets), thereby avoiding the need to directly support this in a modeling tool.

An additional aspect of general-purpose programming languages is that they can support modern language features, like classes and first-class functions, that can be critical when defining complex models.

Pyomo is implemented in Python, a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. When compared with AMLs like AMPL, Pyomo has a more verbose and complex syntax. Thus, a key issue with this approach concerns the target user community and their level of comfort with standard programming concepts. Our examples in this paper compare and contrast AMPL and Pyomo models, which illustrate this trade-off.

1.2.6 Portability

A requirement of Pyomo's design is that it work on a diverse range of compute platforms. In particular, working well on both MS Windows and Linux platforms is a key requirement for many Sandia applications. The main impact of this requirement has been to limit the choice of programming languages. For example, the .Net languages were not considered for the design of Pyomo due to portability considerations.

1.3 Why Python?

Pyomo has been developed in Python for a variety of reasons. First, Python meets the criteria outlined in the previous section:

- **Open Source License:** Python is freely available, and its liberal open source license lets you modify and distribute a Python-based application with few restrictions.
- **Features:** Python has a rich set of datatypes, support for object oriented programming, namespaces, exceptions, and dynamic loading.
- **Support and Stability:** Python is highly stable, and it is well supported through newsgroups and special interest groups.
- **Documentation:** Users can learn about Python from extensive online documentation, and a number of excellent books that are commonly available.
- **Standard Library:** Python includes a large number of useful modules.
- **Extendability and Customization:** Python has a simple model for loading Python code developed by a user. Additionally, compiled code packages that optimize computational kernels can be easily used. Python includes support for shared libraries and dynamic loading, so new capabilities can be dynamically integrated into Python applications.
- **Portability:** Python is available on a wide range of compute platforms, so portability is typically not a limitation for Python-based applications.

Several other popular programming languages were also considered for Pyomo. However, in most cases Python appears to have distinct advantages:

- **.Net:** As mentioned earlier, the .Net languages are not portable to Linux platforms, and thus they were not suitable for Pyomo.
- **Ruby:** At the moment, Python and Ruby appear to be the two most widely recommended scripting languages that are portable to Linux platforms, and comparisons suggest that their core functionality is similar. Our preference for Python is largely based on the fact that it has a nice syntax that does not require users to type weird symbols (e.g. $, %, @). Thus, we expect this will be a more natural language for expressing math programming models.
- **Java:** Java has a lot of the same strengths as Python, and it is arguably as good a choice for Pyomo. However, Python has a powerful interactive interpreter that allows realtime code development and encourages experimentation with Python software. Thus, users can work interactively with Pyomo models to become familiar with these objects and to diagnose bugs.
- **C++:** Models formulated with the FlopC++ [6] package are similar to models developed with Pyomo. They are be specified in a declarative style using classes to represent model components (e.g. sets, variables and constraints). However, C++ requires explicit compilation to execute code, and it does not support an interactive interpreter. Thus, we believe that Python will provide a more flexible language for users.

We also considered developing a domain-specific AML. Domain-specific AMLs have can support a concise, expressive syntax, with a clear semantic interpretation. However, it is difficult to develop and maintain an AML. For example, there is extensive documentation on Python and other standard programming languages. By comparison, AMLs for math programming are sparsely documented. Additionally, it is a significant commitment to develop an AML that provides the full suite of capabilities that are available in modern programming languages (e.g. standard libraries, and interoperability with different programming languages).

Finally, we note that run-time performance was not a key factor in our decision to use Python. Recent emperical comparisons suggest that scripting languages offer reasonable alternatives to languages like C and C++, even for tasks that must handle fair amounts of computation and data [18]. Further, there is evidence that dynamically typed languages like python allow users to be more productive than with statically typed languages like C++ and Java [23, 20]. It is widely acknowledged that Python's dynamic typing and compact, concise syntax makes software development quick and easy. Thus, it is not surprising that Python is widely used in the scientific community [14]. Large Python projects like SciPy [10] and SAGE [21] strongly leverage a diverse set of Python packages to perform complex numerical calculations.

1.4 Background

A variety of different strategies have been developed to facilitate the formulation and solution of complex optimization models. For restricted problem domains, optimizers can be directly interfaced with application modeling tools. For example, modern spreadsheets like Excel integrate optimizers that can be applied to linear programming and simple nonlinear programming problems in a natural way.

Algebraic Modeling Languages (AMLs) are alternative approach that allows applications to be interfaced with optimizers that can exploit problem structure. AMLs are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [11]. AMLs like AIMMS [1], AMPL [2, 7] and GAMS [8] have programming languages with an intuitive mathematical syntax that supports concepts like sparse sets, indices, and algebraic expressions. AMLs provide a mechanism for defining variables and generating constraints with a concise mathematical representation, which is essential for large-scale, real-world problems that involve thousands of constraints and variables.

Standard programming languages can also be used to formulate optimization models when used in conjunction with a software library that uses object-oriented design to support mathematical concepts. Although these modeling libraries sacrifice some of the intuitive mathematical syntax of an AML, they allow the user to leverage the greater flexibility of standard programming languages. For example, modeling tools like FlopC++ [6], OPL [16] and OptimJ [17] can be used to formulate and solve optimization models.

A related strategy is to use a high-level programming language to formulate optimization models that are solved with optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations. This approach is increasingly common in scientific computing tools, and the Matlab TOMLAB Optimization Environment [22] is probably the most mature optimization software using this approach. However, Python has been used to implement a variety of optimization packages that use this approach:

- **APLEpy:** A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [3, 12].
- **CVXOPT:** A package for convex optimization [5].
- **PuLP:** A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [19].
- **POAMS:** A modeling tool for linear and mixed-integer linear programs that defines Python objects for abstract sets, constraints, objectives, decision variables, and solver interfaces.
- **OpenOpt:** A numerical optimization framework that is closely coupled with the SciPy scientific Python package [15].
- **NLPy:** An optimization framework that leverages AMPL to create problem instances, which can then be processed in Python [13].

Pyomo is similar to APLEpy, PuLP and POAMS. All of these packages define Python objects that can be used to express models. POAMS and Pyomo support a clear distinction between abstract models and problem instances. This design has several advantages, which were summarized by Fourer and Gay [7] when presenting AMPL:

- The statement of the symbolic model can be made compact and understandable,
- The independent specification of a symbolic model facilitates the specification of the validity of the associated data,
- Data from different sources can be used with the symbolic model, depending on the computing environment,

The main high-level feature that distinguishes Pyomo from POAMS is Pyomo's support for an instance construction process that is automated by object properties. This is akin to the capabilities of AML's like AMPL and GAMS, and it provides a standardized technique for constructing model instances. Hart [9] provides Python examples that illustrate the differences between PuLP, POAMS and Pyomo.

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] *AIMMS home page.* `http://www.aimms.com`, 2008.

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

[3] *APLEpy: An open source algebraic programming language extension for python.* `http://aplepy.sourceforge.net/`, 2005.

[4] F. Consulting, *Open source softwares expanding role in the enterprise.* `http://www.unisys.com/eprise/main/admin/corporate/doc/Forrester_research-open_source_buying_behaviors.pdf`, 2007.

[5] *CVXOPT home page.* `http://abel.ee.ucla.edu/cvxopt`, 2008.

[6] *FLOPC++ home page.* `https://projects.coin-or.org/FlopC++`, 2008.

[7] 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.

[8] *GAMS home page.* `http://www.gams.com`, 2008.

[9] W. E. Hart, *Python Optimization Modeling Objects (Pyomo)*, in Operations Research and Cyber-Infrastructure, J. W. Chinneck, B. Kristjansson, and M. J. Saltzman, eds., 2009, pp. 3–+.

[10] E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–.

[11] J. Kallrath, *Modeling Languages in Mathematical Optimization*, Kluwer Academic Publishers, 2004.

[12] S. Karabuk and F. H. Grant, *A common medium for programming operations-research models*, IEEE Software, (2007), pp. 39–47.

[13] *NLPy home page.* `http://nlpy.sourceforge.net/`, 2008.

[14] T. E. Oliphant, *Python for scientific computing*, Computing in Science and Engineering, (2007), pp. 10–20.

[15] *OpenOpt home page.* `http://scipy.org/scipy/scikits/wiki/OpenOpt`, 2008.

[16] *OPL home page.* `http://www.ilog.com/products/oplstudio`, 2008.

[17] *Ateji home page.* `http://www.ateji.com`, 2008.

[18] L. Prechelt, *An empirical comparison of seven programming languages*, Computer, 33 (2000), pp. 23–29.

[19] *PuLP: A python linear programming modeler.* `http://130.216.209.237/engsci392/pulp/FrontPage`, 2008.

[20] *Python & java: A side-by-side comparison.* `http://www.ferg.org/projects/python_java_side-by-side.html`, 2008.

[21] W. Stein, *Sage: Open Source Mathematical Software (Version 2.10.2)*, The Sage Group, 2008. `http://www.sagemath.org`.

[22] *TOMLAB optimization environment.* `http://www.tomopt.com/tomlab`, 2008.

[23] L. Tratt, *Dynamically typed languages*, Advances in Computers, 77 (2009), pp. 149–184.