Python Optimization Modeling Objects (Pyomo)

Nicolas L. Benavides* Robert D. Carr† William E. Hart‡

November 11, 2007

Abstract

We describe the Python Optimization Modeling Objects (Pyomo) package. Pyomo is a Python package that can be used to define abstract problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo provides a capability that is commonly associated with algebraic modeling languages like AMPL and GAMS. We introduce Pyomo by contrasting it with the capabilities of AMPL.

1 Introduction

Algebraic Modeling Languages (AMLs) are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [9]. AMLs like AIMMS [1], AMPL [2, 8] and GAMS [5] 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 almost essential for real-world problems that can involve thousands of constraints and variables.

An alternative strategy for modeling mathematical problems is to use a standard programming language in conjunction with a software library that uses object-oriented design to support similar mathematical concepts. Although these modeling libraries sacrifice the intuitive mathematical syntax of an AML, they allow the user to leverage the greater flexibility of standard programming languages. For example, modeling libraries like FLOPC++ [4], OPL [6] enable the solution of large, complex problems within a user-defined application.

This paper describes Pyomo, the Python Optimization Modeling Objects (Pyomo) package. Pyomo is a Python package that can be used to define abstract problems, create concrete problem instances, and solve these instances with standard solvers. Like other modeling libraries, Pyomo can generate problem instances and apply optimization solvers with a fully expressive programming language. Further, Python is a noncommercial language with a very large user community, which will ensure robust support for this language on a wide range of compute platforms.

Python is a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. 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.

Pyomo makes a clear distinction between the abstract specification of a model, generation of model instances, and the solution of model instances. Abstract models are a key element of AML's like AMPL, and this capability clearly distinguishes Pyomo from other Python modeling libraries like CVXOpt [3] and PuLP [7]. Pyomo models can be solved with either Python optimizers, or with externally defined solvers

*Santa Clara University, NBenavides@scu.edu
†Sandia National Laboratories, rdcarr@sandia.gov
‡Sandia National Laboratories, wehart@sandia.gov

(e.g. GLPK, CPLEX and CBC). Further, Python can integrate extension modules in low level languages like C or C++ to directly leverage fast solver libraries, and wrapped modules can be used within Python exactly like native Python code.

Section 2 illustrates how Pyomo would be used to model a simple application. We compare and contrast the Pyomo formulation with a formulation developed in the widely used AMPL modeling language. Section 3 describes the Pyomo classes that are used to define model components.

2 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 basic AMPL program 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:

```
#
# Import Pyomo
#
from pyomo import *

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

This import assumes that Pyomo is available on the users's Python path (see Python documentation for PYTHONPATH for further details). 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.

```
model.P = Set()

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

Note that parameter b is a scalar, while parameters a, c and u are arrays indexed by the set P. Pyomo also defines the **ProductSet** class, which can be defined in a similar manner.

Next, we define the decision variables in this model.

```
def X_bounds(j, model):
    return (0,model.u[j])
model.X = Var(index=model.P, bounds=X_bounds)
```

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.

```
def Objective_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + model.c[j] * model.X[j]
    return ans
model.profit = Objective(rule=Objective_rule)

def Time_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + (1.0/model.a[j]) * model.X[j]
    return ans < model.b
model.Time = Constraint(rule=Time_rule)
```

The rules used to construct these objects use standard Python functions. Finally, note that the **Time_rule** function includes the use of $<$ and $>$ operators on the expression. These operators are used to define upper and lower bounds on the constraints.

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

```
print ''ABSTRACT MODEL''
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'')

print ''MODEL INSTANCE''
instance.pprint()
```

Appendix A shows the final Python code for this example.

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.PICO(path="/home/wehart/bin/PICO",keepFiles=True)
solutions = opt.solve(instance)
```

This creates an optimizer object for the PICO executable defined in a given path, and it indicates that temporary files should be kept. The Pyomo model is handed to this optimizer, which returns the final solutions generated by the optimizer.

3 Documentation of Pyomo Objects

In this section we provide more detail on the definitions of Pyomo classes that are used to define models.

3.1 Sets

The **Set()** class is used to index other objects (e.g. **Param** and **Var**). This class has the same look-and-feel as a **sets.Set** class, but it can be used to define an abstract set. This class contains a concrete set, which can be initialized by the **load()** method, or directly.

Constructor arguments:

- within - A set that defines the type of values that can be contained in this set
- default - Default set members, which may be overriden when setting up this set
- rule - A rule for setting up this set with existing model data. This has the functional form: f: pyomo.Model $->$ pyomo.Set
- restriction - Define a rule for restricting membership in a set. This has the functional form: f: data $->$ bool and returns true if the data belongs in the set

3.2 Product Sets

The **ProductSet()** class represents the cross product of other sets.

Constructor arguments:

- default - Default set members, which may be overriden when setting up this set
- rule - A rule for setting up this set with existing model data. This has the functional form: f: pyomo.Model $->$ pyomo.Set
- restriction - Define a rule for restricting membership in a set. This has the functional form: f: data $->$ bool and returns true if the data belongs in the set

In the following AMPL code, the **rate** paramater's index set is the cross product of two sets:

```
set PROD;
set STAGE;

param rate {PROD,STAGE};
```

In Pyomo, the cross product is created with the **ProductSet** class, and the result of this is used to index other Pyomo objects:

```
model.PROD = Set()
model.STAGE = Set()

model.setprod = ProductSet( (model.PROD, model.STAGE) )
model.rate = Param(index=model.setprod)
steel4mod.rate > 0
```

3.3 Parameters

The **Param()** class defines constant values in a model, and a parameter object may be defined over an index.

Constructor arguments :

- index - The index set that defines the distinct parameters. By default, this is None, indicating that there is a single parameter.
- domain - A set that defines the type of values that each parameter must be.
- validate - A rule for validating this parameter with respect to data that exists in the model
- default - A set that defines default values for this parameter
- rule - A rule for setting up this parameter with existing model data

3.4 Variables

The **Var()** class defines a numeric variable, which may be defined over an index.

Constructor arguments:

- index - The index set that defines the distinct variables. By default, this is None, indicating that there is a single variable.
- domain - A set that defines the type of values that each parameter must be.
- default - A set that defines default values for this variable
- bounds - A function that defines bound constraints for this variable

Simple bound constraints on variables can be specified with the **bounds** rule:

```
model.P     = Set()

model.x_lb = Param(index=model.P)
model.x_ub = Param(index=model.P)

def x_bounds(i, model):
    return (model.x_lb[i], model.x_ub[i])
model.x = Var(index=model.P, rule=x_bounds)
```

3.5 Objectives

The **Objective()** class defines an objective expression.

Constructor arguments:

- rule - A rule for constructing this objective with existing model data.
- sense - Used to define wether this objective should be minimized or maximized (minimization is the default).

3.6 Constraints

The **Constraint()** class defines an expression whose value is constrained in the model.

Constructor arguments:

- rule - A rule for constructing this constraint with existing model data.
- index - Defines a set of constraints over an index.

Note that the **rule** option generally needs to include a definition of the bounds on a constraint. A constraint must have either an upper or lower bound, and it may have both. For example:

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

model.x = Var(index=model.Q)

def c_rule(i, model):
    ans = 0
    for q in model.Q:
      ans = ans + model.x[q]
    ans = ans > 0
    return ans < 1
model.c = Constraint(index=model.P, rule=c_rule)
```

The last two lines in the **c_rule** function define upper and lower bound values for the **c** constraint. Note that this is a non-standard use of the $<$ and $>$ operators; these operators return an expression rather than a boolean value.

3.7 Models

The **Model()** class defines a mixed-integer model that can be optimized by a user. This class takes no arguments, but it is a container for instances of the other Pyomo objects created by the user. For example, consider the statement:

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

This statement registers the variable x in the model, and assigns it the name "x".

4 Conclusions

Pyomo has many of the features of abstract modeling languages and optimization modeling libraries, but the following features of Pyomo are noteworthy:

- Pyomo supports the ability to define abstract problems from which problem instances can be generated. Further, Pyomo can generate multiple instances, which can be analyzed simultaneously in separate Python class objects.
- Pyomo is based on a powerful, commonly available open-source language. Thus, there are no licensing limitations with the use of Pyomo, and the set of Pyomo objects can be customized for an application in ways that are not possible with commercial AMLs and modeling libraries.
- Python has a clean syntax, so Pyomo modeling objects can be used in an intuitive manner.

- Pyomo models can leverage Python's programming language to define complex data structures and standard programming constructs like classes and functions. Further, Python can be naturally linked with external libraries for high-performance kernels.
- Pyomo can integrate optimization solvers in an extensible manner. Optimizers can be defined within Python itself, and external optimizers can be launched using file I/O to communicate with Python.[1]

Pyomo is probably most similar to the FLOPC++ modeling library. FLOPC++ is writen in C++, and it has many of the same objects as are used in Pyomo. While FLOPC++ enables models to be embedded in compiled application codes, Pyomo enables the rapid prototyping of models in a scripting language. Thus, these capabilities seem quite complementary.

The current implementation of Pyomo has been validated on a small set of simple models. In the future, more extensive validation of Pyomo is needed to ensure that it can express a wide range of complex problems. This includes the integration of hooks for other types of optimizers, like the nonlinear optimizers in Dakota. Further, the performance of Pyomo needs to be analyzed to ensure that it can effectively generate large-scale optimization models. Finally, this document needs to be extended to include examples that illustrate how Pyomo can leverage Python to develop complex models more naturally than AMLs like AMPL and GAMS.

This document describes the initial prototype of Pyomo. Once this code has stabilized, we plan to integrate Pyomo into the COIN-OR optimization software repository to encourage its use within the academic and business communities.

Acknowledgements

Thanks to Jon Berry and Cindy Phillips for their critical feedback on the design of Pyomo. 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.

References

[1] *AIMMS home page*. `http://www.aimms.com`.

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

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

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

[5] *GAMS home page*. `http://www.gams.com`.

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

[7] *Pulp: A python linear programming modeler*. `http://www.jeannot.org/~js/code/index.en.html`.

[8] 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.

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

[1] For example, this is similar to the manner in which AMPL launches optimizers.

A Complete Python Implementation of the Simple Model

```
# Imports
from pyomo import *

# Setup the model
model = Model()

model.P = Set()

model.a = Param(index=model.P)
model.b = Param()
model.c = Param(index=model.P)
model.u = Param(index=model.P)

def X_bounds(j, model):
    return (0,model.u[j])
model.X = Var(index=model.P, bounds=X_bounds)

def Objective_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + model.c[j] * model.X[j]
    return ans
model.profit = Objective(rule=Objective_rule)

def Time_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + (1.0/model.a[j]) * model.X[j]
    return ans < model.b
model.Time = Constraint(rule=Time_rule)

print "ABSTRACT MODEL"
model.pprint()

# Create the model instance
instance = model.create("prod.dat")

print "MODEL INSTANCE"
instance.pprint()
```