Noname manuscript No.
(will be inserted by the editor)

Python Optimization Modeling Objects (Pyomo)

William E. Hart · Jean-Paul Watson · David L. Woodruff

Received: December 29, 2009.

Abstract We describe Pyomo, an open source tool for modeling optimization applications in Python. Pyomo can be used to define symbolic problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo provides a capability that is commonly associated with algebraic modeling languages such as AMPL, AIMMS, and GAMS, but Pyomo's modeling objects are embedded within a full-featured high-level programming language with a rich set of supporting libraries. Pyomo leverages the capabilities of the Coopr software library, which integrates Python packages for defining optimizers, modeling optimization applications, and managing computational experiments.

Keywords Python · algebraic modeling language · optimization · open source software

1 Introduction

The Python Optimization Modeling Objects (Pyomo) software package supports the definition and solution of optimization applications using the Python scripting lan-

William E. Hart
Sandia National Laboratories
Discrete Math and Complex Systems Department
PO Box 5800, MS 1318
Albuquerque, NM 87185
E-mail: wehart@sandia.gov

Jean-Paul Watson
Sandia National Laboratories
Discrete Math and Complex Systems Department
PO Box 5800, MS 1318
Albuquerque, NM 87185
E-mail: jwatson@sandia.gov

David L. Woodruff
Graduate School of Management
University of California Davis
Davis, CA 95616-8609
E-mail: dlwoodruff@ucdavis.edu

guage. 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.

The introduction of Pyomo was motivated by a variety of factors that have impacted optimization applications at Sandia National Laboratories. Sandia's discrete mathematics group has successfully used AMPL [3, 14] 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.

However, our experience with these applications has also highlighted the need for more flexible AML frameworks. For example, direct integration with a high-level programming language is needed to allow modelers to leverage modern programming constructs, ensure cross-platform portability, and access the broad range of functionality found in standard software libraries. AML's also need to support extensibility of the core modeling language and associated solver interfaces, since complex applications typically require some degree of customization. Finally, open-source licensing is needed to manage costs associated with the deployment of optimization solutions, and to facilitate the integration of modeling capabilities from a broader technical community.

Pyomo was developed to provide an open-source platform for developing mathematical programming models that leverages Python's rich high-level programming environment to facilitate the development and deployment of optimization capabilities. Pyomo is *not* intended to facilitate modeling *better* than existing AML tools. Instead, it supports a different modeling approach in which the software is designed for flexibility, extensibility, portability, and maintainability. At the same time, Pyomo incorporates the central ideas in modern AMLs, e.g., differentiation between symbolic models and concrete problem instances.

Pyomo is a component of the Coopr software library, a COmmon Optimization Python Repository [10]. Coopr includes a flexible framework for applying optimizers to analyze Pyomo models. This includes interfaces to well-known mixed-integer linear programming (MILP) solvers, as well as an architecture that supports parallel solver execution. Coopr also includes an installation utility that automatically installs the diverse set of Python packages that are used by Pyomo.

The remainder of this paper is organized as follows. Section 2 describes the motivation and design philosophy behind Pyomo, and Section 3 discusses why Python was chosen for the design of Pyomo. Section 4 describes related high-level modeling approaches, and Pyomo is contrasted with other Python-based modeling tools. Section 5 describes the use of Pyomo on a simple application that is also described in AMPL for comparison, and Section 6 describes how Pyomo leverages the capabilities in Coopr. Section 7 illustrates the use of Pyomo in a more complex example: Bender's decomposition for stochastic linear programming. Section 8 provides information for getting started with Coopr, and Section 9 describes future work that is planned for Pyomo and Coopr.

2 Design Goals and Requirements

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

2.1 Open Source Licensing

A key goal of Pyomo is to provide an open source mathematical programming modeling capability. Although open source optimization solvers are widely available in packages like COIN-OR [8], surprisingly few open source tools have been developed to model optimization applications. The open source requirement for Pyomo is motivated by several factors:

- **Transparency and Reliability**: When managed well, open source projects facilitate transparency in software design and implementation. Because 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 and that open source software exhibits similar defect evolution patterns as closed-source software [4, 48].
- **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., who are motivated to minimize the costs of deploying an application model internally within their company). The Coopr software library, which contains Pyomo, is licensed under the BSD [7]. BSD has fewer restrictions for commercial use than alternative open source licenses like the GPL [20].

The use of an open source software development model is not a panacea; ensuring high reliability of the software still requires careful software management and a committed developer community. However, there is increasing recognition that open source software provides many advantages beyond simple cost savings [9], including supporting open standards and avoiding being locked in to a single vendor.

2.2 Customizable Capability

A key limitation of commercial modeling tools is the inability to customize the modeling or optimization processes. 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 can prototype capabilities that may eventually be integrated into future software releases.

More generally, Pyomo is designed to support a “stone soup” development model in which each developer “scratches their own itch.” A key element of this design is the plugin framework that Pyomo uses to integrate components like optimizers, optimizer managers, and optimizer model format converters. The plugin 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 the risk of destabilizing core functionality.

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 access optimization solver libraries (e.g, via static or dynamic dynamic linking). By contrast, *loosely coupled* modeling tools apply external optimization executables (e.g., through the use of 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 optimizer execution. Pyomo uses a high level (scripting) programming language to formulate a problem that can be solved by optimizers written in low-level (compiled) 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.

2.4 Symbolic Models and Concrete Instances

A requirement of Pyomo’s design is that it support the definition of symbolic models in a manner similar to that of AMPL [3, 14] and AIMMS [2, 41]. A symbolic model separates the declaration of a model from the data that generates a model instance; the advantages of this approach are widely appreciated, e.g., see Fourer et al. [15, p.35]. This separation provides an extremely flexible modeling capability, which has been leveraged extensively in optimization applications development at Sandia.

To mimic this capability, Pyomo uses a symbolic representation of data, variables, constraints, and objectives. 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 the translation of models between AMPL and Pyomo.

2.5 Flexible Modeling Language

Another goal of Pyomo is to directly use a modern high-level programming language to support the definition of mathematical programming models. In this manner, Pyomo is similar to tools like FlopC++ [13] and OptimJ [31], which support modeling in C++ and Java respectively. The use of a widely-used high-level programming language to develop mathematical programming models has several advantages:

– **Extensibility and Robustness**: A widely used high-level programming language provides a robust foundation for developing and solving mathematical programming

models: the language has been well-tested in a wide variety of application contexts and deployed on a range of computing platforms. Further, extensions almost never require changes to the core language but instead involve the definition of additional classes and compute routines that can be immediately leveraged in the modeling process. Further, support of the modeling language itself is not a long-term factor when managing software releases.
- **Documentation**: Modern high-level 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 such capabilities in a modeling tool.

An additional benefit of basing Pyomo on a general-purpose high-level programming language is that we can directly support modern programming language features, including classes, looping and procedural constructs, and first-class functions – all of which can be critical when defining complex models. By contrast, such features are not generally available in commercial AMLs like AMPL and GAMS.

Pyomo is implemented in Python, a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. However, when compared with AMLs like AMPL, Pyomo has a more verbose and complex syntax. For example, declarations like set definitions can be expressed as inlined-functions in AMPL, but they require a more verbose syntax in Python because it supports a more generic computing model. 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.

2.6 Portability

A requirement of Pyomo's design is that it work on a diverse range of computing platforms. In particular, inter-operability on both Microsoft Windows and Linux platforms is a key requirement for many Sandia applications. For example, user front-ends are often GUIs on a Windows platform, while the computational back-end may reside on a Linux cluster. The main impact of this requirement has been to limit the choice of the high-level programming language used to develop Pyomo. In particular, the Microsoft .Net languages were not considered for the design of Pyomo due to portability considerations.

3 Why Python?

We chose to develop Pyomo using the Python [38] programming language for a variety of reasons. First, Python meets the criteria outlined in the previous section:

- **Open Source License:** Python is freely available, and has a liberal open source license that allows users modify and distribute a Python-based application with few restrictions.

- **Features:** Python has a rich set of native data types, in addition to support for object oriented programming, namespaces, exceptions, and dynamic module loading.
- **Support and Stability:** Python is highly stable, widely used, and is well supported through newsgroups, web forums, and special interest groups.
- **Documentation:** Users can learn about Python from both extensive online documentation as well as a number of excellent books.
- **Standard Library:** Python includes a large number of useful modules, providing capabilities for (among others) data persistence, interprocess communication, and operating system access.
- **Extensibility 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 integrated. 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, including Windows and Linux, so portability is typically not a limitation for Python-based applications.

We considered several other popular programming languages prior to developing Pyomo. However, in most cases Python appears to have distinct advantages, e.g., relative to:

- **.Net:** As mentioned earlier, the Microsoft .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 enter obtuse symbols (e.g., $, %, and @). Thus, we expect Python will be a more natural language for expressing mathematical programming models. Further, the Python libraries are more integrated and comprehensive than those for Ruby.
- **Java:** Java has many of the same strengths as Python, and it is arguably as good a choice for the development of Pyomo. However, Python has a powerful interactive interpreter that allows real-time 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++ [13] package are similar to models developed with Pyomo. Specifically, the models are 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.

Finally, we note that run-time performance was not a key factor in our decision to use Python. Recent empirical 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 [32]. 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 [46, 39]. It is widely acknowledged that Python's dynamic typing and compact, concise syntax facilitates rapid and easy software development. Thus, it is not surprising that Python is widely used in the scientific community [29]. Large Python projects like SciPy [23] and SAGE [43] strongly leverage a diverse set of Python packages to perform complex numerical calculations.

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 [24]. AMLs like AIMMS [2], AMPL [3, 14], and GAMS [17] provide 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 or millions 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++ [13] and OptimJ [31] 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 [45] 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 [5, 25].
- **CVXOPT:** A package for convex optimization [11].
- **PuLP:** A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [34].
- **POAMS:** A modeling tool for linear and mixed-integer linear programs that defines Python objects for symbolic sets, constraints, objectives, decision variables, and solver interfaces.
- **PyMathProg:** A package that includes PyGLPK, which encapsulates the functionality of the GNU Linear Programming Kit (GLPK) [35].
- **OpenOpt:** A numerical optimization framework that is closely coupled with the SciPy scientific Python package [30].

– **NLPy:** An optimization framework that leverages AMPL to create problem instances, which can then be processed in Python [28].

Pyomo is similar to APLEpy, PyMathProg, PuLP and POAMS. All of these packages define Python objects that can be used to express models, but they can be distinguished according to the extent to which they support symbolic models. Symbolic models provide a data-independent specification of a mathematical program. Fourer and Gay [14] summarized the advantages of symbolic models 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

PuLP and PyMathProg do not support symbolic models; the concrete models that can be constructed by these tools are driven by the current data. APLEpy supports symbolic definitions of set and parameter data, but the objective and constraint specifications are concrete. POAMS and Pyomo support symbolic models, which can be used to generate concrete instances from various data sources. Pyomo also provides an automated construction process for generating concrete instances from a symbolic model. Hart [21] provides Python examples that illustrate the differences between PuLP, POAMS and Pyomo.

5 Pyomo Overview

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

5.1 A Simple Example

In this sub-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` in Figure 1, which is available from `www.ampl.com/BOOK/EXAMPLES/EXAMPLES1`.

To translate this AMPL model into Pyomo, the user must first import the Pyomo Python package and create an empty Pyomo `Model` object:

```
# Imports
from coopr.pyomo import *

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

```
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];
```

Fig. 1 The AMPL model `prod.mod`.

This import assumes that Pyomo is present in the user's Python path (see standard Python documentation for further details about the `PYTHONPATH` environment variable); the Python executable installed by the Coopr installation script described in Section 8 automatically includes all requisite paths.

Next, we create the sets and parameters that correspond to data declarations 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 the parameter b is a scalar, while parameters a, c and u are arrays indexed by the set P. Further, we observe that all AML components in Pyomo (e.g. parameters, sets, variables, constraints, and objectives) are explicitly associated with a particular model. This allows Pyomo to automatically manage the naming of AML components, and multiple Pyomo models can be simultaneously defined.

Next, we define the decision variables in the model with the `Var` class.

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

Model parameters and decision variables are used in the definition of the objectives and constraints in the model. Parameters define constants, and the variable values are determined via optimization. Parameter values are typically defined in a separate data file that is processed by Pyomo, similar to the paradigm used in AMPL and AIMMS.

Objectives and constraints are explicitly defined expressions in Pyomo. The `Objective` and `Constraint` classes typically require a `rule` option that specifies how these expressions are constructed. A rule is a function that takes one or more arguments and returns an expression that defines the constraint or objective. The last argument in a rule is the model of the corresponding objective or constraint, and the preceding arguments are index values for the objective or constraint that is being defined. If only a single argument is supplied, the constraint and objectives are necessarily singletons, i.e., non-indexed. Using these constructs, we express the AMPL objective and constraints in Pyomo as follows (Note: The backslash (\) is the Python line continuation character):

```
# 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)
```

This example illustrates several conventions for generating constraints with standard Python language constructs. 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 standard Python `sum()` function. The `Time_rule` function returns a $\leq$ expression that defines an upper bound on the constraint body. The constraint body is created with Pyomo's `summation()` function, which concisely specifies the sum of one or more expressions. 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, constraint body, and upper bound for a constraint. The Python value `None` can be supplied as one of the limit values if a bound is not enforced.

Once a symbolic Pyomo model has been created, it can be printed as follows:

```
model.pprint()
```

This command summarizes the information in the Pyomo model, but it does not print out explicit expressions. This is due to the fact that a symbolic model needs to be instantiated with data to generate the constraints and objectives. An instance of the `prod` model can be generated as follows:

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

The data in the `prod.dat` file is expressed in the AMPL data file format; this example file is available from `www.ampl.com/BOOK/EXAMPLES/EXAMPLES1`.

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

```
opt = solvers.SolverFactory("cplex")
results = opt.solve(instance)
```

This creates an optimizer object for the CPLEX executable. The Pyomo model instance is optimized, and the optimizer returns an object that contains the solution(s) generated during optimization.

Note that this optimization process is executed using other components of the Coopr library. The `coopr.opt` package manages the setup and execution of optimizers, and Coopr optimization plugins are used to manage the execution of specific optimizers.

Finally, the results of the optimization can be displayed simply by executing the following command:

```
results.write(num=1)
```

Here, the `num` parameter indicates the maximum number of solutions from the results object to be written.

5.2 Advanced Pyomo Modeling Features

The previous example provides a simple illustration of Pyomo's modeling capabilities. Much more complex models can be developed using Pyomo's flexible modeling components. For example, multi-dimensional set and parameter data can be naturally represented using Python tuple objects. The `dimen` option can be used to specify the dimensionality of set or parameter data:

```
model.p = Param(model.A, dimen=2)
```

In this example, the `p` parameter is an array of data values indexed by `A` for which each value is a 2-tuple.

Additionally, set and parameter components can be directly initialized with data using the `initialize` option. In the simplest case, this option specifies data that is used in the component:

```
model.s = Set(initialize=[1,3,5,7])
```

For arrays of data, a Python dictionary can be specified to map data index values to set or parameter values:

```
model.A = Set(initialize=[1,3,5,7])
model.s = Set(model.A, initialize={1:[1,2,3], 5:[1]})
model.p = Param(model.A, initialize={1:2, 3:4, 5:6, 7:8})
```

More generally, the `initialize` option can specify a function that returns data values used to initialize the component:

```
model.n = Param(initialize=10)
def x_init(model):
    return [2*i for i in range(0, model.n.value)]
model.x = Set(initialize=x_init)
```

Set and parameter components also include mechanisms for validating data. The `within` option specifies a set that contains the corresponding set or parameter data, for example:

```
model.s = Set(within=Integers)
model.p = Param(within=model.s)
```

The `validate` option can also specify a function that returns a Boolean indicating whether data is acceptable:

```
def p_valid(val, model):
    return val >= 0.0
model.p = Param(initialize=p_valid)
```

Pyomo includes a variety of *virtual* set objects that do not contain data but which can be used to perform validation, e.g.:

- `Any` - Any numeric or non-numeric value other than the Python `None` value.
- `PositiveReals` - Positive real values
- `NonNegativeIntegers` - Non-negative integer values
- `Boolean` - Zero or one values

Finally, there are many contexts where it is necessary to specify index sets that are more than simple product sets. To simplify model development in these cases, Pyomo supports set index rules that automatically generate temporary sets for indexing. For example, the following example illustrates how to index a variable on 2-tuples (i, j) for which $i < j$:

```
model.n = Param(within=PositiveIntegers)
def x_index(model):
    return [(i,j) for i in range(0,model.n.value)
                  for j in range(0,model.n.value) if i<j]
model.x = Var(x_index)
```

5.3 The Pyomo Command

Although a Pyomo Python script can be executed directly from within the Python interpreter, Pyomo includes the `pyomo` command that can construct an abstract model, create a concrete instance from user-supplied data, apply an optimizer, and summarize the results. For example, the following command line optimizes the `prod` model using the data in `prod.dat`:

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

The `pyomo` command automatically executes the following steps:

– Create a symbolic model
– Read the instance data
– Generate a concrete instance from the symbolic model and instance data
– Apply simple presolvers to the concrete instance
– Apply a solver to the concrete instance
– Load the results into the concrete instance

The `pyomo` script supports a variety of command-line options to guide and provide information about the optimization process; documentation of the various available options is available by specifying the `--help` option. Options can control how much or even if 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 facilitates debugging of the model construction process.

5.4 Run-Time Performance

Run-time performance was not the most significant factor in our choice of Python as a modeling language, but this is clearly an important factor for Pyomo users. Although the optimization solver run-time is typically dominant, in some applications the time needed to construct complex models is nontrivial. Thus, we have simplified and tuned the behavior of Pyomo components to minimize the run-time required for model construction.

Figure 2 shows the run-time performance for Pyomo and AMPL on instances of the well-known p-median facility location problem. We have considerable experience with solving large-scale p-median problem instances arising in water security contexts [22], motivating our investigation of run-time scaling using this particular optimization problem. The three curves in this plot show the time, in seconds, that was needed to generate a model with AMPL, with Pyomo, and with Pyomo when Python's garbage collection mechanism was disabled. The x-axis captures the problem size for p-median problems where the number of customers, N, equals the number of facilities, M, and each point in this plot represents the average of 15 different trials.

This figure shows that there is a substantial performance gap between Pyomo for large problem instances. For the largest problems, Pyomo was 136 times slower than AMPL with garbage collection, and 32 times slower without garbage collection. This difference highlights the fact that Python is notoriously slow at deconstructing (i.e., garbage-collecting) objects. Profile data for these experiments also shows that a large fraction of the run-time is spent constructing expressions. Pyomo uses expression trees that represent each arithmetic operation and data value with a Python object; thus, constructing large expression trees involves the construction of many Python objects, which is a well-known performance issue in Python.

The elimination of garbage collection was suggested by the experimental analysis of APLEpy [25]. Although this option makes sense for simple analyses, this option would probably not make sense for complex Pyomo scripts that generate many problem instances. The APLEpy experiments also employ the Psyco JIT compiler [33] to further improve APLEpy performance. However, we omit Psyco experiments since it can only be used on 32-bit computers, which are no longer commonly available.

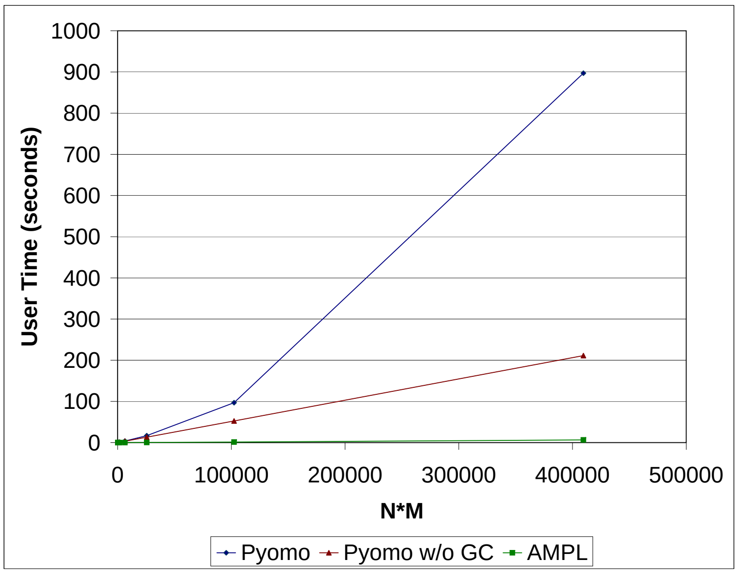

Fig. 2 Execution time needed to construct a model instance with AMPL, Pyomo, and Pyomo with garbage collection disabled.

6 The Coopr Optimization Package

Much of Pyomo's flexibility and extensibility is due to the fact that Pyomo is integrated into Coopr, a COmmon Optimization Python Repository [10]. Coopr utilizes a component-based software architecture to provide plugins that modularize many aspects of the optimization process. This allows Coopr to support a generic optimization process. Coopr components manage the execution of optimizers, including run-time detection of available optimizers, conversion to file formats needed by an optimizer, and transparent parallelization of independent optimization tasks.

6.1 Generic Optimization Process

Pyomo strongly leverages Coopr's ability to execute optimizers in a generic manner. For example, the following Python script illustrates how an optimizer is initialized and executed with Coopr:

```
opt = SolverFactory(solver_name)
results = opt.solve(concrete_instance)
results.write()
```

This example illustrates Coopr's explicit segregation of problems and solvers into separate objects. Such segregation promotes the development of tools like Pyomo that define optimization applications.

The `results` object returned by a Coopr optimizer includes information about the problem, the solver execution, and one or more solutions generated during optimization.

This object supports a general representation of optimizer results that is similar in spirit to the results encoding scheme used by the COIN-OR OS project [16]. The main difference is that Coopr represents results in YAML, a data serialization format that is both powerful and human readable [47]. For example, Figure 3 shows the results output after solving the problem described in Section 5.

6.2 Solver Parallelization

Coopr includes two components that manage the execution of optimization solvers. First, `Solver` objects manage the local execution of an optimization solver. For example, Coopr includes MILP plugins for solvers like CPLEX and CBC. Second, `SolverManager` objects coordinate the execution of `Solver` objects in different environments. The API for solver managers supports asynchronous execution of solvers as well as solver synchronization. Coopr includes solver managers that execute solvers serially, in parallel on compute clusters, and remotely with the NEOS optimization server [12].

Coopr's Pyro solver manager supports parallel execution of solvers using two key mechanisms: the standard Python `pickle` module and the Pyro distributed computing library [37]. The `pickle` module performs object serialization, which is a pre-requisite for distributed computation. With few exceptions, any Python object can be "pickled" for transmission across a communications channel. This includes simple, built-in objects such as lists, and more complex Pyomo objects like `Model` instances.

Pyro (Python Remote Objects) [37] is a mature third-party Python library that provides an object-oriented framework for distributed computing that is similar to Remote Procedure Calls. Pyro is cross-platform, such that different application components can execute on fundamentally different platform types, e.g., Windows and Linux. Inter-process communication is facilitated through a standard name server mechanism, and object serialization and de-serialization is performed via the Python `pickle` module.

The standard Coopr distribution includes both the Pyro distribution and a number of solver-centric utilities to facilitate parallel solution of Pyomo models. All communication is established through the Coopr name server, invoked via the `coopr-ns` script on some host node in a distributed system. A "router" process is then launched on a compute node via the Coopr `dispatch_srvr` script. Finally, one or more solver processes are launched on various compute nodes via the Coopr `pyro_mip_server` script. Each solver process identifies a dispatch server through the name server and notifies the dispatch server that it is available for solving instances. Note that the various solver processes can execute on distinct cores of a single workstation, across multiple workstations, and even across multiple workstations running different operating systems, e.g., hybrid Windows/Linux clusters. Communication with the global name server is typically accomplished by setting the `PYRO_NS_HOSTNAME` environment variable to identify the name server host; in a non-distributed (e.g., SMP) environment, such communication is automatic.

Once initialized, the distributed solver infrastructure is accessed by the Pyro solver manager, which can be constructed using the `SolverManagerFactory` functionality:

```
solver_manager = SolverManagerFactory("pyro")
```

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================

# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: 192000
  Number of objectives: 1
  Number of constraints: 6
  Number of variables: 3
  Number of nonzeros: 7
  Sense: maximize

# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
  Termination condition: unsure
  Error rc: 0

# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: 0.0
  Status: optimal
  Objective:
    f:
      Id: 0
      Value: 192000
  Variable:
    X[bands]:
      Id: 0
      Value: 6000
    X[coils]:
      Id: 1
      Value: 1400
  Constraint:
    c_u_Limit[bands]_:
      Id: 1
      Dual: 4
    c_u_Time_:
      Id: 4
      Dual: 4200
```

Fig. 3 Results output for the production planning model described in Section 5.

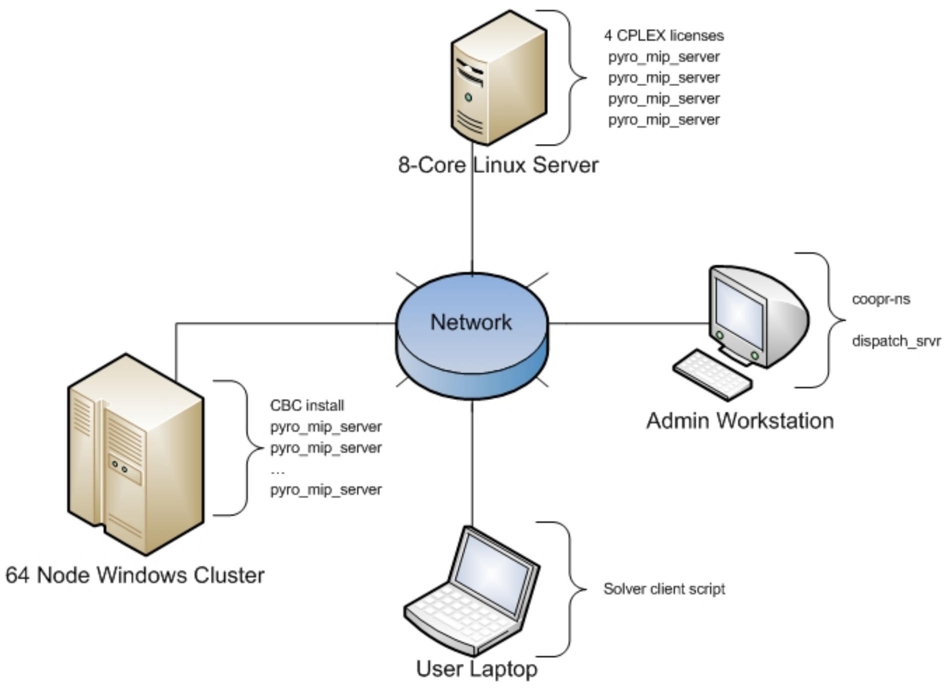

Fig. 4 An example of a hybrid compute architecture and how it can be configured for distributed solves using Pyro and Coopr.

The Pyro solver manager identifies dispatch servers through the Coopr name server, routing problem instances and retrieving results through these conduits. Thus, a simple change in the argument name to the solver manager factory is sufficient to access a distributed solver resource in Coopr!

Figure 4 illustrates a typical architecture for distributed solver computation in Coopr. In this example, the user executes a solver script (e.g., the `runbenders` script described in Section 7) on his or her local machine. The name and dispatch server processes (`coopr-ns` and `dispatch_srvr`) are configured as daemons on an administrator workstation, such that they are persistent. This example network has two major compute resources: a Windows cluster and a multi-core Linux server. The Linux server has four CPLEX licenses, while (due to cost) the Windows cluster has the open-source CBC solver available on all compute nodes. Four `pyro_mip_server` daemons are executing on the server (each is allocated two cores), while a `pyro_mip_server` daemon is executing on each compute node of the Windows cluster. In this particular example, all user-solver communication is performed via the sole dispatch server; in general, multiple dispatch servers can be configured to mitigate communication bottlenecks.

6.3 Solver Plugins

A common object oriented approach for mathematical programming software is to use classes and class inheritance to develop extensible functionality. For example, the OPT++ [27] optimization software library defines base classes with different characteristics (e.g., differentiability), and a concrete optimization solver is instantiated as a subclass of an appropriate base class. In this context, the base class can be viewed as defining the interface for the solvers that inherit from it.

Coopr plugins leverage the PyUtilib Component Architecture (PCA) to separate the declaration of component interfaces from their implementation [40]. For example, the interface to optimization solvers are again declared with a class. However, solver plugins are not required to be subclasses of the interface class. Instead, they are simply required to provide the same interface methods and data.

Coopr uses plugin components to modularize the steps needed to perform optimization. A component is a software package, module, or object that provides a particular functionality. Plugin components augment the execution flow by implementing functionality that is exercised "on demand." Component-based software with plugins is a widely recognized best practice for extending and evolving complex software systems in a reliable manner [42]. Component-based software frameworks manage the interaction between components to promote adaptability, scalability, and maintainability in large software systems [44]. For example, with component-based software there is much less need for major release changes because software changes can be encapsulated within individual components. Component architectures also encourage third-party developers to add value to software systems without risking destabilization of the core functionality.

Coopr uses the PCA to define interfaces for the following plugin components:

– solvers, which perform optimization
– solver managers, which manage the execution of solvers
– converters, which translate between different optimization problem file formats
– solution readers, which load optimization solutions from files
– problem writers, which create files that specify optimization problems
– model transformations, which generate new models via reformulation of existing models

Coopr also contains Pyomo-specific components for preprocessing Pyomo models before they are solved.

Coopr includes a variety of plugins that implement these component interfaces, many of which rely on third-party software packages to implement key functionality. For example, solver plugins are available for the CPLEX, CBC, PICO, and GLPK mixed-integer linear programming solvers. These plugins rely on the availability of binary executables for these solvers, which need to be installed separately. Similarly, Coopr includes plugins that convert between different optimization problem file formats; these plugins rely on binary executables built by the GLPK [19] and Acro [1] software libraries.

Figure 5 illustrates the definition of a solver plugin that can be directly used by the `pyomo` command; this example is available in the standard Coopr distribution, in the directory `coopr/examples/pyomo/p-median`. The `MySolver` class implements the `IOptSolver` interface, which declares this class as a Coopr solver plugin. This plugin implements a `solve` method, which randomly generate solutions to a p-median optimization problem. The only other step needed is to use Coopr's `SolverRegistration` function, which associates the solver name, `random`, with the plugin class, `MySolver`.

Importing the Python module containing `MySolver` activates this plugin; all other registration is automated by the PCA. Thus, if this plugin is in the file `solver2.py`,

```
# Imports from Coopr and PyUtilib
from coopr.pyomo import *
from pyutilib.plugin.core import *
from coopr.opt import *
import random
import copy

class MySolver(object):

    # Declare that this is an IOptSolver plugin
    implements(IOptSolver)

    # Solve the specified problem and return
    # a SolverResults object
    def solve(self, instance, **kwds):
        print "Starting random heuristic"
        val, sol = self._random(instance)
        n = value(instance.N)
        # Setup results
        results = SolverResults()
        results.problem.name = instance.name
        results.problem.sense = ProblemSense.minimize
        results.problem.num_constraints = 1
        results.problem.num_variables = n
        results.problem.num_objectives = 1
        results.solver.status = SolverStatus.ok
        soln = results.solution.add()
        soln.value = val
        soln.status = SolutionStatus.feasible
        for j in range(1,n+1):
            soln.variable[instance.y[j].name] = sol[j-1]
        # Return results
        return results

    # Perform a random search
    def _random(self, instance):
        sol = [0]*instance.N.value
        for j in range(0,instance.P.value):
            sol[j] = 1
        # Generate 100 random solutions, and keep the best
        best = None
        best_sol = []
        for kk in range(100):
            random.shuffle(sol)
            # Compute value
            val=0.0
            for j in range(1,instance.M.value+1):
                val += min([instance.d[i,j].value
                            for i in range(1,instance.N.value+1)
                            if sol[i-1] == 1])
            if best is None or val < best:
                best=val
                best_sol=copy.copy(sol)
        return [best, best_sol]

# Register the solver with the name 'random'
SolverRegistration("random", MySolver)
```

Fig. 5 A simple customized solver for p-median problems.

then the following Python script can be used to apply this solver to Coopr's p-median example:

```
import coopr.opt
import pmedian
import solver2

instance=pmedian.model.create('pmedian.dat')
opt = coopr.opt.SolverFactory('random')
results = opt.solve(instance)
print results
```

The `pyomo` script can also be used to apply a custom optimizer in a natural manner. The following command-line is used to solve the Coopr's p-median example with the `cbc` integer programming solver:

```
pyomo --solver=cbc pmedian.py pmedian.dat
```

Applying the custom solver simply requires the specification of the new solver name, `random`, and an indication that the `solver2.py` file should be imported before optimization:

```
pyomo --solver=random --preprocess=solver2.py pmedian.py \
                                              pmedian.dat
```

Thus, the user can develop custom solvers in Python modules, which are tested directly using the `pyomo` command.

7 Advanced Modeling and Scripting: Benders Decomposition

An important consequence of the Python-based design of Pyomo and its integration with the Coopr environment is that modularity is fully supported over a range of abstraction. At one extreme, model elements can be manipulated explicitly by specifying their names and the values of their indexes. This sort of reference can be made more abstract, as is the case with algebraic modeling languages, by specifying various types of named sets so that the dimensions and details of the data can be separated from the specification of the model. Separation of a symbolic model from the data specification is a hallmark of structured modeling techniques [18]. At the other extreme, elements of a mathematical program can be treated in their fully canonical form as is supported by callable solver libraries. Methods can be written that operate, for example, on objective functions or constraints in a fully general way. This capability is a fundamental tool for general algorithm development and extension [26]. Pyomo provides the full continuum of abstraction between these two extremes to support modeling and development. Furthermore, methods are extensible via overloading of all defined operations. Both modelers and developers can alter the behavior of a package or add new functionality. We provide a glimpse of this general programming capability below; more exhaustive and intricate examples are found in Coopr's PySP stochastic programming package, the discussion of which is outside the present scope.

In Section 5, we presented a straightforward use of Pyomo: to construct a concrete instance from a symbolic model and a data file, and to subsequently solve the instance using a specific solver plugin. A generic optimization process is executed by the `pyomo` command, so the typical user does not need to understand the details of the functionality present in most Pyomo and Coopr libraries. However, this command masks much of the power underlying Pyomo and Coopr, and it limits the degree to which Python's rich set of libraries can be leveraged.

To explore a more complex, programmatic use of Pyomo, we consider the translation of an AMPL example involving the solution of a stochastic linear programming problem involving production planning via Benders decomposition. Given a number of product types and associated production rates, production limits, and inventory holding costs, the objective is to determine a production schedule over a number of weeks that maximizes the expected profit over a range of anticipated revenues. The problem is formulated as a two-stage stochastic linear program; first-stage decisions include the initial production, sales, and inventory quantities, while second-stage decisions (given scenario-specific revenues) include analogous parameters for all time periods other than the initial week. We refer the reader to Bertsimas and Tsitsiklis [6] for a discussion of how such two-stage stochastic linear programs can be solved via Benders decomposition.

The AMPL example consists of the three files `stoch2.run` (an AMPL script), `stoch2.mod` (an AMPL model file), and `stoch.dat` (an AMPL data file), which are available from `http://www.ampl.com/NEW/LOOP2`. The translation of this AMPL example into Pyomo illustrates many of the more powerful capabilities of Pyomo and Coopr, including dynamic construction of model variables and constraints, as well as parallelization of sub-problem solves.

The codes for this example are available in the Coopr distribution, in the directory `coopr/examples/pyomo/benders`. The first step in the translation process involves creation of the master and sub-problem symbolic models, mirroring the process previously documented in Section 5; the resulting models are captured in the files `master.py` and `subproblem.py`. We observe that AMPL allows "pick-and-choose" selection of components from a single model definition file to construct sub-models. However, Pyomo requires the definition of distinct models.

The Python code to execute Benders decomposition for this particular example is found in the file `runbenders`; the remainder of this section will explore key aspects of this code in more detail.

As with the basic Pyomo example introduced in Section 5, the Python script begins by loading the necessary components of the Pyomo, Coopr, PyUtilib, and Python system libraries:

```python
import sys
from pyutilib.misc import import_file
from coopr.opt.base import SolverFactory
from coopr.opt.parallel import SolverManagerFactory
from coopr.opt.parallel.manager import solve_all_instances
from coopr.pyomo import *
```

The need for and role of these various modules will be explained below.

The first executable component of the **runbenders** script involves construction of the symbolic models and concrete problem instances for both the master and second-stage sub-problems:

```
# initialize the master instance.
mstr_mdl = import_file("master.py").model
mstr_inst = mstr_mdl.create("master.dat")

# initialize the sub-problem instances.
sb_mdl = import_file("subproblem.py").model
sub_insts = []
sub_insts.append(sb_mdl.create(name="Base Sub-Problem", \
                               filename="base_subproblem.dat"))
sub_insts.append(sb_mdl.create(name="Low Sub-Problem", \
                               filename="low_subproblem.dat"))
sub_insts.append(sb_mdl.create(name="High Sub-Problem", \
                               filename="high_subproblem.dat"))
```

The PyUtilib function `import_file` imports the Python code specified in the input argument into a Python object; any defined symbols (objects, functions, etc.) can be accessed by referencing attributes of this object. In this example, the `master.py` and `subproblem.py` model definition files are loaded, defining the associated symbolic models; the `runbenders` script accesses the corresponding `model` objects. Given a symbolic model object, a concrete instance can be created by invoking its `create` method supplied with an argument specifying a data file. For reasons discussed below, the second stage sub-problems are gathered into a Python list.

Next, we create the necessary solver and solver manager plugins:

```
# initialize the solver and solver manager.
solver = SolverFactory("cplex")
if solver is None:
    print "A CPLEX solver is not available on this machine."
    sys.exit(1)
solver_manager = SolverManagerFactory("serial") # serial
#solver_manager = SolverManagerFactory("pyro") # parallel
```

In this example, we use CPLEX to solve concrete instances, since it provides the variable and constraint suffixes needed for this Bender's decomposition solver (e.g., reduced-costs, as discussed below). In Coopr, the solver manager is responsible for coordinating the execution of any solver plugins. The code fragment above specifies two alternative solver managers, with serial execution enabled by default; see Section 6.2 for a discussion of parallel solver execution (via the Pyro solver manager).

Benders decomposition is an iterative process; sub-problems are solved, cuts are added to the master problem, the master problem is (re-)solved; the process repeats until convergence. In the `master.py` model, the set of cuts and the corresponding constraint set is defined as follows:

```
model.CUTS = Set(within=PositiveIntegers, ordered=True)

model.Cut_Defn = Constraint(model.CUTS)
```

Initially, the CUTS set is empty. Consequently, no rule is required in the definition of the constraint set Cut_Defn. Similarly, the pricing (i.e., reduced-cost and dual) information from the sub-problems is stored in the following parameters within master.py (such pricing information is integral in the definition of Benders cuts [6]):

```
model.time_price = Param(model.TWOPLUSWEEKS, model.SCEN, \
                         model.CUTS, default=0.0)

model.bal2_price = Param(model.PROD, model.SCEN, model.CUTS, \
                         default=0.0)

model.sell_lim_price = Param(model.PROD, model.TWOPLUSWEEKS, \
                             model.SCEN, model.CUTS, \
                             default=0.0)
```

Again, because the index set CUTS is empty, these parameter sets are initially empty.

Given these definitions, we now examine the main loop in the runbenders script. The first portion of the loop body solves the second stage sub-problems as follows:

```
solve_all_instances(solver_manager, solver, sub_insts)
```

The solve_all_instances function is a Coopr utility function that performs three distinct operations: (1) queue the sub-problem solves with the solver manager, (2) solve each of the sub-problems, and (3) load the results into the sub-problem instances. This function encapsulates the detailed logic of queuing, solver/solver manager interactions, and barrier synchronization; such detail can be exposed as needed, e.g., when sub-problems can be solved asynchronously.

Next, the index set CUTS is expanded, and the pricing parameters are extracted from the sub-problem instances and stored in the master instance:

```
mstr_inst.CUTS.add(i)

for s in range(1, len(subproblems)+1):

   inst = sub_insts[s-1]

   for t in mstr_inst.TWOPLUSWEEKS:
      mstr_inst.time_price[t,s,i] = inst.Time[t].dual
   for p in mstr_inst.PROD:
      mstr_inst.bal2_price[p,s,i] = inst.Balance2[p].dual
   for p in mstr_inst.PROD:
      for t in mstr_inst.TWOPLUSWEEKS:
         mstr_inst.sell_lim_price[p,t,s,i] = inst.Sell[p,t].urc
```

The first line in this code block dynamically expands the size of the CUTS set, adding an element corresponding to the current Benders loop iteration counter i. The code for transferring pricing information from the sub-problems to the master instance is straightforward: access of pricing parameters with a sub-index equal to i in the master instance is legal given the dynamic update of the CUTS set. Additionally, we observe the availability in Pyomo of standard variable "suffix" information (in this case constraint dual variables and variable upper reduced costs).

With pricing information available in the master instance, we can now define the new cut for Benders iteration `i` as follows:

```
cut = sum([mstr_inst.time_price[t,s,i] * mstr_inst.avail[t] \
           for t in mstr_inst.TWOPLUSWEEKS \
           for s in mstr_inst.SCEN]) + \
      sum([mstr_inst.bal2_price[p,s,i] * (-mstr_inst.Inv1[p]) \
           for p in mstr_inst.PROD \
           for s in mstr_inst.SCEN]) + \
      sum([mstr_inst.sell_lim_price[p,t,s,i] * \
           mstr_inst.market[p,t] \
           for p in mstr_inst.PROD \
           for t in mstr_inst.TWOPLUSWEEKS \
           for s in mstr_inst.SCEN]) - \
      mstr_inst.Min_Stage2_Profit

mstr_inst.Cut_Defn.add(i, (0.0, cut, None))
```

The expression for the cut is formed using the Python `sum` function in conjunction with Python's list comprehension syntax. While somewhat more complex, the fundamentals of constraint generation shown above are qualitatively similar to the constraint rule generation examples presented in Section 5.1. Given the cut expression, the corresponding new element of the `Cut_Defn` constraint set is created via the `add` method of the `Constraint` class. Here, the parameters respectively represent (1) the constraint index and (2) a tuple consisting of the constraint (lower bound, expression, upper bound).

The remainder of the `runbenders` script is straightforward. This example illustrates that sophisticated optimization strategies requiring direct access to Pyomo's modeling capabilities and Coopr's optimization capabilities can be easily implemented. The `runbenders` script has roughly the same complexity and length as the original AMPL script. Additionally, this script supports parallelization of this solver in a generic and straightforward manner.

8 Getting Started with Pyomo

The installation of Pyomo is complicated by the fact that Coopr relies on a variety of other Python packages, as well as third-party numerical software libraries. Coopr provides an installation script, `coopr_install`, that leverages Python's online package index [36] to install Coopr-related Python packages. For example, in Linux the command:

```
   coopr_install coopr
```

will create a `coopr` directory that contains a virtual Python installation. All Coopr scripts will be installed in the `coopr/bin` directory, and these scripts are configured to automatically import the Coopr and PyUtilib libraries, in addition to other necessary Python packages. Similarly, in MS Windows the command:

```
python coopr_install coopr
```

will create a `coopr` directory containing a virtual Python installation. The Coopr scripts will be installed in the `coopr/bin` directory, including `*.cmd` scripts that are recognized as DOS commands. In both Linux and MS Windows, the only additional user requirement is that the `PATH` environment variable be updated to include `coopr/bin` on Linux platforms, or `coopr\bin` on MS Windows platforms.

By default, the `coopr_install` script installs the latest official release of all Coopr-related Python packages. The `coopr_install` script also includes options for installing trunk versions (via the `--trunk` option) or the latest stable versions (via the `--stable` option) of the Coopr and PyUtilib Python packages. This facilitates collaborations with non-developers to fix bugs and try out new Coopr capabilities.

More information regarding Coopr is available on the Coopr wiki, available at: `https://software.sandia.gov/trac/coopr/`. This Trac site includes detailed installation instructions, and provides a mechanism for users to submit tickets for feature requests and bugs. Coopr has been proposed as a COIN-OR software package [8], so a COIN-OR Trac site and associated mailing lists may also be available soon.

9 Discussion and Conclusions

Pyomo is being actively developed to support real-world applications at Sandia National Laboratories [22]. Our experience to date with Pyomo and Coopr has validated our initial assessment that Python is an effective language for supporting the development and deployment of solutions to optimization applications. Although it is clear that custom AMLs can support a more concise and mathematically intuitive syntax, Python's clean syntax and programming model make it a natural choice for optimization tools like Pyomo.

It is noteworthy that the use of Python for developing Pyomo has proven quite strategic. First, the development of Pyomo did not require developing a parser and the effort associated with cross-platform deployment, both of which are necessary for the development of an AML. Second, Pyomo users have been able to rely on Python's extensive documentation to rapidly get up-to-speed without relying on developers to provide detailed language documentation. Although general use of Pyomo requires documentation of Pyomo-specific features, this is a much smaller burden than the language documentation required for mathematical programming AMLs. Finally, we have demonstrated that Pyomo can effectively leverage Python's rich set of standard and third-party libraries to support advanced computing capabilities like distributed execution of optimizers. This clearly distinguishes Pyomo from custom AMLs, and it frees Pyomo and Coopr developers to focus on innovative mathematical programming capabilities.

Pyomo and Coopr were publicly released as an open source project in 2008. Future development will focus on several key design issues:

- Nonlinear Problems - Conceptually, it is straightforward to extend Pyomo to support the definition of general nonlinear models. However, the model generation and expression mechanisms need to be re-designed to support capabilities like automatic differentiation.
- Optimized Expression Trees - Our scaling experiments suggest that Pyomo's runtime performance can be significantly improved by using a different representation

for expression trees. The representation of expression trees could be reworked to avoid frequent object construction, either through a low-level representation or a Python extension library.

- Extending the Range of Solver Plugins - We plan to expand the suite of available solver plugins to include the full range of commercial solvers, including XpressMP and Gurobi. Additionally, we plan to leverage optimizers that are available in other Python optimization packages, which are particularly interesting when solving non-linear formulations.
- Direct Optimizer Interfaces - Currently, all Coopr solvers are invoked via system calls. However, direct direct library interfaces to optimizers are also possible, since this is a capability that is strongly supported by Python. This is not a design limitation of Coopr, but instead has been a matter of development priorities.
- Remote Solver Execution - Coopr currently supports solver managers for remote solver execution using Pyro. A preliminary interface to NEOS [12] has been developed, but this solver manager currently only supports `CBC`. We plan to extend this interface, and to develop interfaces for Optimization Services [16], as well as cloud computing solvers.

Acknowledgements

We thank Jon Berry, Robert Carr, and Cindy Phillips for their critical feedback on the design of Pyomo, and David Gay for developing the Coopr interface to AMPL NL and SOL files. 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. ACRO. ACRO optimization framework. `http://software.sandia.gov/acro`, 2009.
2. AIMMS. AIMMS home page. `http://www.aimms.com`, 2008.
3. AMPL. AMPL home page. `http://www.ampl.com/`, 2008.
4. Prasanth Anbalagan and Mladen Vouk. On reliability analysis of open source software - FEDORA. In *19th International Symposium on Software Reliability Engineering*, 2008.
5. APLEpy. APLEpy: An open source algebraic programming language extension for Python. `http://aplepy.sourceforge.net/`, 2005.
6. Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific / Dynamic Ideas, 1997.
7. BSD. Open Source Initiative (OSI) - the BSD license. `http://www.opensource.org/licenses/bsd-license.php`, 2009.
8. COINOR. COIN-OR home page. `http://www.coin-or.org`, 2009.
9. Forrester Consulting. Open source software's expanding role in the enterprise. `http://www.unisys.com/eprise/main/admin/corporate/doc/Forrester_research-open_source_buying_behaviors.pdf`, 2007.
10. COOPR. Coopr: A common optimization python repository. `http://software.sandia.gov/coopr`, 2009.

11. CVXOPT. CVXOPT home page. `http://abel.ee.ucla.edu/cvxopt`, 2008.
12. Elizabeth D. Dolan, Robert Fourer, Jean-Pierre Goux, Todd S. Munson, and Jason Sarich. Kestrel: An interface from optimization modeling systems to the NEOS server. *INFORMS Journal on Computing*, 20(4):525–538, 2008.
13. FLOPC++. FLOPC++ home page. `https://projects.coin-or.org/FlopC++`, 2008.
14. 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.
15. Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A mathematical programming language. *Management Science*, 36:519–554, 1990.
16. Robert Fourer, Jun Ma, and Kipp Martin. Optimization services: A framework for distributed optimization. *Mathematical Programming*, 2008. (submitted).
17. GAMS. GAMS home page. `http://www.gams.com`, 2008.
18. Arthur M. Geoffrion. An introduction to structured modeling. *Management Science*, 33(5):547–588, 1987.
19. GLPK. GLPK: GNU linear programming toolkit. `http://www.gnu.org/software/glpk/`, 2009.
20. GPL. GNU general public license. `http://www.gnu.org/licenses/gpl.html`, 2009.
21. W. E. Hart. Python Optimization Modeling Objects (Pyomo). In J. W. Chinneck, B. Kristjansson, and M. J. Saltzman, editors, *Operations Research and Cyber-Infrastructure*, 2009. doi: 10.1007/978-0-387-88843-9_1.
22. William E. Hart, Cynthia A. Phillips, Jonathan Berry, E.G. Boman, et al. US environmental protection agency uses operations research to reduce contamination risks in drinking water. *INFORMS Interfaces*, 39:57–68, 2009.
23. Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2009. URL `http://www.scipy.org/`.
24. Josef Kallrath. *Modeling Languages in Mathematical Optimization.* Kluwer Academic Publishers, 2004.
25. Suleyman Karabuk and F. Hank Grant. A common medium for programming operations-research models. *IEEE Software*, pages 39–47, 2007.
26. Roy E. Marsten. The design of the XMP linear programming library. *ACM Transactions on Mathematical Software*, 7(4):481–497, 1981.
27. J. C. Meza. OPT++: An object-oriented class library for nonlinear optimization. Technical Report SAND94-8225, Sandia National Laboratories, 1994.
28. NLPy. NLPy home page. `http://nlpy.sourceforge.net/`, 2008.
29. Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, pages 10–20, May/June 2007.
30. OpenOpt. OpenOpt home page. `http://scipy.org/scipy/scikits/wiki/OpenOpt`, 2008.
31. OptimJ. Ateji home page. `http://www.ateji.com`, 2008.
32. Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.876288.
33. Psyco. Psyco. `http://psyco.sourceforge.net/`, 2008.
34. PuLP. PuLP: A python linear programming modeler. `http://130.216.209.237/engsci392/pulp/FrontPage`, 2008.

35. PyMathProg. PyMathProg home page. `http://pymprog.sourceforge.net/`, 2009.
36. PyPI. Python package index. `http://pypi.python.org/pypi`, 2009.
37. PYRO. PYRO: Python remote objects. `http://pyro.sourceforge.net`, 2009.
38. Python. Python programming language – official website. `http://python.org`, 2009.
39. PythonVSJava. Python & Java: A side-by-side comparison. `http://www.ferg.org/projects/python_java_side-by-side.html`, 2008.
40. PyUtilib. PyUtilib optimization framework. `http://software.sandia.gov/pyutilib`, 2009.
41. Marcel Roelofs and Johannes Bisschop. *AIMMS 3.9 – The User's Guide*. lulu.com, 2009.
42. Gigi Sayfan. Building your own plugin framework. *Dr. Dobbs Journal*, Nov 2007.
43. William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. `http://www.sagemath.org`.
44. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.
45. TOMLAB. TOMLAB optimization environment. `http://www.tomopt.com/tomlab`, 2008.
46. Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.
47. YAML. The official YAML web site. `http://yaml.org/`, 2009.
48. Ying Zhou and Joseph Davis. Open source software reliability model: An empirical approach. *ACM SIGSOFT Software Engineering Notes*, 30:1–6, 2005.