Math. Prog. Comp. (2011) 3:219–260
DOI 10.1007/s12532-011-0026-8

FULL LENGTH PAPER

Pyomo: modeling and solving mathematical programs in Python

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

Received: 29 December 2009 / Accepted: 13 June 2011 / Published online: 10 August 2011
© Springer and Mathematical Optimization Society 2011

Abstract We describe Pyomo, an open source software package for modeling and solving mathematical programs in Python. Pyomo can be used to define abstract and concrete problems, create problem instances, and solve these instances with standard open-source and commercial solvers. Pyomo provides a capability that is commonly associated with algebraic modeling languages such as AMPL, AIMMS, and GAMS. In contrast, 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 together with Pyomo is part of IBM's COIN-OR open-source initiative for operations research software. Coopr integrates Python packages for defining optimizers, modeling optimization applications, and managing computational experiments. Numerous examples illustrating advanced scripting applications are provided.

Keywords Python · Algebraic modeling language · Optimization · Open source optimization software

Mathematics Subject Classification (2000) 90C99 · 93A30 · 97N80

W. E. Hart (✉)
Data Analysis and Informatics Department, Sandia National Laboratories,
PO Box 5800, MS 1318, Albuquerque, NM 87185, USA
e-mail: wehart@sandia.gov

J.-P. Watson
Discrete Math and Complex Systems Department, Sandia National Laboratories,
PO Box 5800, MS 1318, Albuquerque, NM 87185, USA
e-mail: jwatson@sandia.gov

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

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 high-level programming language that has a very clear, readable syntax and intuitive object orientation. Pyomo includes Python classes for defining sparse sets, parameters, and variables, which can be used to formulate algebraic expressions that define objectives and constraints. Pyomo can be used to represent linear, mixed-integer, non-linear, and non-linear mixed-integer 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 impact optimization applications at Sandia National Laboratories, a large US Department of Energy Laboratory at which two of the co-authors are employed. Sandia's discrete mathematics group has successfully used AMPL [3,19] to model and solve large-scale integer programs for many years. Our application experience highlighted the value of Algebraic Modeling Languages (AMLs) for solving real-world optimization applications, and AMLs are now an integral part of operations research solutions at Sandia.

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. AMLs 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 required 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 optimization 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, primarily commercial 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 abstract models and concrete problem instances.

Pyomo is a component of the Coopr software library, a COmmon Optimization Python Repository [12]. Coopr includes a flexible framework for applying optimizers to analyze Pyomo models, interfaces to well-known linear, mixed-integer, and non-linear solvers, and provides 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. Coopr is hosted both as part of IBM's COIN-OR open-source software initiative for operations research [10] and at Sandia, and it is actively developed and maintained by Sandia and its collaborators.

The remainder of this paper is organized as follows. Section 2 describes the motivation and design philosophy behind Pyomo, and Sect. 3 discusses why Python was chosen for the design of Pyomo. Section 4 describes related high-level modeling approaches, and Pyomo is briefly contrasted with other Python-based modeling tools.

In Sect. 5, we discuss the novelty and impact of Pyomo, answering the related questions "Why another AML?" and "Why Pyomo?". Section 6 describes the use of Pyomo on a simple application that is also described in AMPL for comparison, introduces a number of advanced features of Pyomo, and discusses Pyomo run-time performance. Section 7 describes how Pyomo leverages the broader solver capabilities in Coopr. Section 8 illustrates the use of Pyomo by discussing a number of advanced case studies. Section 9 provides information for getting started with Coopr, and Sect. 10 describes future work that is planned for Pyomo and Coopr.

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 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 optimization models. Second, projects with external customers frequently require that optimization 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 optimization modeling capability. Although open source optimization solvers are widely available in packages like COIN-OR [10], comparatively 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 proprietary software [4,56].
– *Flexible licensing* A variety of significant operations research applications at Sandia have required the use of a modeling tool with a non-commercial and non-infectious 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 cost of deploying an application model internally within their company). The Coopr software library, which contains Pyomo, is licensed under the BSD [9]. BSD has fewer restrictions for commercial use than alternative open source licenses like the GPL [24], and is non-infectious.

The use of an open source software development model is not a panacea; ensuring high reliability of the software still requires careful management and a committed developer community. However, there is increasing recognition that open source software provides many advantages beyond simple cost savings [11], including supporting open standards and avoiding being locked into a single vendor.

2.2 Customizable capability

A key limitation of proprietary commercial modeling tools is their limited support for customization of the modeling components 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 they 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 software 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 linking). In 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 Abstract models and concrete instances

A requirement of Pyomo's design is that it support the definition of abstract models in a manner similar to that of AMPL [3,19] and AIMMS [2,47]. An abstract model separates the declaration of a model from the data used to generate a specific model instance; the advantages of this approach are widely appreciated, e.g., see [18, p. 35]. This separation provides an extremely flexible modeling capability, which has been leveraged extensively in optimization applications developed 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 sources using construction routines that are provided by the user when defining sets, parameters, and other modeling components. Further, Pyomo is designed to use data specified in the AMPL format to facilitate the translation of models between AMPL and Pyomo.

Pyomo can also create concrete instances directly from specific parameter data, bypassing the abstract modeling layer. This functionality is similar to that provided in the open-source modeling tools PuLP and PyMathProg, in addition to capabilities in many commercial AMLs.

2.5 Flexible modeling language

Another goal of Pyomo is to directly use a modern high-level programming language to support the definition of optimization models. In this manner, Pyomo is similar to tools like FlopC++ [17] and OptimJ [36], which support modeling in C++ and Java, respectively. The use of a broad-based high-level programming language to develop optimization models has several advantages:

- *Extensibility and robustness* A widely used high-level programming language provides a robust foundation for developing and solving optimization 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 functional routines that can be immediately leveraged in the modeling process. Further, support of the 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 conceivable 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 uniformly available in commercial AMLs.

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 between Microsoft Windows, Linux, and Macintosh 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 [44] 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 it has a liberal open source license that allows users to 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 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 and 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 and file system access.
- *Extensibility and customization* Python has a simple model for loading Python code developed by a user. Additionally, compiled code packages (e.g., NumPy and SciPy) 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, 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:

- *.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 based in part 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 optimization 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++ [17] 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 [38]. 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 [45,53]. It is widely acknowledged that Python's dynamic typing and compact, concise syntax facilitates rapid software development. Thus, it is not surprising that Python is widely used in the scientific community [34]. Large Python projects like SciPy [30] and SAGE [50] 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.

AMLs are an alternative approach that allows applications to be interfaced with optimizers that can exploit problem structure. AMLs are specialized high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [31]. AMLs like AIMMS [2], AMPL [3,19], and GAMS [21] provide programming languages with an intuitive mathematical syntax that support 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++ [17] and OptimJ [36] can be used to formulate and solve optimization models in C++ and Java, respectively.

A related strategy is to use a high-level programming language to formulate optimization models, which are then solved with optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating and manipulating 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 [52] is probably the most mature optimization software using this approach. Python has also 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 [5,32].
- *CVXOPT* A package for convex optimization [14].
- *PuLP* A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [40].
- *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) [41].
- *OpenOpt* A numerical optimization framework that is closely coupled with the SciPy scientific Python package [35].

Pyomo is closely related to APLEpy, PyMathProg, PuLP, and POAMS. All of these packages define Python objects that can be used to express optimization models, but they can be distinguished according to the extent to which they support abstract models. Abstract models provide a data-independent specification of a mathematical program. Fourer and Gay [19] summarized the advantages of abstract models when presenting AMPL:

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

PuLP and PyMathProg do not support abstract models; the concrete models that can be constructed by these tools are driven by explicit data. APLEpy supports symbolic definitions of set and parameter data, but the objective and constraint specifications are concrete. POAMS and Pyomo support abstract 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 an abstract model. Hart [26] provides Python examples that illustrate the differences between PuLP, POAMS and Pyomo.

5 Why Pyomo: the value proposition

Before introducing Pyomo and subsequently illustrating advanced capabilities and scripting features, we first discuss the "value proposition" for Pyomo. In other words, we will address the related questions of "Why another AML?" and "Why Pyomo in particular?" In doing so, we argue that Pyomo holds a unique position in the field of both commercial and open-source AMLs. In what follows, we provide forward references to subsequent sections containing examples that support our arguments. Our immediate goal in this section is simply to outline the arguments.

The following features are what we view as the primary benefits of Pyomo:

– *Open-source with flexible licensing* As we argued in Sects. 2 and 3, a flexible open-source license is often critical when deploying real-world applications. Further, open-source software is customizable and typically more extensible than commercial alternatives. Pyomo is obviously differentiated from commercial AMLs in this regard, and it is differentiated from open-source AMLs such as PyMathProg that use restrictive licenses.
– *Embedded in a high-level, full-featured programming language* Commercial AMLs are embedded in proprietary languages, which typically lack the spectrum of language features found in modern high-level programming languages, e.g., functions, classes, and advanced looping constructs. For example, the lack of functions in AMPL makes programming many of the types of advanced applications shown in Sect. 8 very difficult. Other open-source AMLs share this advantage with Pyomo.
– *Access to extensive third-party library functionality* By embedding Pyomo in Python, users immediately gain access to a remarkable range of powerful and free third-party libraries, including: SciPy, NumPy, Matplotlib, Pyro, and various database / spreadsheet interfaces.
– *Support for abstract and concrete math programming models* As we argued in Sect. 2.4, the separation of model from data is extremely useful in practice. While commercial AMLs support this distinction, Pyomo is the only open-source AML to fully support the specification of both abstract and concrete math programming models.
– *Support for non-linear math programming models and solvers* While many commercial AMLs provide capabilities to express and solve non-linear programs, few open-source tools support non-linear capabilities; OpenOpt [35] is a notable exception. This Pyomo capability is discussed in Sect. 6.4.
– *Integrated support for distributed computation* Pyomo provides integrated support for distributed computation (see Sect. 7.2) by leveraging and extending capabilities in both Python and the Python library Pyro. This capability is unique in the context of open-source AMLs, and is uncommon in commercial AMLs. Support for distributed computation is becoming increasingly critical for application deployment, given the growing availability of cluster-based and cloud computing.
– *Cross-platform deployment capabilities* By embedding Pyomo in Python, we are able to rapidly deploy applications in cross-platform environments. Examples of such environments are discussed in Sect. 7.2, drawn from real-world computing

environments exercised by the co-authors and their collaborators. Similar capabilities could be embedded in related open-source AMLs such as PuLP and APLEpy, but are not currently available.

– *Integrated support for obtaining data from external sources* Pyomo provides a rich range of interfaces to extract data from structured text files, spreadsheets, and databases, driven by the requirements for real-world deployment of optimization applications. As discussed in Sect. 6.5, this capability distinguishes Pyomo from other open-source AMLs.
– *Extensibility via component-based software architecture* Drawing on best practices in commercial software design, Pyomo and Coopr are designed in a modular, component-based fashion. In particular, users can provide extensions without impacting the core software design, e.g., in the form of customized solvers and problem writers. Such functionality is discussed in Sect. 7.3, along with an example of a simple custom solver. The ability to seamlessly extend the core functionality is unique among open-source and commercial AMLs.
– *Advanced application scripting* Perhaps the most important feature of Pyomo is the comparable ease with which applications requiring advanced scripting can be developed. For complex optimization applications—beyond those requiring a simple "express model, generate model, solve model" capability, such scripting is often integral to successful deployment. Of course, it is difficult to quantify "ease" in this context, and we make no attempt to do so here. Rather, we point to specific examples illustrated in Sect. 8.

6 Pyomo: an overview

Pyomo can be used to define abstract models, create concrete problem instances (both directly and from abstract models), and solve those instances with standard solvers. Pyomo can generate problem instances and apply optimization solvers within a fully expressive programming language. 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.

6.1 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. We focus here on the specification of symbolic or abstract models; concrete or directly specified models are discussed in Sect. 6.3.

Consider the following AMPL model `prod.mod`, which is available from http://www.ampl.com/BOOK/EXAMPLES/EXAMPLES1:

```
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 AMPL model into Pyomo, the user must first import the Pyomo Python package and create a Pyomo `AbstractModel` object:

```
# Imports
from coopr.pyomo import *

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

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 virtual Python executable installed by the Coopr installation script described in Sect. 9 automatically includes all requisite paths.

Next, we create the sets and parameters that correspond to the data declarations used in the AMPL model. This can be done very intuitively using the `Set` and `Param` classes defined by Pyomo:

```
# 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 and co-exist within a single application.

Next, we define the decision variables in the model using the Pyomo `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 can be defined in a separate data file that is processed by Pyomo (see Sect. 6.5), similar to the paradigm used in AMPL and AIMMS.

Objectives and constraints are explicitly defined expressions in Pyomo. In abstract models, the `Objective` and `Constraint` classes require a `rule` keyword option that specifies how these expressions are to be constructed. A rule is a function that takes one or more arguments and returns an expression that defines the constraint (body and bounds) or objective (expression). The last argument in a rule is the model containing 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:

```
# Objective
def objective_rule(model):
    return summation(model.c, model.X)
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 using standard Python language constructs. The function `objective_rule` returns an algebraic expression that defines the objective, and the function `time_rule` returns a less-than-or-equal expression that defines an upper bound on the constraint body. They are created with Pyomo's `summation` function, which concisely specifies a vector sum of one or more arguments. An alternative is to use the Python `sum` function. So, for example, the sum over i in `model.P` of X_i/a_i could be done using the `summation` function as shown or using `sum(model.X[j]/model.a[j] for j in model.P)`, which has the same effect. The function `limit_rule` 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 an abstract 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 an abstract model needs to be instantiated with data to generate the variables, constraints, and objectives. An instance of the `prod` model can be generated and displayed as follows:

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

The file `prod.dat` file contains `set` and `param` data commands that are consistent with AMPL data commands; this example file is also available from http://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:

```
from coopr.opt import *
opt = SolverFactory("cplex")
results = opt.solve(instance)
```

This code fragment imports the solver interfaces associated with Pyomo, and creates an optimizer interface 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` option indicates the maximum number of solutions from the results object to be written; some solvers (including CPLEX) can return multiple solutions that represent alternative optima, or other feasible points.

6.2 Advanced Pyomo modeling features

The previous example provides a simple illustration of Pyomo's symbolic 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 specifies the dimensionality of set or parameter data, e.g., as follows:

```
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 constructed 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:

```
def x_init(model):
    return [2*i for i in range(0, 10)]
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 as follows:

```
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(validate=p_valid)
```

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

– `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 complex 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)
```

6.3 Constructing concrete models

Concrete models are the most direct type of model that Pyomo supports. These models can be formulated in the same way as abstract models using Pyomo components for variables, objectives, and (optionally) constraints. Additionally, a user can easily leverage native Python data structures while constructing concrete models. The `ConcreteModel` class is used to represent concrete models whose data is provided as the model components are declared. In contrast to `AbstractModel`, a `ConcreteModel` immediately initializes model components as they are added to a instance.

6.4 Non-linear modeling extensions

A key differentiating feature of Pyomo relative to other open-source (and some commercial) modeling languages is the ability to specify and generate non-linear optimization models. Currently, the non-linear model output format supported by Pyomo is NL, the native AMPL file format. We have chosen this format principally because it is used by a variety of non-linear solvers, including Ipopt [29] and BONMIN [8]. Further, the NL format supports the full range of non-linear operators. Examples of such operators, also provided in Pyomo, include:

– `pow`, `exp`, and related operators (including `sqrt` and `log10`)
– `sin`, `cos`, `tan`, and related hyperbolic and inverse operators
– `abs`, the absolute value operator

To illustrate the use of non-linear operators in Pyomo, consider the objective associated with the `brownden` CUTEr test problem instance [13]:

```
def f(model):
    expa = sum([((model.x[1]+model.t[i]*model.x[2]- \
                exp(value(model.t[i])))**2 + \
                (model.x[3]+model.x[4]*sin(value(model.t[i])) - \
                cos(value(model.t[i])))**2 )**2 for i in model.St])
    return expa
model.f = Objective(rule=f,sense=minimize)
```

As the example illustrates, non-linear operators are naturally incorporated into the syntax for specifying algebraic expressions in both constraints and objectives.

6.5 Importing data

While commercial AMLs provide streamlined functionality for accessing external data sources, the same is not true of open-source AMLs. As discussed below in Sect. 6.7, Python-based AMLs other than Pyomo require the user to specify data via native Python constructs, e.g., in dictionaries or low-level input/output routines. The same holds for non-Python open-source AMLs such as FlopC++. While flexible, this interface can be cumbersome and inefficient for non-programmers. In the case of abstract models, where data varies across a single core abstraction, users must maintain independent data stores for each instance. Further, in deployed optimization applications, data is almost exclusively maintained in central data stores, e.g., Excel or a database. To address these concerns, Pyomo provides streamlined initialization of data from external data sources.

The data file format discussed in Sect. 6.1 is one example of an external method by which Pyomo data can be initialized, and this is the most common mechanism for novice and academic Pyomo users. These files contain custom Pyomo data commands that are similar to those used by AMPL.

To support data access from more general sources such as structured ASCII file, csv files, spreadsheets, and databases, Pyomo supports the `import` command. The `import` command directs Pyomo to load a *relational table* that represents set and parameter data. A relational table consists of a matrix of numeric string values, simple strings, and quoted strings. All rows and columns must have the same number of entries, and the first row represents labels for the column data.

We now discuss some aspects of the Pyomo `import` syntax using small illustrative examples. First, consider data stored in a simple text white-space delimited text file called `Y.tab`:

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

This file specifies the values of a parameter Y, which is indexed by set A. The following `import` command loads the parameter data:

```
import Y.tab : [A] Y ;
```

The first argument is the name of the file containing the data. The options after the "**:**" character indicate how the table data is mapped onto model data. The option `[A]` indicates that the set `A` is used as the index, and the option `Y` indicates the name of the parameter to be initialized.

Similarly, set data can be loaded from an ASCII file named `A.tab` containing a single column:

```
A
A1
A2
A3
```

The `format` option must be specified to denote the fact that the relational data is to be interpreted as a set, as follows:

```
import A.tab format=set : A ;
```

Simple extensions of this general syntax can be used to specify data tuples, or indexed sets; additional keywords are provided to guide how relational table data is interpreted by Pyomo. The same general syntax can be used to access relational table data in csv (comma-separated) and xls (Excel) files. Similarly, data can be extracted from relational databases by adding appropriate keyword specifying the database driver, query, and optional user name and password, for example:

```
import ABCD.mdb using=pyodbc query="SELECT * FROM ABCD" : \
                 Z=[A,B,C] Y=D ;
```

This command creates a relational table using all of the columns in the database table `ABCD`, extracted using the `pyodbc` database interface. The `using` keyword indicates that data is to be extracted using an external Python package, the Python ODBC database interface.

Other data partitioning and organizational commands are also supported in Pyomo, e.g., `include` statements that import nested. dat files and `namespace` commands that logically partition the contents of a single .dat file. Namespaces are particularly useful, allowing for the specification of data for multiple model instances in a single file.

6.6 The Pyomo command

While Pyomo-based Python code can be entered and executed directly from within the Python interpreter, Pyomo includes the `pyomo` command-line tool that can construct an abstract or concrete model, create a concrete instance from user-supplied data (if applicable), apply an optimizer, and summarize the results. For example, the following command line optimizes the AMPL `prod` model using the data in `prod.dat`:

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

The `pyomo` command automatically executes the following steps:

- Create an abstract or concrete model
- Read the instance data (if applicable and specified)
- Generate a concrete instance from the abstract model and instance data (if applicable)

– Apply simple preprocessors to the concrete instance
– Apply a solver to the concrete instance
– Load the results into the concrete instance
– Display results

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 obtained 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.

6.7 Comparison with other Python-based modeling languages

We now briefly contrast Pyomo with several open-source Python-based math program modeling packages providing related functionality. A comprehensive comparison is beyond the scope of this paper. Instead, our goal here is to briefly outline the primary differences in the design and functionality of these packages. We focus first and in most detail on a comparison of Pyomo and PuLP, as the latter is arguably the most widely used open-source AML.

PuLP The PuLP package [40] is a widely used, light-weight Python package for expressing and solving optimization problems. The biggest functional differences between PuLP and Pyomo are: (1) PuLP can only express concrete model instances, (2) PuLP only allows for expression of linear and mixed-integer models, and (3) PuLP provides no built-in mechanism (other than what is available through Python) to load and save model data from external sources, e.g., text files, spreadsheets, and databases. As discussed previously, the availability of these features is either critical for or extremely useful in the deployment of solutions to real-world applications.

For concrete model specification, the PuLP and Pyomo syntax are qualitatively similar. However, some differences are worth briefly highlighting. Consider the following PuLP model:

```
from pulp import *
prob = LpProblem("A small concrete model",LpMaximize)
x1 = LpVariable("x1",5, 10) # 5 <= x1 <= 10
x2 = LpVariable("x2",10, 30) # 10 <= x1 <= 30
prob += x1+x2, "The Objective"
prob += 0.5*x1 + 1.5*x2 <= 50, "The Constraint"
```

In this example, the variables x1 and x2 are not explicitly associated with the model. They can be accessed by scanning the set of model constraints, but this is in practice expensive. In Pyomo, all model components (parameters, sets, variables, etc.) are consistently and explicitly associated with a specific model, and can be directly accessed by querying the model in a clean, object-oriented fashion.

The mechanisms for expressing indexed components—in particular variables and constraints—are more natural and flexible in Pyomo than in PuLP. Indexed variables in PuLP are specified by creating a dictionary (mapping of keys to values) of variables, e.g., as follows:

```
index_a = [1, 2, 3]
index_b = [4, 5, 6]
indexed_var = LpVariable.dicts('my_vars' , \
            (index_a, index_b), 0, 1, LpInteger)
```

This construct is expressed in Pyomo as follows:

```
index_a = [1, 2, 3]
index_b = [4, 5, 6]
model.indexed_var = Var(index_a, index_b, within=Binary)
```

One minor limitation of the PuLP indexed variable syntax is that variable lower and upper bounds are assumed to be homogeneous. In contrast, Pyomo provides for index-specific bounds by supplying an arbitrary function via the `bounds` keyword in the `Var` constructor.

Next, we consider an example in which constraints are imposed on a variable value on a per-index basis. In PuLP, an example of such a constraint is as follows:

```
for x in index_a:
   for y in index_b:
      prob += indexed_var[x,y] <= \
         some_param[x,y] * other_var[x,y], " "
```

Note that the looping constructs must be manually specified, and that the constraints are not logically grouped within the model. In Pyomo, this constraint would be expressed as follows:

```
def some_rule(model, x, y)
   return model.indexed_var[x,y] <= \
         model.some_param[x,y] * model.other_var[x,y]
model.my_constraint = Constraint(index_a, index_b, rule=some_rule)
```

Contrasting the two approaches, we observe that the related constraints are automatically grouped in Pyomo, and the explicit (and potentially error-prone) looping constructs in PuLP are avoided. Similarly, constraint names are assigned automatically in Pyomo (e.g., `my_constraint[1,3]`), whereas the PuLP looping constructs require the user to explicitly form the constraint name (which is left unspecified for this reason in our example).

PuLP is actively supported, and is distributed under an open, BSD-like license. PuLP was recently accepted into the COIN-OR project.

APLEpy The APLEpy package [5] is similar to PuLP in both design and functionality, although there are some differences worth highlighting. In contrast to PuLP, all model components are implicitly members of a single, globally accessible model object. As a consequence, maintaining and manipulating multiple model instances (e.g., as shown for Pyomo in Sect. 8) is not possible. The syntax for constructing constraints is very similar to that of PuLP. For example, explicit user looping is required in the case of indexed constraints. Like PuLP, APLEpy only supports concrete model specification, provides no integrated facilities for accessing external data, and does support specification of indexed variables and constraints. Although still available for download, APLEpy does not appear to be actively supported. APLEpy is distributed under the Common Public License.

PyMathProg and POAMS Other than PuLP and APLEpy, POAMS and PyMathProg are the two most similar open-source AMLs to Pyomo. Unfortunately, POAMS is not widely available for download and experimentation (there has been no formal release). PyMathProg [41] is associated with the GNU GLPK project, and as a consequence, only interfaces to the GLPK solver. Models are specified in a PuLP-like syntax; no functionality for specifying abstract models is provided, nor are integrated interfaces to external data sources. PyMathProg is distributed under the GPL license.

Numberjack and Google OR-Tools A number of Python modeling and solver packages have recently originated from the constraint programming community, in particular Google OR tools [37] and Numberjack [28]. The goals of Numberjack are broader than Pyomo and related AMLs from the mathematical programming community, in that the objective is to support specification and solution of constraint programming, mixed-integer linear programming, and boolean satisfiability models. Consequently, a different set of solvers is supported, e.g., the SCIP mixed-integer solver, the Mistral constraint programming solver, and the MiniSat satisfiability solver. Like the other open-source AMLs discussed above, models may not be specified symbolically, and there is no integrated support for data input from external sources such as Excel. Non-linear operators for mathematical programming are not supported. The Numberjack syntax is very similar to that of PuLP, e.g., models are constructed using the += syntax and variables are specified externally to a model object. Numberjack is distributed under the LGPL. The solver interfaces in Numberjack are similar in design spirit to those in Coopr. In contrast to Numberjack, Google's OR-tools package is primarily focused at the present time on specifying and solving constraint programs, and supports a single integrated solver. Like Numberjack, the OR-tools package does not allow specification of abstract models, and lacks integrated support for obtaining data from external data sources. Google's OR-tools package is distributed under the Apache license, while Numberjack is licensed under the LGPL.

6.8 Run-time performance

Run-time performance was not the most significant factor in our choice of Python as a modeling language, but it is clearly an important factor for many Pyomo users. Although the optimization solver run-time is typically dominant, in some applications the time required to construct complex models can be nontrivial. Thus, we have simplified and tuned the behavior of Pyomo modeling components in an attempt to minimize the run-time required for model construction. This tuning effort is on-going, driven by experience with specific test applications.

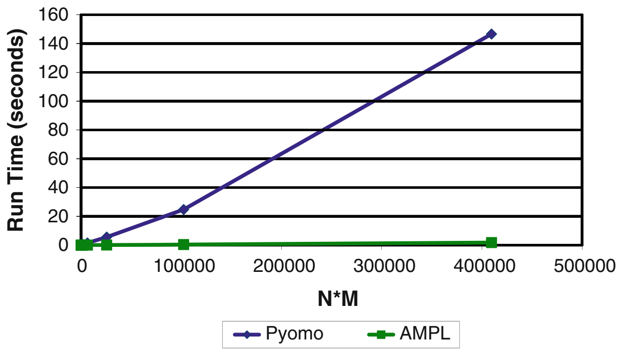

Fig. 1 The real (wall clock) run time required to construct and write dense p-median model instances to an NL file with AMPL and Pyomo

Figure 1 shows the run-time performance for Pyomo and AMPL on dense instances of the well-known p-median facility location problem. We have considerable experience solving large-scale p-median problem instances arising in water security contexts [27], motivating our investigation of run-time scaling using this particular optimization problem. The two curves in this plot show the time, in seconds, that was required to generate and write (but not solve) a model with AMPL and Pyomo. In both cases, we write the resulting mixed-integer program in the NL problem format, i.e., the preferred format in AMPL. The x-axis captures the problem size $N \cdot M$ for p-median problems where the number of customers, N, equals the number of facilities, M. Each point in this plot represents the average of 15 different trials, executed on a modern (3.33 GHz dual-processor Intel Xeon ($\times$5680) with 24GB RAM and no hypher threading) multi-core workstation running Linux. This graph was generated with the version of Pyomo that is included in the Coopr 3.0 release.

This figure shows that there is a substantial performance gap between Pyomo and AMPL for large p-median instances. For the largest problems, Pyomo with garbage collection enabled (the default) was 82 times slower than AMPL. This difference largely stems from the fact that Python is notoriously slow at constructing and destructing large numbers of small objects, e.g., such as parameter and variable values appearing in expressions. Constraint and objective expressions in Pyomo are encoded as trees, which represent each arithmetic operator and data value with a distinct Python object. Thus, constructing large expression trees involves the construction of many Python objects, yielding the observed performance differences. We are currently exploring alternative encodings of expression trees, specifically to avoid this source of performance degradation. Other sources of performance bottlenecks include string (component label) manipulation, expression preprocessing in preparation to write the NL file, and expression verification and simplification. Our analysis of Pyomo profiler output suggests that these bottlenecks can also be mitigated with a moderate development effort. Python performance can also be improved by using the Psyco JIT compiler [39], but this tool can only be used on 32-bit versions of Python on 64-bit computers. Finally, we observe that the performance discrepancies vary significantly, depending strongly on the model structure and the comparative baseline. For example, Dimitrov observed a 6× discrepancy between the performance of GAMS and Pyomo on a complex production/ transportation model [15].

7 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 [12]. Coopr utilizes a component-based software architecture to define 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 required by an optimizer, and transparent parallelization of independent optimization tasks.

7.1 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 [20]. The main difference is that Coopr represents results in YAML, a data serialization format that is both powerful and human readable [55]. For example, Fig. 2 shows the results output after solving the AMPL example production planning problem described in Sect. 6.

7.2 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 plugins for MIP solvers like CPLEX, GUROBI, 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 [16].

```
# ==========================================================
# = 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: OK
  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. 2 Results output for the production planning model described in Sect. 6

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 [43]. The `pickle` module performs object serialization, which is a pre-requisite for distributed computation. With very 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 `AbstractModel` and `ConcreteModel` instances. For example, the following code fragment illustrates the use of the `pickle` module to write and restore a Pyomo model instance via an intermediate file:

```
import pickle
instance = model_builder() # function to build Pyomo instance
pickle.dump(instance, open('tempfile', 'wb'))
instance_copy = pickle.load(open('tempfile','r'))
```

The simplicity of complex object serialization in Python is remarkable, especially relative to low-level languages like C++ in which users must develop complex class-specific methods to achieve equivalent functionality. This issue is amplified by the presence of complex inter-relationships among objects, i.e., such as those present in Pyomo models.

Pyro (Python Remote Objects) [43] 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. Interprocess 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 library 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")
```

The Pyro solver manager identifies dispatch servers through the Coopr name server. Thus, a simple change in the argument name to the solver manager factory is sufficient to access distributed solver resources in Coopr. We note that if no solver manager is specified (as is the case for examples shown early in this paper), a default `serial` solver manager is constructed automatically; this particular manager executes solves locally, in a sequential fashion.

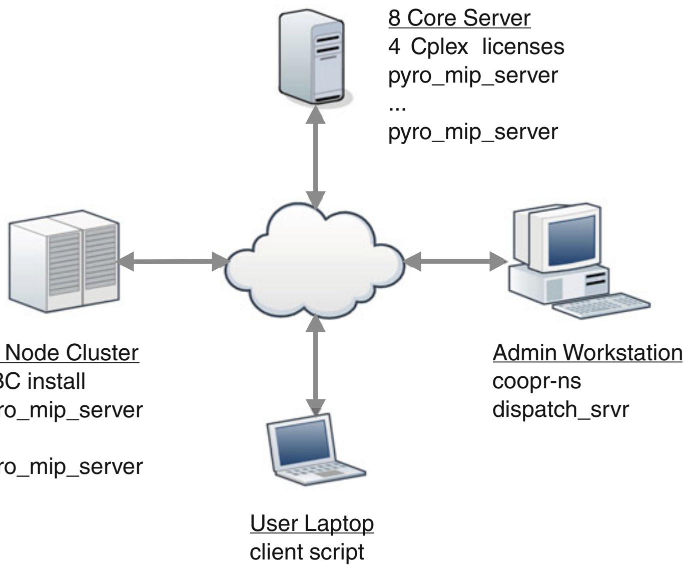

Fig. 3 An example of a hybrid compute architecture and how it can be configured for distributed solves using Pyro and Coopr

Given a Pyro solver manager, a Pyomo instance can be solved via the following code:

```
instance = model_builder() # function to build Pyomo instance
ah = solver_manager.queue(instance)
solver_manager.wait_for(ah)
results = solver_manager.get_results(ah)
```

The `ah` object is known as an action handle, and informally serves as a "tracking number" through which users can interrogate the status of a submitted solve request.

Figure 3 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 Sect. 8.2) on his or her local machine. The name and dispatch server processes (`coopr-ns` and `dispatch_srvr`) are configured as daemons on an administrative workstation, such that they are persistently available. 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 the Windows cluster has the open-source CBC solver available on all of its 64 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.

7.3 Solver plugins

A common object-oriented characteristic of open-source optimization software is the ability to use classes and class inheritance to develop extensible functionality. In contrast, Coopr leverages the PyUtilib Component Architecture (PCA) to separate the declaration of component interfaces from their implementation [46], through a mechanism referred to as a *plugin*. A plugin specifies the required method and data associated with a specific interface, but does not require that all instances of this interface be derived from a particular base class. For example, in Coopr all optimization solvers are implemented by defining a distinct class encapsulating the associated functionality. However, these classes are not required to be sub-classes of a solver interface class. Instead, they are simply required to provide the same interface methods and data.

Coopr uses plugin components to modularize the process workflow commonly associated with optimization. A component is a software package, module, or object that provides a specific functionality. Plugin components augment the standard optimization workflow 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 [48]. Component-based frameworks manage the interaction between components to promote adaptability, scalability, and maintainability in large software systems [51]. For example, with component-based software there is much less need for major releases 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 solver output files
- Problem writers, which create solver input files that specify optimization problems
- 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 provide key functionality. For example, solver plugins are available for the CPLEX, GUROBI, 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 solver input file formats; these plugins rely on binary executables built by the GLPK [23] and Acro [1] software libraries.

Figure 4 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.

```
# 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]*value(instance.N)
        for j in range(0,value(instance.P)):
            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,value(instance.M)+1):
                val += min(value([instance.d[i,j])
                              for i in range(1,value(instance.N)+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. 4 A simple customized Coopr solver for p-median problems

This plugin implements a `solve` method, which randomly generates solutions to a p-median optimization problem. The only other step required is to invoke 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. For example, if this plugin is contained within the file `solver2.py`, then the following Python script can be used to apply this solver to Coopr's p-median example model:

```
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` mixed-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 (via the `load` option) that the `solver2.py` file should be imported before optimization:

```
pyomo --solver=random --load=solver2.py pmedian.py pmedian.dat
```

Thus, users can develop custom solvers in Python modules, which can be executed and tested directly using the `pyomo` command.

This example serves to illustrate the ease with which new solver interfaces can be prototyped and deployed using Pyomo and Coopr. Similar plugins could be easily implemented for a wide range of metaheuristic or problem-specific solvers; the user simply needs to extract the necessary information from a Pyomo instance, transfer it to the specific solver, and then extract and store the results of the solve in the canonical Coopr results format. Further, such plugins can be seamlessly integrated into existing optimization workflows, e.g., via the `pyomo` command line utility.

8 Advanced scripting and algorithm development

In Sect. 6, we presented a straightforward use of Pyomo: to construct a concrete instance from an abstract 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.

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 abstractions. 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 other AMLs, by specifying various types of named sets and parameters so that the dimensions and details of the data can be separated from the specification of the model. Separation of an abstract model from the data specification is a hallmark of structured modeling techniques [22]. At the other extreme, elements of a optimization model can be treated in their fully canonical form as is supported by callable solver libraries. Methods can be written that access or manipulate, e.g., objective functions or constraints in a fully general way. This capability is a fundamental tool for general algorithm development and extension [33]. Pyomo provides the full continuum of abstraction between these two extremes to support modeling, scripting, and algorithm 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.

In the remainder of this section, we introduce a variety of example case studies highlighting the relative ease of both scripting and algorithm development in Coopr. The example in Sect. 8.1 discusses a script implementing a hybrid MIP-NLP optimization algorithm. Section 8.2 describes the Pyomo-based implementation of a Benders-based decomposition algorithm for a simple production planning problem. Together, these two examples illustrate the use of Pyomo and Coopr to solve optimization models requiring some degree of algorithmic customization, emphasizing different coding approaches to achieve similar goals. Finally, in Sect. 8.3 we introduce an example highlighting features of Pyomo and Python that facilitate the development of generic algorithms.

8.1 Advanced scripting: hybrid optimization

Hybrid methods may be required to solve particularly difficult real-world optimization problems. Due to the deviation from the standard optimization workflow process, in which a model is handed to an off-the-shelf solver, implementation of hybrid methods typically requires non-trivial scripting.

One instance of such a hybrid algorithm implemented in Coopr/Pyomo was developed by our collaborators in Texas A&M's Department of Chemical Engineering. The specific problem of interest involved the development of a global optimization algorithm to solve a parameter estimation problem arising in the context of a model for childhood infectious disease transmission. For further information regarding both the model and its Pyomo implementation we refer to Hackebeil and Laird [25]. Below, we briefly survey the high-level solution strategy, and highlight key fragments of Pyomo code implementing the strategy.

The parameter estimation model considered is a difficult non-convex, non-linear program for which no efficient solution algorithm exists. Consequently, to facilitate tractable solution, the problem is reformulated using a MIP under-estimator and an NLP over-estimator. Information is exchanged between the two formulations, and the process is iterated until the two solutions converge.

The main script for this process is implemented in Pyomo as follows; some initialization code is omitted for clarity:

```
data_file = "disease_data.dat"
results_file = "global_opt_results"

# the function "initialize_dicts" is a utility specific to
# this example, which extracts data from the input file and
# various input arguments (not shown), and returns two Python
# dictionaries.
mdl_inputs,data_inputs = initialize_dicts(data_file, ....)

for i in range(1,MAX_ITERS+1):

   # define the full optimization model for this iteration.
   # data is significantly changing each iteration...
   mstr_mdl = disease_mdl(mdl_inputs,data_inputs)

   # create and solve MIP over-estimator.
   inst, MIP_results = solve_MIP(mstr_mdl,MIP_options)

   # create and solve the NLP under-estimator.
   inst, NLP_results = solve_NLP(mstr_mdl,MIP_results, \
                                 NLP_options)

   # load results, report status, and compute the gap/ub.
   GAP, UB = output_results(inst, ..., MIP_results, \
                            NLP_results, ...)

   # use results to determine parameters for the next
   # iteration, via updates to the "mdl_inputs"
   # dictionary.
   mdl_inputs, POINTS_ADDED = update_points(mdl_inputs,inst, \
                                            ... MIP_results)

   if (UB != None) and (i == 1):
     # perform solves to strengthen model.
     mdl_inputs = tighten_bounds(inst,mdl_inputs,data_inputs,\
                                 UB, num_lb_points,MIP_options)

   if (POINTS_ADDED == 0) or (GAP <= MAX_GAP):
      break
```

In this example, user-defined Python functions are used to organize and modularize the optimization workflow. In contrast to examples shown previously, the optimization model in this case is constructed via a function (`disease_model`). A fragment of code for this function is as follows:

```
def disease_mdl(INPUTS, DATA):

    model = ConcreteModel()

    # attach non-Pyomo data values to the model instance.
    model.pts_LS = INPUTS['LSP']
    model.pts_LS_lower = INPUTS['LSPL']
    model.pts_LI = INPUTS['LIP']
    model.pts_LI_lower = INPUTS['LIPL']

    model.TIME = Set(ordered=True, initialize=DATA['TIME'])
    model.BIRTHS = Param(model.TIME, initialize=DATA['BIRTHS'])

    # more parameter and set definitions...

    model.logS = Var(model.TIME, bounds=logS_bounds_rule)
    model.logI = Var(model.TIME, bounds=logI_bounds_rule)

    # more model variables...

    model.obj = Objective(rule=obj_rule)
    model.pn_con = Constraint(model.TIME, rule=pn_rule)

    # more model constraints ...

    # automatically generate, via a function, additional
    # constraints associated with linearization and add
    # them to the model...
    linearize_exp(model.TIME, model.S, model.logS, \
                  model.pts_LS, model.pts_LS_lower)
    linearize_exp(model.TIME, model.I, model.logI, \
                  model.pts_LI, model.pts_LI_lower)

    return model
```

Beyond the standard Pyomo model constructs, we observe the ability to dynamically augment Pyomo models with arbitrary data, e.g., the definition of the `pts_LS` and `pts_LI` attributes; these are not model components, but rather raw Python data types. Pyomo itself is unaware of of these attributes, but other components of a user-defined script can access and manipulate these attributes. Such a mechanism is invaluable when information is being propagated across disparate components of a complex, multi-step optimization process. We note the use of an auxiliary function (`linearize_exp`, introduced to modularize the code) to construct specific classes of constraint, which are then added to the input Pyomo model.

Following instance construction, MIP and NLP variants of the master model instance are solved using the user-defined utility functions `solve_MIP` and `solve_NLP`. These functions simply activate the relevant model components (all model components in Pyomo can be independently activated and deactivated), solve the corresponding model, and return the results. The function `output_results` reports and caches results, and computes the current gap and upper bound. Finally, the functions `update_points` and `tighten_bounds` perform computations (obtained both analytically and via MIP solves) that yield the input data required for the next iteration of the process.

This example serves to illustrate, at a very high level, the scripting of a complex hybrid optimization algorithm using Pyomo and Coopr. Despite the complexity of the process (hidden in large part due to code modularization), the code (including that for the model definition) is relatively compact—a total of approximately 650 lines of Python code, including white-space. The full code is available upon request by contacting one of the authors.

8.2 Benders decomposition

To further illustrate advanced scripting in Pyomo, we next consider the translation of an AMPL example involving the solution via Benders decomposition of a production planning problem formulated as a stochastic linear program. This example emphasizes different aspects of Pyomo and Coopr than the hybrid optimization discussed in Sect. 8.1, specifically in terms of how models are defined, manipulated, and solved.

The production planning problem considered is defined as follows. 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 revenue scenarios. 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 general two-stage stochastic linear programs can be solved via Benders decomposition.

The original 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 abstract models, mirroring the process previously documented in Sect. 6; 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. In contrast, Pyomo requires that distinct models be self-contained in their definition.

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 Sect. 6, the `runbenders` script begins by loading the necessary components of the Pyomo, Coopr, PyUtilib, and Python system libraries:

```
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 abstract 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"))
```

In this code fragment, the `master.py` and `subproblem.py` model definition files are loaded (via the PyUtilib function `import_file`), defining the associated abstract models; the `runbenders` script accesses the corresponding `model` objects in the respectively Python modules. Given an abstract 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, as it provides the variable and constraint suffixes needed for the Bender's procedure (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 Sect. 7.2 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, and 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. The parameters are indexed by sets whose declaration is not shown here: `PROD` is the set of products, `SCEN` is the set of scenarios, and `TWOPLUSWEEKS` is the set of weeks beginning with week 2.

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 function `solve_all_instances` is a Coopr utility 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 Sect. 6.1. 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 method arguments 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, involving a simple looping construct and checks for convergence. This example further 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.

8.3 Generic algorithms: generating extensive forms of stochastic programs

Moving beyond model-specific scripting, we now briefly discuss capabilities of Pyomo and Python that, when used in conjunction, can be used to develop generic, model-independent algorithms. For purposes of illustration, we consider the case of stochastic linear and mixed-integer programs. A stochastic program can be viewed in terms of a scenario tree, i.e., a tree representing the evolution of model parameter uncertainty. To simplify exposition, we consider a two-stage stochastic program in which a single root node has n children, each of which represents a leaf node. Associated with each node is a set of variables. The variables corresponding to the root node are so-called first-stage variables, and represent decisions that must be made before the actual realization of uncertainty is revealed. The values of variables associated with the leaf nodes (also known as second-stage variables) are determined once the corresponding parameter uncertainty is revealed. For an in-depth introduction to stochastic programming, we defer to Shapiro et al. [49].

Conceptually, an n-scenario stochastic program can be viewed as a collection of n independent deterministic mathematical programs (each representing a single scenario), with one key augmentation: so-called *non-anticipativity* constraints are added to ensure equality of the first-stage decision variables across all n scenarios. Non-anticipativity is required to avoid prescient decision-making. This conceptualization is useful in practice, in part because it facilitates the transition from deterministic to stochastic programs, and further because many decomposition-based solution strategies explicitly operate on such a conceptualization. We now briefly discuss the implementation of a generic algorithm for constructing the mathematical program associated with this conceptualization, which is known as the *extensive form*. Further details concerning stochastic programming in the context of Pyomo and Coopr can be found in Watson et al. [54], which describes the PySP module of Coopr.

To generate the extensive form of any stochastic program using Pyomo/Coopr, the first step is to construct the individual scenario model instances. This can be accomplished with any of the approaches detailed above in Sects. 8.1 and 8.2. The more difficult step is to construct the non-anticipativity constraints, in a generic and model-independent fashion. To understand the underlying programmatic mechanism, we first discuss how scenario trees are specified by users.

In PySP, there is a canonical Pyomo model of the scenario tree structure, specified as follows:

```
scenario_tree_model=AbstractModel()

scenario_tree_model.Stages=Set(ordered=True)
scenario_tree_model.Nodes=Set()

scenario_tree_model.NodeStage=Param(  \
            scenario_tree_model.Nodes, \
            within=scenario_tree_model.Stages)
scenario_tree_model.Children=Set(scenario_tree_model.Nodes, \
            within=scenario_tree_model.Nodes, ordered=True)
scenario_tree_model.ConditionalProbability=Param( \
            scenario_tree_model.Nodes)

scenario_tree_model.Scenarios=Set(ordered=True)
scenario_tree_model.ScenarioLeafNode= Param( \
            scenario_tree_model.Scenarios , \
            within=scenario_tree_model.Nodes)

scenario_tree_model.StageVariables=Set( \
            scenario_tree_model.Stages)
scenario_tree_model.StageCostVariable=Param( \
            scenario_tree_model.Stages)
```

A key observation concerning this model is that all set and parameter values are *strings*, i.e., names given by the user to various objects. Of particular interest with respect to generation of a stochastic program extensive form is the definitions of the set `StageVariables`. For the well-known Birge and Louveaux "farmer" example (see [7]), the corresponding data are specified as follows:

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

This data indicates that all indices of the variable `DevotedAcreage` are first-stage decision variables, for which non-anticipativity constraints must be constructed. In terms of implementation, the three key mechanisms required are (1) identification of the relevant `Var` objects on each Pyomo scenario instance, (2) construction of a "master" variable corresponding to each such object, and (3) construction of the non-anticipativity constraints enforcing equality between the master variable and each of the corresponding scenario instance variables. The following code fragment illustrates the implementation of these three steps; we have intentionally omitted various error-checking mechanisms (vital in the real PySP implementation), in addition to the code translating the scenario tree information above into a traversable, object-oriented structure:

```
ef_instance = ConcreteModel()

# the "stage" object is a component of an object-oriented
# encoding of the scenario tree structure. the code below is
# executed for each stage, although we only show a single
# stage for compactness.

for stage_reference_variable in stage._variables:

  # for all but the last stage...
  for tree_node in stage._tree_nodes[:-1]:

    stage_variable_name = stage_reference_variable.name
    stage_variable_indices = \
           tree_node._variable_indices[stage_variable_name]

    ef_variable_name = ... # some unique name

    ef_variable_index = getattr(reference_instance, \
           stage_variable_name).index
    ef_variable = Var(ef_variable_index,name=ef_variable_name)
    setattr(ef_instance, ef_variable_name, ef_variable)

    for index in stage_variable_indices:

      for scenario_instance in tree_node._scenarios:

        scenario_variable = getattr(scenario_instance, \
            stage_variable_name)
    def makeEqualityRule(ef_variable, scenario_variable, \
                         index):
      def equalityRule(model):
        return (0.0, \
          ef_variable[index] - scenario_variable[index], \
          0.0)
      return equalityRule

      ef_constraint_name = ... # some unique name
      ef_constraint =Constraint(name=ef_constraint_name, \
        rule=makeEqualityRule(ef_variable, \
          scenario_variable, index))
      setattr(ef_instance, ef_constraint_name, \
          ef_constraint)
```

The most salient observation regarding this code is its compactness, which is facilitated by a combination of Pyomo and Python functionality. In particular, the Python function `getattr` function is used to extract attributes of objects *by name*, at run-time—something that is not possible in statically typed languages such as C++. This capability, known as introspection, is used in the above fragment to access the `Var` objects on each Pyomo scenario instance in a completely generic and model-independent manner. Other features worth noting include (1) the ability to clone `Var` objects using index sets (as occurs when the `ef` variables are constructed), (2) the use of the Python function `setattr` to add components to the extensive form model instance, and (3) the use of in-line function definitions to construct the expressions associated with the non-anticipativity constraints.

The example above serves to illustrate the use of Pyomo/Coopr in the role of algorithm development, as opposed to just simple scripting. In particular, generic algorithms can be constructed with relatively little code. For example, the full construction routine for the extensive form in PySP is approximately 300 lines (including all white-space and error checking). Similarly, the core code for a complex decomposition solver strategy (Progressive Hedging) is approximately 1,000 lines. While we are not advocating the development of core numerical routines in Pyomo/Coopr (e.g., mixed-integer solvers), our efforts to date indicate that such an approach is effective when developing high-level computational strategies that in turn leverage fast numerical routines through, for example, the use of solver plugins.

9 Getting started with Pyomo

The installation of Pyomo is complicated by the fact that Coopr is comprised of an ensemble of Python packages, as well as third-party numerical software libraries. Coopr provides an installation script, `coopr_install`, that leverages Python's online package index [42] to install Coopr-related Python packages. For example, in Linux and MacOS 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. On all platforms, the only additional user requirement is that the PATH environment variable be updated to include coopr/bin (Linux and MacOS) or coopr\bin (MS Windows). Windows installer executables are also available.

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) 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 page, 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 is a COIN-OR software package [10]; further information can be obtained by e-mailing the associated COIN-OR or Google groups mailing lists.

10 Discussion and conclusions

Pyomo was developed and is actively supported for real-world optimization applications at Sandia National Laboratories [27]. 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 implementation of 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 optimization 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 optimization capabilities.

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

- *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 pre-written solver plugins to include other commercial solvers, e.g., XpressMP. Additionally, we plan to leverage optimizers that are available in other Python optimization packages, which are particularly interesting when solving nonlinear formulations.
- *Direct optimizer interfaces* Currently, most Coopr solvers are invoked via system calls. However, direct library interfaces to optimizers are also possible. For example, both CPLEX and GUROBI ship with Python library bindings. Direct solver interfaces promise to yield improved performance, due to the lack of a need for file manipulation and the overhead associated with spawning processes.
- *Remote solver execution* Coopr currently supports solver managers for remote solver execution using Pyro. A preliminary interface to NEOS [16] has been developed, but this solver manager currently only supports `CBC`. We plan to extend this interface, and to develop interfaces for Optimization Services [20], as well as cloud computing solvers based on, for example, Amazon's EC2 compute cloud.

Acknowledgments We thank Jon Berry, Robert Carr, Carl Laird, Cindy Phillips, and John Siirola for their critical feedback on the design of Pyomo, and David Gay for developing the Coopr interfaces to AMPL NL and SOL files. We would like to thank Gabe Hackebeil for providing the hybrid optimization scripting example. Additionally, we thank the anonymous referees and associate editor for their suggestions, which served to significantly improve the paper. 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 (2009). http://software.sandia.gov/acro
2. AIMMS: AIMMS home page (2008). http://www.aimms.com
3. AMPL: AMPL home page (2008). http://www.ampl.com/
4. Anbalagan, P., Vouk, M.: 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 (2005). http://aplepy.sourceforge.net/
6. Bertsimas, D., Tsitsiklis, J.N.: Introduction to Linear Optimization. Athena Scientific/Dynamic Ideas, Belmont (1997)
7. Birge, J.R., Louveaux, F.: Introduction to Stochastic Programming. Springer, Berlin (1997)
8. Bonmin: The Bonmin wiki page (2011). https://projects.coin-or.org/Bonmin
9. BSD: Open Source Initiative (OSI)—the BSD license (2009). http://www.opensource.org/licenses/bsd-license.php
10. COINOR: COIN-OR home page (2009). http://www.coin-or.org
11. Forrester Consulting: Open source software's expanding role in the enterprise (2007). http://www.unisys.com/eprise/main/admin/corporate/doc/Forrester_research-open_source_buying_behaviors.pdf
12. COOPR: Coopr: A common optimization python repository (2009). http://software.sandia.gov/coopr
13. CUTEr: Cuter: A constrained and unconstrained testing environment, revisited (2011). http://www.hsl.rl.ac.uk/cuter-www/index.html
14. CVXOPT: CVXOPT home page (2008). http://abel.ee.ucla.edu/cvxopt
15. Dimitrov: Nedialki Dimitrov, naval postgraduate school. Personal Communication (2011)

16. Dolan, E.D., Fourer, R., Goux, J.-P., Munson, T.S., Sarich, J.: Kestrel: an interface from optimization modeling systems to the NEOS server. INFORMS J Comput **20**(4), 525–538 (2008)
17. FLOPC++: FLOPC++ home page (2008). https://projects.coin-or.org/FlopC++
18. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: a mathematical programming language. Manag. Sci. **36**, 519–554 (1990)
19. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: a Modeling Language for Mathematical Programming, 2nd edn. Brooks/Cole Thomson Learning, Pacific Grove (2003)
20. Fourer, R., Ma, J., Martin, K.: Optimization services: a framework for distributed optimization. Oper. Res. **58**(6), 1624–1636 (2010)
21. GAMS: GAMS home page (2008). http://www.gams.com
22. Geoffrion, A.M.: An introduction to structured modeling. Manag. Sci. **33**(5), 547–588 (1987)
23. GLPK: GLPK: GNU linear programming toolkit (2009). http://www.gnu.org/software/glpk/
24. GPL: GNU general public license (2009). http://www.gnu.org/licenses/gpl.html
25. Hackebeil, G., Laird, C.: Global optimization for estimation of on/off seasonality in infectious disease spread using pyomo (2010). https://software.sandia.gov/trac/coopr/attachment/wiki/Pyomo/global_opt.pptx
26. Hart, W.E.: Python Optimization Modeling Objects (Pyomo). In: Chinneck, J.W., Kristjansson, B., Saltzman, M.J. (eds.) Operations Research and Cyber-Infrastructure (2009). doi:10.1007/978-0-387-88843-9_1
27. Hart, W.E., Phillips, C.A., Berry, J., Boman, E.G. et al.: US Environmental Protection Agency uses operations research to reduce contamination risks in drinking water. INFORMS Interfaces **39**, 57–68 (2009)
28. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint Programming and Combinatorial Optimisation in Numberjack. In: Lodi, A., Milano, M., Toth, P. (eds.) Proceedings of CPAIOR 2010, LNCS, vol. 6140. Springer, Berlin (2010)
29. Ipopt: The Ipopt wiki page (2011). https://projects.coin-or.org/Ipopt
30. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: open source scientific tools for Python (2009). http://www.scipy.org/
31. Kallrath, J.: Modeling Languages in Mathematical Optimization. Kluwer, Dordrecht (2004)
32. Karabuk, S., Grant, F.H.: A common medium for programming operations-research models. In: Proceedings of the IEEE Software, pp. 39–47 (2007)
33. Marsten, R.E.: The design of the XMP linear programming library. ACM Trans. Math. Softw. **7**(4), 481–497 (1981)
34. Oliphant, T.E.: Python for scientific computing. Computing in Science and Engineering, pp. 10–20 (2007)
35. OpenOpt: OpenOpt home page (2008). http://scipy.org/scipy/scikits/wiki/OpenOpt
36. OptimJ: Ateji home page (2008). http://www.ateji.com
37. Ortools: Google OR tools—operations research tools developed at Google (2011). http://code.google.com/p/or-tools
38. Prechelt, L.: An empirical comparison of seven programming languages. Computer **33**(10), 23–29 (2000). doi:10.1109/2.876288. ISSN: 0018-9162
39. Psyco: Psyco (2008). http://psyco.sourceforge.net/
40. PuLP: PuLP: a Python linear programming modeler (2008). http://130.216.209.237/engsci392/pulp/FrontPage
41. PyMathProg: PyMathProg home page (2009). http://pymprog.sourceforge.net/
42. PyPI: Python package index (2009). http://pypi.python.org/pypi
43. PYRO: PYRO: Python remote objects (2009). http://pyro.sourceforge.net
44. Python: Python programming language—official website (2009). http://python.org
45. PythonVSJava: Python & Java: a side-by-side comparison (2008). http://www.ferg.org/projects/python_java_side-by-side.html
46. PyUtilib: PyUtilib optimization framework (2009). http://software.sandia.gov/pyutilib
47. Roelofs, M., Bisschop, J.: AIMMS 3.9—The User's Guide (2009). http://lulu.com
48. Sayfan, G.: Building your own plugin framework. Dr. Dobbs J. (2007)
49. Shapiro, A., Dentcheva, D., Ruszczynski, A.: Lectures on Stochastic Programming: Modeling and Theory. Society for Industrial and Applied Mathematics (2009)
50. Sage, W.S.: Open Source Mathematical Software (Version 2.10.2). The Sage Group (2008). http://www.sagemath.org

51. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press, New York (1998)
52. TOMLAB: TOMLAB optimization environment (2008). http://www.tomopt.com/tomlab
53. Tratt, L.: Dynamically typed languages. Adv. Comput. **77**, 149–184 (2009)
54. Watson, J.-P., Woodruff, D.L., Hart, W.E.: Pysp: modeling and solving stochastic programs in python (2010). https://software.sandia.gov/trac/coopr/attachment/wiki/PySP/pysp_jnl.pdf
55. YAML: The official YAML web site (2009). http://yaml.org/
56. Zhou, Y., Davis, J.: Open source software reliability model: an empirical approach. ACM SIGSOFT Softw. Eng. Notes **30**, 1–6 (2005)