Coopr User Manual:
Pyomo Modeling Language and Extension Packages

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

November 7, 2009

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

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

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

Contents

Chapter 1

Introduction

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

- What is the scope of this document? Should it include Coopr Opt? How about PySP?
- Should this include all aspects of Coopr?
- What specific chapters should we include ... even if we stay focused on Pyomo?

1.1 Introduction

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

The design of Pyomo is motivated by a variety of factors that have impacted applications at Sandia National Laboratories. Sandia's discrete mathematics group has successfully used AMPL [3, 10] to model and solve large-scale integer programs for many years. This application experience has highlighted the value of Algebraic Modeling Languages (AMLs) for solving real-world applications, and AMLs are now an integral part of operations research solutions at Sandia.

Pyomo was developed to provide an alternative platform for developing math programming models that leverages Python's rich programming environment to facilitate the application and deployment of optimization capabilities. Pyomo is not intended to perform modeling *better* than existing tools. Instead, it supports a different modeling approach for which the software is designed for flexibility, extensibility, portability, and maintainability.

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

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

TODO: introduce Coopr here

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

1.2 Design Goals and Requirements

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

1.2.1 Open Source

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

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

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

Although the use of an open-source model is not a panacea; ensuring high reliability of the software requires careful software management and a commited developer community. However, there is increasing recognition that open source software provides many advantages beyond simple cost savings [6], including supporting open standards and avoiding being locked in to a single vendor.

1.2.2 Customizable Capability

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

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

1.2.3 Solver Integration

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

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

1.2.4 Abstract Models

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

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

1.2.5 Flexible Modeling Language

Another goal of Pyomo is to directly use a modern programming language to support the definition of math programming models. In this manner, Pyomo is similar to tools like FlopC++ [9] and OptimJ [26], which support modeling in C++ and Java respectively. The use of an existing programming language has several advantages:

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

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

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

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

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

1.2.6 Portability

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

1.3 Why Python?

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

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

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

- **.Net:** As mentioned earlier, the .Net languages are not portable to Linux platforms, and thus they were not suitable for Pyomo.
- **Ruby:** At the moment, Python and Ruby appear to be the two most widely recommended scripting languages that are portable to Linux platforms, and comparisons suggest that their core functionality is similar. Our preference for Python is largely based on the fact that it has a nice syntax that does not require users to type weird symbols (e.g. $, %, @). Thus, we expect this will be a more natural language for expressing math programming models.
- **Java:** Java has a lot of the same strengths as Python, and it is arguably as good a choice for Pyomo. However, Python has a powerful interactive interpreter that allows realtime code development and encourages experimentation with Python software. Thus, users can work interactively with Pyomo models to become familiar with these objects and to diagnose bugs.
- **C++:** Models formulated with the FlopC++ [9] package are similar to models developed with Pyomo. They are be specified in a declarative style using classes to represent model components (e.g. sets, variables and constraints). However, C++ requires explicit compilation to execute code, and it does not support an interactive interpreter. Thus, we believe that Python will provide a more flexible language for users.

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

Finally, we note that run-time performance was not a key factor in our decision to use Python. Recent emperical comparisons suggest that scripting languages offer reasonable alternatives to languages like C and C++, even for tasks that must handle fair amounts of computation and data [27]. 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 [36, 29]. It is widely acknowledged that Python's dynamic typing and compact, concise syntax makes software development quick and easy. Thus, it is not surprising that Python is widely used in the scientific community [23]. Large Python projects like SciPy [16] and SAGE [33] strongly leverage a diverse set of Python packages to perform complex numerical calculations.

1.4 Background

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

Algebraic Modeling Languages (AMLs) are alternative approach that allows applications to be interfaced with optimizers that can exploit problem structure. AMLs are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [18]. AMLs like AIMMS [2], AMPL [3, 10] and GAMS [12] have programming languages with an intuitive mathematical syntax that supports concepts like sparse sets, indices, and algebraic expressions. AMLs provide a mechanism for defining variables and generating constraints with a concise mathematical representation, which is essential for large-scale, real-world problems that involve thousands of constraints and variables.

Standard programming languages can also be used to formulate optimization models when used in conjunction with a software library that uses object-oriented design to support mathematical concepts. Although these modeling libraries sacrifice some of the intuitive mathematical syntax of an AML, they allow the user to leverage the greater flexibility of standard programming languages. For example, modeling tools like FlopC++ [9], OPL [25] and OptimJ [26] can be used to formulate and solve optimization models.

A related strategy is to use a high-level programming language to formulate optimization

models that are solved with optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations. This approach is increasingly common in scientific computing tools, and the Matlab TOMLAB Optimization Environment [35] is probably the most mature optimization software using this approach. However, Python has been used to implement a variety of optimization packages that use this approach:

- **APLEpy:** A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [4, 19].
- **CVXOPT:** A package for convex optimization [7].
- **PuLP:** A package that can be used to describe linear programming and mixed-integer linear programming optimization problems [28].
- **POAMS:** A modeling tool for linear and mixed-integer linear programs that defines Python objects for abstract sets, constraints, objectives, decision variables, and solver interfaces.
- **OpenOpt:** A numerical optimization framework that is closely coupled with the SciPy scientific Python package [24].
- **NLPy:** An optimization framework that leverages AMPL to create problem instances, which can then be processed in Python [22].

Pyomo is similar to APLEpy, PuLP and POAMS. All of these packages define Python objects that can be used to express models. POAMS and Pyomo support a clear distinction between abstract models and problem instances. This design has several advantages, which were summarized by Fourer and Gay [10] when presenting AMPL:

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

The main high-level feature that distinguishes Pyomo from POAMS is Pyomo's support for an instance construction process that is automated by object properties. This is akin to the capabilities of AML's like AMPL and GAMS, and it provides a standardized technique for constructing model instances. Hart [15] provides Python examples that illustrate the differences between PuLP, POAMS and Pyomo.

Chapter 2

Getting Started with Pyomo

2.1 Introduction

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

The design of Pyomo is motivated by a variety of factors that have impacted applications at Sandia National Laboratories. Sandia's discrete mathematics group has successfully used AMPL [3, 10] to model and solve large-scale integer programs for many years. This application experience has highlighted the value of Algebraic Modeling Languages (AMLs) for solving real-world applications, and AMLs are now an integral part of operations research solutions at Sandia.

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

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

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

2.2 Installing Coopr

There are several different ways that Coopr can be installed:

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

source Coopr can be installed from source.

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

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

```
easy_install Coopr
```

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

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

```
coopr_install coopr
```

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

```
python coopr_install coopr
```

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

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

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

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

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

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

```
coopr_install --trunk coopr
```

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

```
coopr_install --trunk coopr
```

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

```
coopr_install --clear coopr
```

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

```
coopr_install --update coopr
```

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

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

going directly through the Coopr software repository. For example, the `coopr.plugins.neos` package provides a simple example of how Coopr can be extended with plugins to enable optimization on the NEOS optimization server [8]. This plugin package can be installed with Coopr using the `--forum-pkg` option:

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

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

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

Finally, here are some notes about `coopr_install`:

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

Chapter 3

The Coopr Optimization Package

3.1 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. Coopr utilizes a component architecture to provide plugins that modularize many aspects of the optimization process. This includes components that manage the execution of optimizers, which enables transparent parallelization of independent optimization tasks. This component architecture also supports a generic optimization process. Finally, Coopr provides a simple installation mechanism that leverage's Python's online package index.

3.1.1 Optimization Plugins

Coopr uses plugin components to modularize the steps needed to perform optimization. A component is a software package, module, or object that provides a particular functionality. Plugin components augment the execution flow by implementing functionality that is exercised "on demand." Component-based software with plugins is a best practice for extend and evolve complex software systems in a reliable manner [32]. Component-based software frameworks manage the interaction between components to promote adaptability, scalability and maintainability in large software systems [34]. For example, with component-based software there is much less need for major release changes because software changes can be encapsulated within individual components. Component architectures also encourage third-party developers to add value to software systems without risk of destabilizing the core functionality.

Coopr uses the PyUtilib plugin framework [30] to define interfaces for the following plugin components:

- solvers, which perform optimization
- solver managers, which manage the execution of solver plugins
- converters, which translate between different optimization problem file formats

- solution readers, which load optimization solutions from files
- problem writers, which create files that contain optimization problems

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

Coopr includes a variety of plugins that implement these component interfaces, many of which rely on third-party software packages to implement key functionality. For example, solver plugins are available for the CPLEX, CBC, PICO and GLPK mixed-integer linear programming solvers. These plugins naturally rely on the availability of binary executables for these solvers, which need to be installed separately. Similarly, Coopr includes plugins that convert between different optimization problem file formats, which rely on binary executables built by the GLPK [14] and Acro [1] software libraries.

Taken together, plugins provide the following capabilities that simplify optimization within Pyomo:

- **Dynamic Registration of Optimizers**: Optimizers are registered as plugins, which provides an extensible architecture for developers of third-party optimizers. Coopr uses a dynamic registration process that disables plugins whose executables are not available at runtime. This minimizes the effort needed to integrate new optimizers into Pyomo.
- **Problem Transformation**: A key challenge for optimization packages is the need to support a diverse set of problem formats. This is an issue even for LP and MILP solver packages, where MPS is the least common denominator for users. Coopr includes plugins that can write problems in NL and CPXLP formats. Additionally, Coopr includes an automatic problem transformation mechanism that enables the application of optimizers to problems with a wide range of formats. This mechanism employs plugins, which simplifies the process for adding new conversion capabilities.
- **Solver Managers**: Some optimization techniques involve the execution of multiple, independent optimization subproblems. In this context, Pyomo's solver manager plugin components provide a high-level abstraction of the execution of these subproblems. This abstraction enables the parallel execution of optimizers for these subproblems in a transparent manner.

3.1.2 Generic Optimization Process

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

```
opt = SolverFactory( name )
opt.reset()
results = opt.solve( problem )
```

```
results.write()
```

Note that this relies on Coopr's explicit segregation of problems and solvers into separate objects. This promotes the development of tools like Pyomo that support flexible definition of optimization applications, and it enables automatic transformation of problem instances.

Coopr borrows and extends the representation used by the COIN-OR OS project [11] to support a general representation of optimizer results. The *results* object returned by a Coopr optimizer includes information about the problem, the solver execution, and one or more solutions generated during optimization.

For example, if the problem in Appendix **??** is being solved, the simple Coopr optimization script would print the following information that is contained in the `results` object:

```
==========================================================
- Solver Results                                         -
==========================================================

----------------------------------------------------------
------ Problem Information                          ------
----------------------------------------------------------
name: None
num_constraints: 5
num_nonzeros: 6
num_objectives: 1
num_variables: 2
sense: maximize
upper_bound: 192000
----------------------------------------------------------
------ Solver Information                           ------
----------------------------------------------------------
error_rc: 0
nbounded: None
ncreated: None
status: ok
systime: None
usrtime: None
----------------------------------------------------------
------ Solution 0
----------------------------------------------------------
gap: 0.0
status: optimal
value: 192000
Primal Variables
      X_bands_        6000
      X_coils_        1400
```

```
Dual Variables
      c_u_Limit_1      4
      c_u_Time_0       4200
```

Chapter 4

PySP

4.1 Overview

The pysp package extends the pyomo modeling language to support multi-stage stochastic programs with enumerated scenarios. Pyomo and pysp are Python version 2.6 programs. The underlying algorithm in pysp is based on Progressive Hedging (PH) [31], which uses *weights* corresponding to each variable to force convergence. In order to specify a program, the user must provide a reference model and a scenario tree.

The software is executed with a command of the form

```
python phdriver.py
```

but "python" might be replaced by a command to execute Python version 2.6 and "phdriver.py" might include a path specification (e.g., `..\phdriver.py` or `../phdriver.py` when there are various data subdirectories for one project directory and phdriver.py is in the project directory). It is possible, and generally necessary, to invoke extensions to the basic algorithm and to override default parameters; this is described in Sections 4.3 and 4.3.2.

4.1.1 Reference Model

The reference model describes the problem for a canonical scenario. It does not make use of, or describe, a scenario index or any information about uncertainty. Typically, it is just the model that would be used if there were only a single scenario. It is given as a pyomo file. Data from an arbitrary scenario is needed to instantiate.

The objective function needs to be separated by stages. The term for each stage should be "assigned" (i.e., constrained to be equal to) a variable. These variables names are reported in ScenarioStructure.dat so that they can be used for reporting purposes.

4.1.2 Scenario Tree

The scenario tree provides information about the time stages and the nature of the uncertainties. In order to specify a tree, we must indicate the time stages at which information

becomes available. We also specify the nodes of a tree to indicate which variables are associated with which realization at each stage. The data for each scenario is provided in separate data files, one for each scenario.

4.2 File Structure

- ReferenceModel.py (A pyomo model file)
- ReferenceModel.dat (data for an arbitrary scenario)
- ScenarioStructure.py (do not edit this file)
- ScenarioStructure.dat (among other things: the scenario names: Sname)
- *Sname.dat (full data for now) one file for each scenario

In this list we use "Sname" as the generic scenario name. The file scenariostructure.dat gives the names of all the scenarios and for each scenario there is a data file with the same name and the suffix ".dat" that contains the full specification of data for the scenario.

4.2.1 ScenarioStructure.dat

The file ScenarioStructure.py defines the python sets and parameters needed to describe stochastic elements. This file should not be edited. Data to instantiate these sets and parameters is provided by users in the file ScenarioStructure.dat, which can be given in AMPL [3] format. This file contains the following data:

- set Scenarios: List of the names of the scenarios. These names will subsequently be used as indexes in this data file and these names will also be used as the root file names for the scenario data files (each of these will have a .dat extension).

  item set Stages: List of the names of the time stages, which must be given in time order. In the sequel we will use StageName to represent a node name used as an index.

- set Nodes: List of the names of the nodes in the scenario tree. In the sequel we will use NodeName to represent a node name used as an index.

- param NodeStage: A list of pairs of nodes and stages to indicate the stage for each node.

- param Parent: A list of node pairs to indicate the parent of each node that has a parent (the root node will not be listed).

- set Children[NodeName]: For each node that has children, provide the list of children. No sets will be give for leaf nodes.

- param ConditionalProbability: For each node in the scenario tree, give the conditional probability. For the root node it must be given as 1 and for the children of any node with children, the conditional probabilities must sum to 1.

- param ScenarioLeafNode: A list of scenario and node pairs to indicate the leaf node for each scenario.

- set StageVariables[StageName]: For each stage, list the pyomo model variables associated with that stage.

The default behavior is one file per scenario and each file has the full data for the scenario. An alternative is to specify just the data that changes from the root node in one file per tree node. To select this option, add the following line to ScenarioStructure.dat:

```
param ScenarioBasedData := False ;
```

This will set it up to want a per-node file, something along the lines of what's in `examples/pysp/farmer/NODEDATA`.

4.3 Extensions via Callbacks

Basic PH can converge slowly, so it is usually advisable to extend it or modify it. In PYSP, this is done via the pyomo plug-in mechanism. The basic PH implementation provides callbacks that enable access to the data structures used by the algorithm. In §4.3.1 we describe extensions that are provided with the release. In §4.3.3, we provide information to power users who may wish to modify or replace the extensions.

4.3.1 Watson and Woodruff Extensions

Watson and Woodruff describe innovations for accelerating PH [37], some of which are implemented in the file `wwextension.py`. Many of the examples described in §4.4 use this plug-in. The main concept is that some integer variables should be fixed as the algorithm progresses for two reasons:

- Convergence detection: A detailed analysis of PH algorithm behavior on the problems indicates that individual decision variables frequently converge to specific, fixed values scenarios in early PH iterations. Further, despite interactions among the the variables, the value frequently does not change in subsequent PH iterations. Such variable "fixing" behaviors lead to a potentially powerful, albeit obvious, heuristic: once a particular variable has been the same in all scenarios for some number of iterations, fix it to that value. For problems where the constraints effectively limit x from both sides, these methods may result in PH encountering infeasible scenario sub-problems even though the problem is ultimately feasible.

- Cycle detection: When there are integer variables, cycling is sometimes encountered, consequently, cycle detection and avoidance mechanisms are required to force eventual convergence of the PH algorithm in the mixed-integer case. To detect cycles, we focus on repeated occurrences of the weights, implemented using a simple hashing scheme [38] to minimize impact on run-time. Once a cycle in the weight vectors associated with any decision variable is detected, the value of that variable is fixed.

Variable Specific Parameters

The plug-in makes use of parameters to control behaviour at the variable level. Global defaults (to override the defaults stated here) should be set using methods described in §4.3.2. Values for each variable should be set using methods described in §4.3.2. Note that for variable fixing based on convergence detection, iteration zero is treated separately. The parameters are as follows:

- fix_continuous_variables: True or False. If true, fixing applies to all variables. If false, then fixing applies only to discrete variables.

- Iter0FixIfConvergedAtLB: 1 (True) or 0 (False). If 1, then discrete variables that are at their lower bound in all scenarios after the iteration zero solves will be fixed at that bound.

- Iter0FixIfConvergedAtUB: 1 (True) or 0 (False). If 1, then discrete variables that are at their upper bound in all scenarios after the iteration zero solves will be fixed at that bound.

- Iter0FixIfConvergedAtNB: = 1 1 (True) or 0 (False). If 1, then discrete variables that are at the same value in all scenarios after the iteration zero solves will be fixed at that value, without regard to whether it is a bound. If this is true, it takes precedence. A value of zero, on the other hand, implies that variables will not be fixed at at a non-bound.

- FixWhenItersConvergedAtLB: The number of consecutive PH iterations that discrete variables must be their lower bound in all scenarios before they will be fixed at that bound. A value of zero implies that variables will not be fixed at the bound.

- FixWhenItersConvergedAtUB: The number of consecutive PH iterations that discrete variables must be their upper bound in all scenarios before they will be fixed at that bound. A value of zero implies that variables will not be fixed at the bound.

- FixWhenItersConvergedAtNB: The number of consecutive PH iterations that discrete variables must be at the same, consistent value in all scenarios before they will be fixed at that value, without regard to whether it is a bound. If this is true, it takes precedence. A value of zero, on the other hand, implies that variables will not be fixed at at a non-bound.

- FixWhenItersConvergedContinuous: The number of consecutive PH iterations that continuous variables must be at the same, consistent value in all scenarios before they will be fixed at that value. A value of zero implies that continuous variables will not be fixed.

- CanSlamToLB: True or False. If True, then slamming can be to the lower bound for any variable.

- CanSlamToMin: True or False. If True, then slamming can be to the minimum across scenarios for any variable.

- CanSlamToAnywhere: True or False. If True, then slamming can be to any value.

- CanSlamToMax: True or False. If True, then slamming can be to the maximum across scenarios for any variable.

- CanSlamToUB: True of False. If True, then slamming can be to the upper bound for any variable.

4.3.2 General Parameters

The plug-in also makes use of the following parameters, which should be set using methods described in §4.3.2.

- SlamAfterIter: Iteration number after which one variable every other iteration will be slammed to force convergence. Default: the number of scenarios.

- hash_hit_len_to_slam: Ignore possible cycles for which the only evidence of a cycle is less than this. Default: the number of scenarios.

Setting Parameter Values

The parameters of ph and of any callbacks can be set using the file wwph.cfg, which is executed by the python interpreter.

Setting Suffix Values

Suffixes are set using the data file named `wwph.suffixes` using this syntax:

```
VARSPEC SUFFIX VALUE ...
```

where VARSPEC is replaced by a variable specification, SUFFIX is replaced by a suffix name and VALUE is replaced by the value of the suffix for that variable or those variables. Here is an example:

```
Delta CanSlamToLB False
Gamma[*,Ano1] SlammingPriority 10
Gamma[*,Ano2] SlammingPriority 20
...
```

4.3.3 Callback Details

A callback class definition named iphextension is in the file iphextension.py and can be used to implement callbacks at a variety of points in PH. For example, the method post_iteration_0_solves is called immediately after all iteration zero solves, but before averages and weights have been computed while the method post_iteration_0 is called after averages and weights based on iteration zero have been computed. The file iphextension is in the coopr/pysp directory and is not intended to be edited by users.

The user defines a class derived from SingletonPlugin that implements iphextension. Its name is given to phdriver as an option (e.g., on the command line). This class will be automatically instantiated by phdriver. It has access to data and methods in the PH class, which are defined in the file ph.py. An example of such a class is in the file named testphextension.py in the pysp example directory.

If you copy testphextension.py to your working directory or create it there, you can have it used by adding a line to phdriver.py:

```
from testphextension import *
```

Here are the callbacks:

- post_iteration_0_solves: Called after iteration zero solutions and some statistics about solutions have been computed, but before averages weights are updated.
- post_iteration_0: Called after all processing for iteration zero is complete.
- post_iteration_k_solves: Called after solutions some statistics about solutions have been computed, but before averages weights are updated for iterations after iteration zero.
- post_iteration_k: Called after all processing for each iteration after iteration 0 is complete.
- post_ph_execution:

4.4 Examples

A number of examples are provided with pysp.

4.4.1 Farmer Example

This two-stage example is composed of models and data for the "Farmer" stochastic program, introduced in Section 1.1 of "Introduction to Stochastic Programming" by Birge and Louveaux [5].

- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)
- ScenarioStructure.py: defines the scenario tree structure for the SP. SHOULD NOT BE MODIFIED.
- ScenarioStructure.dat: data file defining the scenario tree.
- AboveAverageScenario.dat: one of the scenario data files.
- BelowAverageScenario.dat: one of the scenario data files.
- AverageScenario.dat: one of the scenario data files.

The file phdriver.py executes PH, assuming the ReferenceModel.* and ScenarioStructure.* files are present and correct. This example is probably in a directory with a name something like:

```
pyomodist\packages\coopr\examples\pysp\farmer
```

The data is in a subdirectory called SCENARIODATA.

To invoke PH for this problem, connect to the directory containing the data files and use the command:

```
python ..\phdriver.py
```

or a similar command so that Python version 2.6 is passed the name of the file `phdriver.py`.

4.4.2 Forestry Example

This four-stage example is composed of models and data for the "forestry" stochastic program [], which consists of the following files:

- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)
- ScenarioStructure.py: defines the scenario tree structure for the SP. SHOULD NOT BE MODIFIED.
- ScenarioStructure.dat: data file defining the scenario tree.
- Scenario1.dat: one of the scenario data files.
- Scenario2.dat: one of the scenario data files.
- ...

The file phdriver.py executes PH, assuming the ReferenceModel.* and ScenarioStructure.* files are present and correct. This example is probably in a directory with a name something like:

```
pyomodist\packages\coopr\examples\pysp\sizes
```

There are two families of instances: "Chile" and "Davis," each with four stages and eighteen scenarios. This is also a small two-stage, four scenario instances in the subdirectory DAVIS2STAGE.

To invoke PH for this problem, connect to the directory containing the data files and use the command:

```
python ..\phdriver.py
```

or a similar command so that Python version 2.6 is passed the name of the file `phdriver.py`.

4.4.3 Sizes Example

This two-stage example is composed of models and data for the "Sizes" stochastic program [17, 20], which consists of the following files:

- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)

- ScenarioStructure.py: defines the scenario tree structure for the SP. SHOULD NOT BE MODIFIED.
- ScenarioStructure.dat: data file defining the scenario tree.
- Scenario1.dat: one of the scenario data files.
- Scenario2.dat: one of the scenario data files.
- ...

The file phdriver.py executes PH, assuming the ReferenceModel.* and ScenarioStructure.* files are present and correct. This example is probably in a directory with a name something like:

```
pyomodist\packages\coopr\examples\pysp\sizes
```

The data for a three scenario version is in a subdirectory called SIZES3 and a ten scenario dataset is in SIZES10.

To invoke PH for this problem, connect to the directory containing the data files and use the command:

```
python ..\phdriver.py
```

or a similar command so that Python version 2.6 is passed the name of the file `phdriver.py`.

Chapter 5

Advanced Pyomo Examples

5.1 XXX Advanced Pyomo Examples

5.1.1 Parallel Benders Decomposition

TODO: Does the following paragraph go here? I think that Dave's talking about the abilities that we're leveraging in PH, but we haven't discussed that in this paper. Perhaps this should go back into the introduction, but if so then the intent needs to be clarified for me (BILL).

An important consequence of the design using Python and integration with the Coopr environment is that modularity is fully supported over a range of abstraction. At one extreme, the model elements can be manipulated explicitly by specifying their names and the values of their indexes. This sort of reference can be made more abstract, as is the case with algebraic modeling languages, by specifying various types of named sets so that the dimensions and details of the data can be separated from the specification of the model. Separation of an abstract declarative model from the data specification is a hallmark of structure modeling techniques for efficient modeling [13]. At the other extreme, elements of a mathematical program can be treated in their fully canonical form as is supported by callable solver libraries. Methods can be written that operate, for example, on objective functions or constraints in a fully general way. This capability is a fundamental tool for general algorithm development and extension [21]. Pyomo provides the full continuum of abstraction between these two extremes to support modeling and development. Furthermore, methods are extensible via overloading of all defined operations. Both modelers and developers can alter the behavior of the package or add new functionality.

Chapter 6

Discussion

6.1 XXX Discussion

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

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

- Nonlinear Problems - Conceptually, it is straightforward to extend Pyomo to support the definition of general nonlinear models. However, the model generation and expression mechanisms need to be re-designed to support capabilities like automatic differentiation.

- Optimized Expression Trees - Our scaling experiments suggest that Pyomo's runtime performance can be 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.

- Python Optimizers - A variety of Python optimization packages are now available, which we would like to leverage. In particular, these will be important when nonlinear problems are supported in Pyomo.

- Direct Optimizer Interfaces - Coopr currently does not support direct library interfaces to optimizers, although this is a capability that is strongly supported by Python. This is not a design limitation of Coopr, but instead has been a matter of development priorities. Similarlly, extensions to external solver packages like Acro's COLIN optimization library [1] will be quite natural; Coopr has preliminary support for applications that are defined using a system call interface.

Finally, it should be straightforward to extend Coopr to support remote solver execution with NEOS [8] and Optimization Services [11].

Acknowledgements

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

Bibliography

[1] *ACRO optimization framework*. http://software.sandia.gov/acro, 2009.

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

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

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

[5] J. Birge and F. Louveaux, *Introduction to Stochastic Programming*, Springer, 1997.

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

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

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

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

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

[11] R. Fourer, J. Ma, and K. Martin, *Optimization services: A framework for distributed optimization*, Mathematical Programming, (2008). (submitted).

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

[13] A. M. Geoffrion, *An introduction to structured modeling*, Management Science, 33 (1987), pp. 547–588.

[14] *Glpk: Gnu linear programming toolkit*. http://www.gnu.org/software/glpk/, 2009.

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

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

[17] S. Jorjani, C. Scott, and D. Woodruff, *Selection of an optimal subset of sizes*, International journal of production research, 37 (1999), pp. 3697–3710.

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

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

[20] A. Løkketangen and D. Woodruff, *Progressive hedging and tabu search applied to mixed-integer (0,1) multistage stochastic programming*, Journal of Heuristics, 2 (1996), pp. 111–128.

[21] R. E. Marsten, *The design of the xmp programming library*, ACM Transactions on Mathematical Software, 7 (1981), pp. 481–497.

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

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

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

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

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

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

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

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

[30] *PyUtilib optimization framework*. `http://software.sandia.gov/pyutilib`, 2009.

[31] R. Rockafellar and R. J.-B. Wets, *Scenarios and policy aggregation in optimization under uncertainty*, Mathematics of Operations Research, (1991), pp. 119–147.

[32] G. Sayfan, *Building your own plugin framework*, Dr. Dobbs Journal, (2007).

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

[34] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, 1998.

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

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

[37] J.-P. Watson and D. Woodruff, *Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems*, tech. rep., Sandia National Laboratories, 2007.

[38] D. Woodruff and E. Zemel, *Hashing vectors for tabu search*, Annals of Operations Research, 41 (1993), pp. 123–137.