Springer Optimization and Its Applications 67

William E. Hart
Carl Laird
Jean-Paul Watson
David L. Woodruff

Pyomo – Optimization Modeling in Python

Springer

Springer Optimization and Its Applications

VOLUME 67

Aims and Scope

Optimization has been expanding in all directions at an astonishing rate during the last few decades. New algorithmic and theoretical techniques have been developed, the diffusion into other disciplines has proceeded at a rapid pace, and our knowledge of all aspects of the field has grown even more profound. At the same time, one of the most striking trends in optimization is the constantly increasing emphasis on the interdisciplinary nature of the field. Optimization has been a basic tool in all areas of applied mathematics, engineering, medicine, economics, and other sciences.

The series *Springer Optimization and Its Applications* publishes undergraduate and graduate textbooks, monographs and state-of-the-art expository work that focus on algorithms for solving optimization problems and also study applications involving such problems. Some of the topics covered include nonlinear optimization (convex and nonconvex), network flow problems, stochastic optimization, optimal control, discrete optimization, multiobjective programming, description of software packages, approximation techniques and heuristic approaches.

For further volumes:
http://www.springer.com/series/7393

William E. Hart • Carl Laird • Jean-Paul Watson
David L. Woodruff

Pyomo—Optimization Modeling in Python

Springer

William E. Hart
Data Analysis and Informatics Department
Sandia National Laboratories
Albuquerque, NM 87185
USA
wehart@sandia.gov

Carl Laird
Department of Chemical Engineering
Texas A&M
College Station, TX 77843
USA
carl.laird@tamu.edu

Jean-Paul Watson
Discrete Mathematics and Complex Systems Department
Sandia National Laboratories
Albuquerque, NM 87185
USA
jwatson@sandia.gov

David L. Woodruff
Graduate School of Management
University of California, Davis
Davis, CA 95616
USA
dlwoodruff@ucdavis.edu

ISSN 1931-6828
ISBN 978-1-4614-3225-8 e-ISBN 978-1-4614-3226-5
DOI 10.1007/978-1-4614-3226-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2012930924

For Valerie, Christy, Michelle and Barbara.
Thank you for your support and patience
during the many nights and weekends that we
have spent on Pyomo and this book.

Preface

This book describes a new tool for mathematical modeling: the Python Optimization Modeling Objects (Pyomo) software. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with algebraic modeling languages (AMLs), which support the description and analysis of mathematical models with a high-level language. Although most AMLs are implemented in custom modeling languages, Pyomo's modeling objects are embedded within Python, a full-featured high-level programming language that contains a rich set of supporting libraries.

Modeling is a fundamental process in many aspects of scientific research, engineering and business, and the widespread availability of computing resources has made the numerical analysis of mathematical models a commonplace activity. Furthermore, AMLs have emerged as a key capability for robustly formulating large models for complex, real-world applications [40]. AMLs simplify the process of formulating complex models by simplifying the management of sparse data and supporting the natural expression of model components. Additionally, AMLs like Pyomo support scripting with model objects which facilitates rapid development of new analysis tools.

Goals of the Book

In this book, we provide an introduction to the Pyomo modeling software. A key goal of this book is to provide a comprehensive reference that will enable the user to develop optimization models with Pyomo. The book contains many example models, and the presentation of Pyomo's capabilities highlights different techniques that can be used to formulate models. The presentation in the book is roughly broken down into three parts:

1. Introduction - Introducing mathematical modeling and Pyomo, an overview of Pyomo's design, and an illustration of Pyomo with increasingly complex models.
2. Modeling Components - Detailed descriptions of the core modeling components that are supported by Pyomo.
3. Advanced Capabilities - Presentations of advanced features, including modeling of nonlinear and stochastic programs, as well as high-level scripting with Python.

Another goal of this book is to illustrate the breadth of the modeling and analysis capabilities that are supported by Pyomo. Pyomo supports the formulation and analysis of common optimization models, including linear programs, mixed-integer linear programs, nonlinear programs, mixed-integer nonlinear programs and stochastic programs. Additionally, Pyomo includes solver interfaces for a variety of widely used optimization software packages, including CBC, CPLEX, GLPK, GUROBI, and PICO. Additionally, Pyomo can execute optimizers that employ the AMPL Solver Library interface.

Finally, this book provides the information needed to install and get started with Pyomo. Pyomo is a component of the Coopr software project. This book documents the capabilities of the Coopr 3.1 release, which includes version 3.0 of `coopr.pyomo`, which defines Pyomo. Appendix A describes installation options for Coopr. Coopr leverages a variety of third-party Python packages, and installation options described in the appendix include the installation of these auxilliary packages.

Who Should Read This Book

This book is intended to be a reference for students, academic researchers and practitioners. The design of Pyomo is simple enough that it has been effectively used in the classroom with undergraduate and graduate students. However, we assume that the reader is generally familiar with optimization and mathematical modeling. Although this book does not contain a glossary, we recommend the Mathematical Programming Glossary [35] as a reference for the reader.

A goal of this book is to help users get started with Pyomo even if they have little knowledge of Python. Appendix B provides a quick introduction to Python, but we have been impressed with how well standard Python reference texts support new Pyomo users. Although Pyomo introduces Python objects and a process for applying

them, the expression of models with Pyomo strongly reflects Python's clean, concise syntax.

Note that our discussion of Pyomo's advanced modeling capabilities assumes some background in object-oriented design and features of the Python programming language. For example, our discusion of modeling components distinguishes between class definitions and class instances. Similarly, our discussion of Pyomo expressions requires a description of how operator overloading is used. We have not attempted to describe these advanced features of Python in the book. Thus, a user should expect to develop some familiarity with Python in order to effectively understand and use advanced modeling features.

Pyomo is also a valuable tool for academic researchers and practitioners. A key focus of Pyomo development has been on the ability to support the formulation and analysis of real-world applications. Pyomo supports our work with complex, real-world applications, so key issues like run-time performance and robust solver interfaces are a priority.

Additionally, we believe that researchers will find that Coopr provides an effective framework for developing high-level optimization and analysis tools. For example, Pyomo supports stochastic programming with extensions that are defined in Coopr's PySP package. PySP provides generic solvers for stochastic programming, and it leverages the fact that Pyomo's modeling objects are embedded within a full-featured high-level programming language. This allows for transparent parallelization of sub-problems using Python parallel communication libraries. This ability to support generic solvers for complex models is very powerful, and we believe that it can be used with many other optimization analysis techniques.

Comments and Questions

Further information about Pyomo and Coopr is available on the Coopr wiki:

```
https://software.sandia.gov/trac/coopr
```

Coopr is also hosted at COIN-OR:

```
https://projects.coin-or.org/Coopr
```

We strongly encourage feedback from readers, either through direct communication with the authors or with the Coopr Forum:

```
coopr-forum@googlegroups.com
```

We hope this will include feedback on typos and errors in our examples. Additionally, we welcome comments on the presentation of this material, and suggestions for material that we should develop in the other book chapters.

Good Luck!

Albuquerque, New Mexico, USA — *William E. Hart*
College Station, Texas, USA — *Carl Laird*
Albuquerque, New Mexico, USA — *Jean-Paul Watson*
Davis, California, USA — *David L. Woodruff*
December, 2011

Acknowledgements

The development of this book was supported in part by the Office of Advanced Scientific Computing Research within the DOE Office of Science as part of the Complex Interconnected Distributed Systems program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The development of this book was also supported in part by the National Science Foundation under Grant CBET#0941313 and CBET#0955205. The authors gratefully acknowledge this support.

This book would not have been possible without the efforts of many people. We thank Elizabeth Loew at Springer for helping shepherd this book from an initial concept to final production; her enthusiasm for publishing is contagious. Also, we thank Madelynne Farber at Sandia National Laboratories for her guidance with the legal process for releasing open source software and book publishing. Finally, we thank Doug Prout for developing the Pyomo, PySP and Coopr logos.

We are indebted to our reviewers for the time and effort they put into helping this book be successful. Without them, this book would contain many typos and software bugs. So, thanks to Zev Friedman, Gabriel Hackebeil, Harvey Greenberg, Sean Legg, Angelica Wong, and Daniel Word. Special thanks to Amber Gray-Fenner.

We are particularly grateful to the growing community of Pyomo users. Your interest and enthusiasm for Pyomo was the most important factor in our decision to write this book. We are particularly thankful for the early adopters of Pyomo who have provided detailed feedback on the design and utility of the software: Fernando Badilla, Steven Chen, Ned Dmitrov, YueYue Fan, Eric Haung, Allen Holder, Andres Iroume, Darryl Melander, Carol Meyers, Pierre Nancel-Penard, Mehul Rangwala and Eva Worminghaus. Your feedback continues to have a major impact on the design and capabilities of Pyomo.

We also thank our friends in the COIN-OR project for supporting the Coopr software, which includes Pyomo. Although the main development site for Coopr is

hosted at Sandia National Laboratories, our partnership with COIN-OR is a key part of our strategy to ensure that Coopr remains a viable open source software project.

A special thanks goes to our collaborators who have developed Pyomo and other packages in Coopr: Timothy Ekl, Gabriel Hackebeil, Kevin Hunter, John Siirola, Patrick Steele, and Daniel Word. We also thank Tom Brounstein, Dave Gay, and Nick Benevidas for helping develop Python modules and documentation for Pyomo.

And finally, we would like to thank our families and friends for putting up with our passion for optimization software.

Contents

Chapter 1
Introduction

Abstract This chapter introduces and motivates Pyomo, a Python-based modeling tool for optimization models. Modeling is a fundamental process in many aspects of scientific research, engineering, and business. Algebraic modeling languages like Pyomo are high-level programming languages for simplifying the specification and solution of mathematical problems. Pyomo provides a flexible, extensible modeling framework that supports the central ideas of modern algebraic modeling languages within a widely used programming language.

1.1 Mathematical Modeling

This book describes a new tool for mathematical modeling: the Python Optimization Modeling Objects (Pyomo) software. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with algebraic modeling languages (AMLs) such as AMPL [4], AIMMS [2], and GAMS [24], but Pyomo's modeling objects are embedded within Python, a full-featured, high-level programming language that contains a rich set of supporting libraries.

Modeling is a fundamental process in many aspects of scientific research, engineering, and business. Modeling involves the formulation of a simplified representation of a system or real-world object. These simplifications allow structured representation of knowledge about the original system that facilitates the analysis of the resulting model. Models are used [52]to

- **Explain phenomena** that arise in a system;
- **Make predictions** about future states of a system;
- **Assess key factors** that influence phenomena in a system;
- **Identify extreme states** in a system that might represent worst-case scenarios or minimal cost plans; and
- **Analyze trade-offs** to support human decision makers.

W.E. Hart et al., *Pyomo—Optimization Modeling in Python,*
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_1,

Additionally, the structured aspect of a model's representation facilitates communication of the knowledge associated with a model. For example, a key aspect of models is the level of detail needed for the scope of application where the model is used. The level of detail in a model reflects the system knowledge that is needed to employ the model in an application.

Mathematical models represent system knowledge with a formalized mathematical language. The advantage of mathematical models is that they can be precisely analyzed using mathematical theory and computational algorithms. This allows modelers to rigorously identify system limitations. For example, planning activities can be analyzed to assess feasibility given limited resources and to identify bounds on the minimum cost for any possible plan.

Mathematics has always played a fundamental role in representing and formulating our knowledge. Mathematical modeling has become increasingly formal as new mathematical frameworks have emerged to express complex systems. The following mathematical concepts are central to modern modeling activities:

- **variables**: These represent *unknown* or changing parts of a model (e.g., whether or not to make a decision, or the characteristic of a system outcome).
- **parameters**: These are symbolic representations for real-world data, which might vary for different scenarios.
- **relations**: These are *equations*, *inequalities*, or other mathematical relationships that define how different parts of a model are connected to each other.

Optimization models are mathematical models that include functions that represent goals or objectives for the system being modeled. Optimization models can be analyzed to explore system trade-offs in order to find solutions that optimize system objectives. Consequently, these models can be used for a wide-range of scientific, business, and engineering applications.

Note that a mathematical model does not need to be defined with real-world data. For example, the following equations represent a linear program (LP) with parameters n and b, and parameter vectors a and c:

$$\begin{array}{ll} \min & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} a_i x_i \geq b \\ & x_i \geq 0 \qquad\qquad \forall i = 1 \ldots n \end{array}$$

We call this an *abstract* or *symbolic* mathematical model since it relies on unspecified parameter values. Data values can be used to specify a *model instance*. For example, the following LP model is an instance of the previous abstract model:

$$\begin{array}{ll} \min & x_1 + 2x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{array}$$

Note that in some contexts a mathematical model can be directly defined with real-world data. We call these *concrete* mathematical models. Thus, an instance of an abstract model is a concrete model.

1.2 Modeling Languages for Optimization

The widespread availability of computing resources has made the numerical analysis of mathematical models commonplace. The computational analysis of a mathematical model requires a concrete model in computer data structures that are used by a solver software package to perform its analysis. Without a modeling language, the process of setting up input files, executing a solver, and extracting the final results from the solver output is tedious and error prone. This difficulty is compounded in complex, large-scale, real-world applications which are difficult to debug when errors occur. Additionally, optimization packages use many different formats, but few of them can be considered to be standard across many optimizers. Thus, the application of multiple optimization solvers to analyze a model introduces additional complexities.

AMLs are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [30, 40]. AMLs minimize the difficulties associated with analyzing mathematical models by enabling a high-level specification of a mathematical problem. Furthermore, AML software provides standard interfaces to external solver packages that are used to analyze problems.

AMLs like AIMMS [2], AMPL [4, 23], and GAMS [24] provide programming languages with an intuitive mathematical syntax for defining variables and generating constraints with a concise mathematical representation. Further, these AMLs support concepts like sparse sets, indices, and algebraic expressions, which are essential for large-scale, real-world problems with thousands or millions of constraints and variables. These AMLs can represent a wide variety of optimization models, and they interface with a rich set of solver packages.

AMLs are increasingly being extended to include custom scripting capabilities, which can describe solution algorithms alongside of model declarations. Similarly, standard programming languages like Java and C++ have been extended to include modeling constructs. For example, modeling tools like FlopC++ [22] and OptimJ [46] support the formulation of optimization models using an object-oriented design. Although these modeling libraries sacrifice some of the intuitive mathematical syntax of a custom AML, they allow the user to leverage the greater flexibility of standard programming languages. An advantage of these AML extensions is that they can link directly to high-performance optimization libraries, which can be an important consideration in some applications.

A related strategy is to use an AML that extends a *standard* 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 an increasingly common approach for scientific computing software, and the Matlab TOMLAB Optimization Environment [57] is probably the most mature optimization software using this approach.

Pyomo is another example of this last approach [33]. Pyomo is an AML that extends Python to include objects for mathematical modeling. These objects can be used to formulate optimization models which are then analyzed with external solvers. Python's clean syntax allows Pyomo to express mathematical concepts in a reasonably intuitive and concise manner. Further, Python's expressive programming environment can be used to formulate complex models as well as to define custom high-level solvers that customize the execution of high-performance optimization libraries. Finally, Pyomo can be used within an interactive Python shell, thereby allowing a user to interactively interrogate Pyomo-based models.

1.3 Modeling Graph Coloring

We illustrate Pyomo's modeling capabilities with a simple application: minimum graph coloring. Graph coloring is the assignment of colors to vertices of a graph such that no two adjacent vertices share the same color. (This is also known as vertex coloring.) Graph coloring has many practical applications, including register allocation in compilers, scheduling, pattern matching, and recreational puzzles like Sudoku.

Given a graph, the goal of the minimum graph coloring problem is to find a valid coloring that uses the minimum number of colors. Let $G = (V, E)$ be the graph with vertices V and edges $E \subseteq V \times V$. For simplicity, we assume that the edges in E are ordered such that if $(v, w) \in E$ then $v < w$. Let k be the maximum number of colors that will be considered, and let $C = 1 \ldots k$. We consider the following decision variables: (1) $x_{v,c}$ is one if vertex v is colored with color c and zero otherwise, and (2) y is the minimum number of colors that are used.

We can represent the minimum graph coloring problem with the following integer program (IP):

$$\begin{array}{lll} \min & y & \\ \text{s.t.} & \sum_{c \in C} x_{v,c} = 1 & \forall v \in V \\ & x_{v,c} + x_{w,c} \leq 1 & \forall (v,w) \in E \\ & y \geq c \cdot x_{v,c} & \\ & x_{v,c} \in \{0,1\} & \forall v \in V, c \in C \end{array} \tag{1.1}$$

The first constraint indicates that each vertex is colored with exactly one color. The second constraint indicates that vertices that are connected by an edge must have different colors. The next constraint defines a lower bound on y that guarantees that y will be greater than the number of colors used in a solution. The final constraint defines $x_{v,c}$ as a binary variable.

As discussed earlier, this IP is an abstract model for the minimum graph coloring problem. The parameter values associated with the graph and maximum number of colors are not defined in this model. Figure 1.1 shows the formulation of a Pyomo model for this problem; Appendix B provides Python tutorial for readers who are new to Python. This formulation consists of Python commands that define an `AbstractModel` object, and then define attributes of this object for the sets, pa-

rameters, variables, constraints, and objectives in this model. Note that all of these modeling components are explicitly associated with a particular model instance. This allows Pyomo to automatically manage the naming of modeling components, and it naturally segregates modeling components within different model objects.

```
1  # Python import statement
2  from coopr.pyomo import *
3
4  # Create a Pyomo model object
5  model = AbstractModel()
6
7  # Define model sets and parameters
8  model.vertices = Set()
9  model.edges = Set(within=model.vertices*model.vertices)
10 model.ncolors = Param(within=PositiveIntegers)
11 model.colors = RangeSet(1,model.ncolors)
12
13 # Define model variables
14 model.x = Var(model.vertices, model.colors, within=Binary)
15 model.y = Var()
16
17 # Each node is colored with one color
18 def node_coloring_rule(model, v):
19     return sum(model.x[v,c] for c in model.colors) == 1
20 model.node_coloring = Constraint(model.vertices,
21                                  rule=node_coloring_rule)
22
23 # Nodes that share an edge cannot be colored the same
24 def edge_coloring_rule(model, v, w, c):
25     return model.x[v,c] + model.x[w,c] <= 1
26 model.edge_coloring = Constraint(model.edges, model.colors,
27                                  rule=edge_coloring_rule)
28
29 # Provide a lower bound on the minimum number of colors
30 # that are needed
31 def min_coloring_rule(model, v, c):
32     return model.y >= c * model.x[v,c]
33 model.min_coloring = Constraint(model.vertices,
34                                  model.colors,
35                                  rule=min_coloring_rule)
36
37 # Minimize the number of colors that are needed
38 model.obj = Objective(expr=model.y)
```

Fig. 1.1 An abstract Pyomo formulation for the minimum graph coloring problem.

Line 2 is a standard Python import statement that adds all of the symbols in `coopr.pyomo` to the current Python namespace. Pyomo is a subpackage of Coopr [17], a collection of Python software packages for formulating and analyzing optimization models. Line 5 creates the `model` object, which is a class instance of

the `AbstractModel` component class. Lines 8–11 define sets and parameters of this model using the `Set`, `Param`, and `RangeSet` classes. Lines 14 and 15 define the decision variables in this model. Note that y is a single variable, while x is declared as two-dimensional array of variables.

The remaining lines define the constraints and objectives for this model. The `Objective` and `Constraint` classes typically require a `rule` option that specifies how these expressions are constructed. A rule is a function that takes one or more arguments and returns an expression that defines the constraint or objective. The last argument in a rule is the model of the corresponding objective or constraint, and the preceding arguments are index values for the objective or constraint that is being defined. As with other modeling components, the non-keyword arguments define the sets used to index constraint and variable objects.

When compared with other AMLs, Pyomo models are clearly more verbose (e.g., see Hart et al. [33]). However, this example illustrates how Python's clean syntax allows Pyomo to express mathematical concepts intuitively and concisely. Aside from the Pyomo classes, this entire example employs standard Python syntax and methods. For example, line 19 employs Python's generator syntax to iterate over all elements of the `colors` set and apply the Python `sum` function to the result. Although Pyomo does include some utility functions to simplify the construction of expressions, Pyomo does not rely on sophisticated extensions of core Python functionality.

1.4 Motivating Pyomo

AMLs have been used to develop optimization modeling applications for nearly four decades [40] including a variety of commercial and research modeling tools that are commonly used in a wide range of scientific and business applications. In point of fact, these AMLs can be used to develop similar formulations for applications like graph coloring.

The goal of Pyomo is to provide a platform for expressing optimization models that supports the central ideas of modern AMLs within a framework that promotes flexibility, extensibility, portability, and maintainability. Pyomo is built within Python to leverage the rich capabilities of this modern programming language, and it employs open source license and software management to promote the distribution and extension of this software. Pyomo is a software package within Coopr, a COmmon Optimization Python Repository [17]. Coopr supports a generic optimization process that can be applied to Pyomo models, and it defines a variety of optimization components that can be used to customize the optimization process (e.g., solver interfaces, parallelization frameworks, etc.).

1.4.1 Open Source

A distinguishing feature of Pyomo is its management as an open source project. Open source is a software development methodology that promises better quality, higher reliability, more flexibility, and lower cost through flexible software licenses and community-based software management. Although open source optimization solvers are widely available in packages like COIN-OR [15], surprisingly few open source tools have been developed to model optimization applications.

When managed well, open source projects facilitate transparency in software design and implementation. Because any developer can study and modify the software, bugs and performance limitations can be identified and resolved by a wide range of developers with diverse software experience. Consequently, there is growing evidence that managing software as open source can improve its reliability and that open source software exhibits similar defect evolution patterns as closed source software [5, 63].

Open source licenses promote the use of software without commercial limitations, though many open source licenses allow for the integration of open source software in commercial applications. Open source licenses also provide a mechanism for integrating contributions from a diverse community of developers. Additionally, open source licenses are increasingly valued in government and commercial environments due to their flexibility [16]. For example, open source licenses support open standards and allow users to avoid being locked into a single vendor. Software with open source licenses are easier to deploy since they do not have complex and costly licenses and copy protection software (which can be cost prohibitive for enterprise deployments).

The use of an open source software development model is not a panacea; ensuring high reliability of the software still requires careful software management and a committed developer community. Pyomo is included within the Coopr software, which is managed as an open source package by Sandia National Laboratories [17] and within the COIN-OR repository [15]. Coopr developer and user mailing lists are managed on Google Groups, and the Coopr software is licensed under the BSD [13], which has few restrictions for commercial use.

1.4.2 Customizable Capability

Few AMLs provide capabilities for customizing the modeling and optimization processes. Pyomo's open source project model allows a diverse range of developers to prototype new capabilities. Thus, developers can customize the software for specific applications, and can prototype capabilities that may eventually be integrated into future software releases.

More generally, Pyomo is designed to support a "stone soup" development model in which each developer "scratches their own itch." A key element of this design is the plug-in framework that Pyomo uses to integrate components like optimizers,

optimizer managers, and optimizer model format converters. The plug-in 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.

Similarly, the Coopr software is decomposed into a variety of Python packages that reflect different aspects of the modeling and optimization process. This decomposition leverages advanced features in Python to integrate these diverse package within a common `coopr` namespace. Thus, the end-user is not exposed to this complexity. However, this decomposition promotes the integration of third party packages. For example, the COIN-OR `Coin Bazaar` project hosts projects like `coopr.neos` that extend Coopr's functionality to add new solver interfaces.

1.4.3 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). By contrast, *loosely coupled* modeling tools apply external optimization executables (e.g., through the use of system calls). Of course, these options are not exclusive, and a goal of Pyomo is to support both types of solver interfaces.

This design goal has led to a distinction in Pyomo between model formulation and optimizer execution. Pyomo models are formulated in Python, a high-level programming language. Solvers can be written in either in Python or in compiled, low-level languages. Thus, Pyomo supports a two-language approach that leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations.

1.4.4 Modern Programming Language

Another goal of Pyomo is to leverage a modern high-level programming language to support the definition of optimization models. In this manner, Pyomo is similar to tools like FlopC++ [22] and OptimJ [46], which support modeling in C++ and Java, respectively. The use of a widely used high-level programming language like Python has several advantages:

Extensibility and Robustness Python 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 do not require changes to Python itself but instead involve defining additional classes and compute routines that can be immediately leveraged in the modeling process.

Software Management and Documentation Because Pyomo relies on Python, support for a domain-specific AML is not a long-term factor in the management of Pyomo. Additionally, Python is well-documented by dozens of books, and there is a large online community to provide feedback to new users. Thus, there are many more resources for Pyomo than there are with domain-specific AMLs.

Standard Libraries Python has a rich set of libraries for tackling just about every programming task. For example, standard libraries can support capabilities like data integration (e.g., working with spreadsheets), thereby avoiding the need to directly support such capabilities within Pyomo.

Scripting Python directly supports modern programming language features, including classes, looping and procedural constructs, and first-class functions. However, modern AMLs have only gradually introduced these *scripting* features. Thus, Python provides a richer environment for developing complex models.

Portability Python works on a diverse range of computing platforms, and in particular it supports Microsoft Windows, Mac OS, and Linux. Thus, Pyomo can be used in most scientific and business applications.

1.5 Getting Started

Several chapters are included in the appendices to assist users when getting started with Pyomo and Coopr. Appendix A provides detailed installation instructions for Coopr; Appendix B provides a brief tutorial of the Python scripting language. Appendix C summarizes the optimization solvers that are currently supported by Coopr, including solvers in commonly available extension packages.

The examples in this book assume that the following software is installed:

- Either Python 2.6 or 2.7 can be used. The CPython installations are required for Coopr 3.1.
- Coopr 3.1 is used throughout the book.
- The `PyYAML` package was used to generate human readable output of solver results in the YAML format. Note that this package is not strictly necessary when using Coopr; the default output format for solver results is JSON.
- The `GLPK` [29] solver is used to generate output for most examples in this book. Other LP and MILP solvers can be used for these examples, but the `GLPK` software is easily installed.
- The `IPOPT` [37] solver is used to generate output for the nonlinear models in Chapter 8. Other nonlinear optimizers can be easily used for these examples if they are compiled with the AMPL Solver Library [26].
- The `CPLEX` [18] solver is used to generate output for the stochastic programming models in Chapter 9 and the Benders example in Chapter 10. This commercial solver provides capabilities needed for these examples that are not commonly available in open source optimization software (e.g., optimization of quadratic integer programs).
- The `matplotlib` is used to generate the plots in Chapters 8 and 10.

Table	Page	Summary
3.1	29	Modeling components in Pyomo that are used to create models.
4.2	44	Predefined virtual sets in Pyomo
5.1	60	Pyomo utility functions that support the construction and evaluation of model expressions.
7.1	95	Call-back functions that can be used in a Pyomo model file to customize the workflow in the `pyomo` command.
8.1	107	Python operators that are redefined to generate Pyomo expressions.
8.2	107	Functions supported by Pyomo for the definition of nonlinear expressions.
C.1	226	The Python packages that are included in the Coopr 3.1 release.
C.2	227	Predefined solver interfaces supported by Coopr.

Table 1.1 Tables in this book that describe the key classes and functions in Pyomo.

1.6 Book Summary

The chapters in this book are organized around a variety of overlapping themes:

Pyomo Overview This chapter (Chapter 1) provides a broad introduction to Pyomo; Chapter 2 provides an overview of modeling with Pyomo and the design philosophy that has guided Pyomo. Finally, Appendix C provides some perspective on the role of other Coopr packages, and information on Coopr resources (e.g., solver interfaces). Table 1.1 provides a quick reference for the tables in this book that enumerate the classes and functions that are defined by Pyomo.

Pyomo Modeling Components As with other modern AMLs, Pyomo models are defined with a standard set of modeling components. The overview in Chapter 2 includes a discussion of the core modeling components supported by Pyomo. Further details on these components are provided in Chapter 3, Chapter 4, and Chapter 5.

Using Pyomo The overview in Chapter 2 includes a brief description of how Pyomo models can be created and optimized within Python and using the `pyomo` command. Chapter 10 provides further detail and examples that illustrate Pyomo's advanced features. Chapter 6 describes how data command files can be used to define the data in a model using commands that are similar to those used by AMPL.

Advanced Modeling Capabilities Two advanced modeling capabilities are highlighted to provide further detail of how they are managed within Pyomo. Chapter 8 describes how nonlinear models are managed and how solver interfaces for nonlinear solvers differ from LP and IP solvers. Chapter 9 describes how stochastic programs are managed with the PySP extension package, including solvers like Progressive Hedging.

Examples Although short Python snippets are used throughout the book, many chapters contain complete examples that illustrate features and use-cases for Pyomo. Examples are included as chapter appendices to enhance the readability of the chapters. Additionally, these examples are also included in table of contents to help the reader navigate the book.

1.7 Discussion

Each chapter in this book ends with a discussion about the material presented. These sections provide some perspective about each chapter by discussing related design decisions, missing capabilities, and other features that are beyond the scope of this book. We hope that these sections will both fill in the gaps in our presentation as well as illustrate the strengths and weaknesses of Pyomo.

These sections will also be used to compare and contrast Pyomo with other Python modeling tools. In recent years, a variety of developers have realized that Python's clean syntax and rich set of supporting libraries make it an excellent choice for optimization modeling [33]. A variety of open source software packages provide optimization modeling capabilities in Python, such as PuLP [47], APLEpy [6], and OpenOpt [45]. Additionally, there are many Python-based solver interface packages, including open source packages like PyGlpk [48] and pyipopt [49] as well as commercial Python interfaces for CPLEX [18] and GUROBI [31].

Several features distinguish Pyomo from these other Python-based optimization modeling tools. First, Pyomo's integration with Coopr provides mechanisms for extending the core modeling and optimization functionality without requiring edits to Pyomo itself. Second, Pyomo supports the definition of both concrete and abstract models. This allows the user a lot of flexibility for determining how closely data is integrated with a model definition. Finally, Pyomo can support a broad class of optimization models, including both standard linear programs as well as general nonlinear optimization models and stochastic programs.

Finally, we note that the examples in this book are taken from the Coopr 3.1 release, which includes version 3.0 of `coopr.pyomo`, which defines Pyomo. Although the Coopr software is actively developed by a variety of research groups, new software developments are tested with these examples to ensure the stability of this software.

Chapter 2
Pyomo Modeling Strategies

Abstract This chapter illustrates different strategies for formulating and optimizing algebraic optimization models using Pyomo. We provide a brief overview of the core modeling components supported by Pyomo. Then, we describe how to formulate both concrete and abstract models with Pyomo. Finally, we provide a brief tutorial on how these models can be analyzed with the `pyomo` command.

2.1 Modeling Components

Pyomo supports an object-oriented design for the definition of optimization models. The basic steps of a simple modeling process are as follows:

1. Create model and declare components
2. Instantiate the model
3. Apply solver
4. Interrogate solver results

In practice, these steps may be applied repeatedly with different data or with different constraints applied to the model. However, we focus on this simple modeling process to illustrate different strategies for modeling with Pyomo.

A Pyomo *model* consists of a collection of modeling *components* that define different aspects of the model. Pyomo includes the modeling components that are commonly supported by modern AMLs: index sets, symbolic parameters, decision variables, objectives, and constraints. These modeling components are defined in Pyomo through the following Python classes:

`Set`	set data that is used to define a model instance
`Param`	parameter data that is used to define a model instance
`Var`	decision variables in a model
`Objective`	expressions that are minimized or maximized in a model
`Constraint`	constraint expressions in a model

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_2,

Two model classes provide a context for defining and initializing these modeling components: `ConcreteModel` and `AbstractModel`.[1] For example, modeling components can be directly added to an `AbstractModel` object as an attribute of the object:

```
model = AbstractModel()
model.I = Set()
model.p = Param(model.I)
```

The `model` object is a class instance of the `AbstractModel` class, and `model.I` is a `Set` object that is contained by this model. Many modeling components in Pyomo can be optionally specified as *indexed components*: collections of components that are referenced using one or more values. In this example, the parameter `p` is indexed with set `I`.

NOTE: For simplicity, many short examples in this book omit the Python import statement for `coopr.pyomo`. These examples assume that this package has been previously imported with the following command:

```
from coopr.pyomo import *
```

This command imports all of the classes, functions and data from the `coopr.pyomo` package; see Appendix B for further details about importing Python packages.

The following sections describe different strategies for creating models with Pyomo. We begin with strategies for generating concrete models and progress to more general strategies for abstract models. To illustrate this flexibility, we consider a generalization of the simple LP that we introduced in Chapter 1:

$$\begin{array}{lll} \min & \sum_{i=1}^{n} c_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} a_{ij} x_i \geq b_j & \forall j = 1 \ldots m \\ & x_i \geq 0 & \forall i = 1 \ldots n \end{array} \tag{2.1}$$

The following LP is a model instance (i.e., a model object with fully specified data) of the abstract LP 2.1:

$$\begin{array}{ll} \min & x_1 + 2x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & 2x_1 + 5x_2 \geq 2 \\ & x_1, x_2 \geq 0 \end{array} \tag{2.2}$$

[1] In versions 1.x and 2.x of Coopr, the `Model` class serves as an alias for the `AbstractModel` class. Both the aliasing and the `Model` class will be deprecated in Coopr 3.1 and subsequent releases.

The examples in this chapter illustrate different ways of formulating both abstract and concrete models.

We conclude this chapter with a brief introduction to the `pyomo` command, which can be used to optimize Pyomo models within a Unix or Windows command shell. Additionally, we summarize how Pyomo model scripts can be setup to construct and solve model instances. Note that additional detail about Pyomo modeling components can be found in Chapter 3, Chapter 4, and Chapter 5. Additional detail about the `pyomo` command and scripting with Pyomo models can be found in Chapter 7.

2.2 Concrete Models: Specifying Components Via Expressions

The simplest way to express a concrete model is by directly creating the modeling components with expressions that specify numerical values; the alternative, two-step process involving an abstract model is described in Section 2.4. The `ConcreteModel` class is used to define these types of models:

```
model = ConcreteModel()
```

In contrast to `AbstractModel` objects, model components contained in `ConcreteModel` objects are immediately initialized as they are added to the model instance.

Decision variables are required to define non-trivial expressions in Pyomo. In the simplest case, we can define each decision variable separately:

```
model.x_1 = Var(within=NonNegativeReals)
model.x_2 = Var(within=NonNegativeReals)
```

The `within` argument used in these declarations constrains the values of the defined variables to the set of non-negative real numbers, in this case by having Pyomo to automatically generate the associated lower bound constraints.

An optimization objective is defined by passing an algebraic expression involving numeric constants and Pyomo variables as an argument – via the `expr` keyword – to the `Objective` class constructor:

```
model.obj = Objective(expr=model.x_1 + 2*model.x_2)
```

The objective expression is defined implicitly by Pyomo, using overloading of the Python multiplication and addition operators. This is done automatically so the modeller need only give the expression as shown in the example. Similarly, constraints of a concrete model are defined by passing an expression as an argument – again via the `expr` keyword – to the `Constraint` class constructor:

```
model.con1 = Constraint(expr=3*model.x_1 + 4*model.x_2 >= 1)
model.con2 = Constraint(expr=2*model.x_1 + 5*model.x_2 >= 2)
```

Note that this expression additionally specifies a constraint bound, via overloading of the Python >= operator. Example 2.A.1 includes the entire formulation for this concrete model.

For large concrete models, the use of explicit variables is cumbersome, because it requires separate declarations for each variable in the model. To address this issue, Pyomo supports the notion of variable *indexing*. For example, the variable `x` can be defined with an explicit index set:

```
model.x = Var([1,2], within=NonNegativeReals)
```

This indexed variable can be used to specify the objectives and constraints with a natural, extensible syntax:

```
model.obj = Objective(expr=model.x[1] + 2*model.x[2])
model.con1 = Constraint(expr=3*model.x[1] + 4*model.x[2]>=1)
model.con2 = Constraint(expr=2*model.x[1] + 5*model.x[2]>=2)
```

Indexed variables allow logically related sets of variables to be grouped under a common name, and naturally differentiated via a sub-script.

Example 2.A.2 includes the formulation for the indexed concrete model of Formulation (2.2).

A limitation of the formulation in Example 2.A.2 is that the data used is explicitly expressed in the specification of the index set for `x` and the numerical values in the constraints and objective. Any change in data values requires potentially extensive modifications to the concrete model. Alternatively, this data can be symbolically specified using standard Python data structures and subsequently used in the formulation of the expressions for objectives and constraints. For example, the following declarations define Python dictionary objects that represent the data used in the objectives and constraints for Formulation (2.2):

```
N = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}
```

Using this symbolic data, the objective can be simply redefined as follows:

```
model.obj = Objective(expr=c[1]*model.x[1] + c[2]*model.x[2])
```

An even more extensible formulation of this objective uses Python iteration syntax to sum over a set of related (indexed) decision variables. The Python `sum` function and *generator* syntax can be used to provide a concise specification of a summation:

```
model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N))
```

This syntax specifies that the terms `c[i]*model.x[i]` are generated by iterating over the set `N`. As these terms are generated, the function `sum` adds them together to form the overall expression. This type of inlining of summations is a powerful feature that provides a syntax that is similar to that used in AMLs with custom languages. Example 2.A.3 includes the complete formulation of the concrete model using these external data declarations.

2.3 Concrete Models: Specifying Components Via Rules

Although the abstract formulation in Formulation (2.1) specifies a set of constraints, each constraint declared in Example 2.A.3 is defined separately. As in the case of decision variables, this modeling approach will become cumbersome for large models.

Pyomo addresses this and other, related issues by allowing modeling components to be initialized with user-defined functions, which we call *rules*. The idea is that complex initialization of a collection of constraints or objectives (for example) can be managed by a function that generates each constraint or objective expression individually. Similarly, rules can be used to construct complex sets and parameters in a generic manner.

Example 2.A.4 illustrates the use of a rule to define the constraints for Formulation (2.2) in a generic manner. The index set `M` defines the indices of the constraint `con`, and the function `con_rule` is used to construct the individual constraint expressions. The arguments to this rule are the constraint indices, in addition to the model instance that is being constructed. As with all components added to a `ConcreteModel` object, initialization is immediately executed for all values of the specified index set. Note that the name of a rule used to define a constraint (or other modeling component) is not restricted. However, some advanced Pyomo features require the use of unique rule names (e.g., see Chapter 10).

Although the function arguments for component rules are similar for all component types, the expected type of the values returned are different:

`Set`	rule must return a Python set or list object
`Param`	rule must return an integer or float value
`Objective`	rule must return an expression
`Constraint`	rule must return a constraint expression.

The distinction between an objective and constraint expression is simply the imposition of a lower or upper bound or, the case of a constraint, both. The `Set` and `Param` classes also support direct initialization with the `initialize` argument. This argument allows the user to specify the data that initializes this class, thereby avoiding the need to specify a Python function for a `rule` argument.

Example 2.A.5 illustrates the use of `rule` and `initialize` arguments in the context of the full spectrum of modeling components. This example is more verbose than Example 2.A.4, and represents the model in a less flexible way. Thus, the modeling representation in Example 2.A.4 might be preferable. However, Example 2.A.5 avoids the specification of global data, which promotes encapsulation that may be important in complex applications.

2.4 Abstract Models

In many contexts a strong separation of model and data is either helpful or necessary. Model data may not be immediately available, or it may be stored in an external database or spreadsheet. Alternatively, it may simply be desirable to represent a model in an abstract form that is independent of the manner in which the data is managed.

A simple strategy for managing models abstractly is to create a Python function whose arguments reflect the data that is needed to create a concrete model. Example 2.A.6 illustrates this approach using the concrete model developed in Example 2.A.4. The function `create_model` has arguments `N`, `M`, `c`, `a`, and `b` that represent the data needed to create a concrete model for Formulation (2.1). Consequently, this function provides a strong separation of model and data, and it can be viewed as a constructor for an abstract model. The code in Examples 2.A.4 and 2.A.6 can be executed using the `pyomo` command (see Section 2.5), ultimately yielding identical model instances and therefore identical optimization solutions.

Pyomo also supports the definition of *abstract* models that are defined independent of any specific data. Example 2.A.7.1 illustrates the definition of an abstract model in Pyomo for Formulation (2.1). In contrast to concrete models, abstract models strictly define the existence of model components and the relationships between them (e.g., the fact that set `N` is used to index parameter `c`).

The abstract model in Example 2.A.7.1 differs from the concrete model in Example 2.A.5 in three main respects. The first difference is that the class `AbstractModel` is used instead of `ConcreteModel`. This informs Pyomo that this is an abstract model that will be constructed with auxiliary data. Because data is not immediately available, the construction of modeling components is deferred until a model instance is generated. The second difference is that declarations of data components do not contain references to the data that will be used to construct these components. The final difference is that all objective and constraint components must be defined via rules. This is a requirement in abstract models, since the construction of these components ultimately depends on the availability of specific data.

NOTE: Pyomo does allow for hybrid models where some components are initialized with data while others are defined abstractly. Further, Pyomo data components can be defined with default values that are used when data is not specified. These options are discussed in Chapters 3, 4, and 5. However, with few exceptions these hybrid models must be defined with the `AbstractModel` class, and thus Pyomo treats them as an abstract model.

Pyomo includes a rich set of options for initializing an abstract model to create a model instance that can then be optimized (see Chapter 6). A simple strategy is to supply a *data command file* that specifies values for set and parameter data.

The syntax of Pyomo's data command files is very similar to the data command syntax supported by AMPL [4]. A goal of Pyomo is to support the same syntax for `set` and `param` data commands that is used in AMPL. However, the syntax for other commands relating to file inclusion and table imports and exports is somewhat different (see Chapter 6). Example 2.A.7.2 shows a file of data commands that can be used to initialize the abstract model in Example 2.A.7.1. This process is described in Section 2.5.

2.5 Optimizing Models

The previous sections have outlined different strategies for creating and populating a Pyomo model object. After the modeler has specified how this will be done, the Python commands for doing it are generally invoked using an executable script. The most straightforward way is to invoke the `pyomo` command that calls a pre-written script to perform optimization in a standard manner. Alternatively, a script can be created to customize the process using solver components from Coopr as introduced in Section 2.5.2.

2.5.1 Optimization with the `pyomo` Command

The Pyomo software distribution includes the `pyomo` command that can be used to construct a Pyomo model, create a model instance from user-supplied data (in the case of abstract models), apply an optimizer, and summarize the results. For example, the following command line optimizes the `concrete1.py` model using the default LP solver `glpk`:

```
pyomo --solver=glpk concrete1.py
```

Similarly, the following command line optimizes the `abstract5.py` model using data in `abstract5.dat`, also using `glpk`:

```
pyomo --solver=glpk abstract5.py abstract5.dat
```

When the `pyomo` command loads a user-defined Pyomo model, by default it looks for a `ConcreteModel` or `AbstractModel` named `model` in the supplied Python file. We have used this name for all models introduced in this chapter. However, if a name other than `model` is used, this can be specified via the `pyomo` option `--model-name=MODEL_NAME` (e.g., `--model-name=mymodel`).

The `pyomo` command automatically executes the following steps:

1. Create a model
2. Read the instance data (if applicable)
3. Generate a model instance (if the model is abstract)
4. Apply simple preprocessors to the model instance

5. Apply a solver to the model instance
6. Load the results into the model instance
7. Display the solver results

A variety of optional command-line arguments are provided to further guide and provide additional information about the optimization process; documentation of the various available options is available by specifying the `--help` option. Options can control how much or even if debugging information is printed, including logging information generated by the optimizer and a summary of the model generated by Pyomo. Further, Pyomo can be configured to keep all intermediate files used during optimization, which facilitates debugging of the model construction process. See Chapter 7 for further details about this command.

2.5.2 Optimization Scripts

Scripts that control the optimization process provide users with powerful programmability. To introduce the topic, we describe a simple Python script to perform optimization of a Pyomo model instance. Suppose that the model in Example 2.A.1 is stored in the file `concrete1.py`. The script in Example 2.A.8 can then be used to import this model, display the created model, create a solver interface, perform optimization, and display the results. Note that this script does not rely on the `coopr.pyomo` package directly. Pyomo is only used to create an optimization model. Subsequent optimization and analysis of this model is handled in a generic manner with other Coopr packages.

Once a Pyomo model has been created, it can be printed using the `pprint` method:

```
model.pprint()
```

This command summarizes the information in the Pyomo model. For concrete models, this includes the constraint and objective expressions. In abstract models, this information is omitted unless the model object has been constructed with externally supplied data.

Before performing optimization, Pyomo needs to perform various preprocessing steps to collect variables, simplify expressions, and other pre-optimization tasks. Such preprocessing is automatically performed by the `create` method of both the concrete and abstract Pyomo models:

```
instance = model.create()
instance.pprint()
```

For concrete models this method simply returns the model instance, with various annotations performed by the preprocessor. For abstract models, additional arguments specifying the data used to construct the model instance must be supplied. We note that Pyomo's preprocessing capabilities do not currently involve the *presolve* oper-

ations that are commonly employed by industrial AMLs to simplify models before sending them to an optimizer.

Next, we apply an optimizer to find an optimal solution to our model instance. For example, the GLPK [29] linear programming solver can be used within Pyomo as follows:

```
opt = SolverFactory("glpk")
results = opt.solve(instance)
```

The first line in this example creates a Python object to interface to the GLPK solver. The Pyomo model instance is then optimized, with the solver returning an object that contains the solution(s) generated during optimization. This optimization process is executed using components of the `coopr.opt` package, which manage the setup and execution of optimizers. Additionally, 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()
```

This output is in the YAML format [62], which is a highly structured data format that is also human readable.

The process for optimizing abstract models (given a data source) is only slightly different. Suppose that the model in Example 2.A.7.1 is stored in the file `abstract5.py`, while the data in Example 2.A.7.2 is stored in the file `abstract5.dat`. The script in Example 2.A.9 imports this model, creates a model instance from the supplied data, creates a solver interface, performs optimization, and displays the results. The primary difference (other than the use of an abstract model) is the specification of a data file, which defines the sets and parameters that are used to construct the model instance.

2.6 Discussion

Perhaps the two Python modeling tools most similar to Pyomo are PuLP [47] and APLEpy [6]. These packages both support the construction of concrete models using Python objects. However, Pyomo is clearly distinguished by its ability to support the definition of abstract models. Although most of this chapter has focused on examples using concrete models, most of the development effort in Pyomo has focused on the mechanisms that support abstract models. In fact, the `ConcreteModel` class is just a specialization of that mechanism to immediately construct model components!

Another distinguishing aspect of Pyomo is the fact that models are complete and self-contained Python objects that a user creates. We believe that this is a feature, since it allows the user to create and managed multiple models simultaneously. However, we have heard users complain about the additional syntax that this re-

quires. In this way, PuLP and APLEpy may be a bit simpler for beginning users, perhaps at the expense of some object-orientation.

In practice, we have observed that users working on abstract models tend to work with the `pyomo` command, while users working on concrete models tend to work with Python scripts. The `pyomo` command is relatively mature, since its interface is well-defined. However, Pyomo's scripting capabilities are still being extended. Although the simple examples shown in this chapter are straightforward, more complex examples that we have developed often rely on features of Coopr and Pyomo that were not intended for general, public use. Consequently, we expect that the scripting capabilities of Coopr and Pyomo will significantly evolve in future releases of this software.

2.A Examples

2.A.1 Concrete Pyomo Model with Explicit Variables

A concrete Pyomo model for Formulation (2.2) using explicit variables.

```
from coopr.pyomo import *

model = ConcreteModel()
model.x_1 = Var(within=NonNegativeReals)
model.x_2 = Var(within=NonNegativeReals)
model.obj = Objective(expr=model.x_1 + 2*model.x_2)
model.con1 = Constraint(expr=3*model.x_1 + 4*model.x_2 >= 1)
model.con2 = Constraint(expr=2*model.x_1 + 5*model.x_2 >= 2)
```

2.A.2 Concrete Pyomo Model with Indexed Variables

A concrete Pyomo model for Formulation (2.2) using indexed variables.

```
from coopr.pyomo import *

model = ConcreteModel()
model.x = Var([1,2], within=NonNegativeReals)
model.obj = Objective(expr=model.x[1] + 2*model.x[2])
model.con1 = Constraint(expr=3*model.x[1] + 4*model.x[2]>=1)
model.con2 = Constraint(expr=2*model.x[1] + 5*model.x[2]>=2)
```

2.A.3 Concrete Pyomo Model with External Data

A concrete Pyomo model for Formulation (2.2) with (a) external data declarations and (b) expressions defined using Python's generator syntax.

```
from coopr.pyomo import *

N = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}

model = ConcreteModel()
model.x = Var(N, within=NonNegativeReals)
model.obj = Objective(expr=
            sum(c[i]*model.x[i] for i in N))
model.con1 = Constraint(expr=
            sum(a[i,1]*model.x[i] for i in N) >= b[1])
model.con2 = Constraint(expr=
            sum(a[i,2]*model.x[i] for i in N) >= b[2])
```

2.A.4 Concrete Pyomo Model with Constraint Rules

A concrete Pyomo model for Formulation (2.2) using a constraint rule to create constraints `con`.

```
from coopr.pyomo import *

N = [1,2]
M = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}

model = ConcreteModel()
model.x = Var(N, within=NonNegativeReals)
model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N))

def con_rule(model, m):
   return sum(a[i,m]*model.x[i] for i in N) >= b[m]
model.con = Constraint(M, rule=con_rule)
```

2.A.5 Concrete Pyomo Model with Abstract Component Declarations

A concrete Pyomo model for Formulation (2.2) that uses `rule` and `initialize` arguments for all modeling components.

```
from coopr.pyomo import *


model = ConcreteModel()

def N_rule(model):
   return [1,2]
model.N = Set(rule=N_rule)

model.M = Set(initialize=[1,2])
model.c = Param(model.N, initialize={1:1, 2:2})
model.a = Param(model.N, model.M,
         initialize={(1,1):3, (2,1):4, (1,2):2, (2,2):5})
model.b = Param(model.M, initialize={1:1, 2:2})

model.x = Var(model.N, within=NonNegativeReals)

def obj_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = Objective(rule=obj_rule)

def con_rule(model, m):
   return sum(model.a[i,m]*model.x[i] for i in model.N) \
              >= model.b[m]
model.con = Constraint(model.M, rule=con_rule)
```

2.A.6 Using a Function to Construct a Concrete Pyomo Model

A function that creates a concrete Pyomo model for Formulation (2.1) using only data provide in the argument list.

```
from coopr.pyomo import *


def create_model(N=[], M=[], c={}, a={}, b={}):
   model = ConcreteModel()
   model.x = Var(N, within=NonNegativeReals)
   model.obj = Objective(expr=
                  sum(c[i]*model.x[i] for i in N))

   def con_rule(model, m):
      return sum(a[i,m]*model.x[i] for i in N) >= b[m]
   model.con = Constraint(M, rule=con_rule)
   return model

model = create_model(N = [1,2], M = [1,2], c = {1:1, 2:2},
            a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5},
            b = {1:1, 2:2})
```

2.A.7 Abstract Pyomo Model

2.A.7.1 Pyomo Model

An abstract Pyomo model for Formulation (2.1).

```
from coopr.pyomo import *

model = AbstractModel()

model.N = Set()
model.M = Set()
model.c = Param(model.N)
model.a = Param(model.N, model.M)
model.b = Param(model.M)

model.x = Var(model.N, within=NonNegativeReals)

def obj_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = Objective(rule=obj_rule)

def con_rule(model, m):
   return sum(model.a[i,m]*model.x[i] for i in model.N) \
               >= model.b[m]
model.con = Constraint(model.M, rule=con_rule)
```

2.A.7.2 Data Commands

Data commands for the abstract Pyomo model in Example 2.A.7.1 that are used to generate the concrete model in Formulation (2.2).

```
set N := 1 2 ;

set M := 1 2 ;

param c :=
1 1
2 2 ;

param a :=
1 1 3
2 1 4
1 2 2
2 2 5 ;

param b :=
1 1
2 2 ;
```

2.A.8 A Python Script that Optimizes a Concrete Pyomo Model

A Python script that creates the concrete Pyomo model in Example 2.A.1 and performs optimization using the GLPK linear programming solver.

```
from coopr.opt import SolverFactory
from concrete1 import model

model.pprint()

instance = model.create()
instance.pprint()

opt = SolverFactory("glpk")
results = opt.solve(instance)

results.write()
```

2.A.9 A Python Script that Optimizes an Abstract Pyomo Model

A Python script that creates a model instance from the abstract Pyomo model in Example 2.A.7.1 and performs optimization using the GLPK linear programming solver.

```
from coopr.opt import SolverFactory
from abstract5 import model

model.pprint()

instance = model.create('abstract5.dat')
instance.pprint()

opt = SolverFactory("glpk")
results = opt.solve(instance)

results.write()
```

Chapter 3
Model Components: Variables, Objectives, and Constraints

Abstract A Pyomo model is populated with modeling components that define model data, as well as model structures like objectives, and constraints. This chapter describes the Pyomo classes for modeling components that define variables, objectives and constraints, which are the core components needed to define a Pyomo model.

3.1 Modeling with Components

Pyomo supports an object-oriented mechanism for representing mathematical models. A model object is constructed, and then modeling components are added as attributes of this object. Pyomo includes the modeling components that are commonly supported by modern AMLs: index sets, symbolic parameters, decision variables, objectives, and constraints. Table 3.1 describes the components that are provided by Pyomo.

`BuildAction`	A component that injects arbitrary actions into the model construction process
`BuildCheck`	A component that performs tests during the model construction process
`Constraint`	Constraint expressions in a model
`ConstraintList`	A list of constraints in a model
`Objective`	Expressions that are minimized or maximized in a model
`Param`	Parameter data that is used to define a model instance
`Piecewise`	A variable constrained by a piecewise function
`RangeSet`	A sequence of numeric values
`SOSConstraint`	SOS constraint expressions in a model
`Set`	Set data that is used to define a model instance
`Var`	Decision variables in a model

Table 3.1 Modeling components in Pyomo that are used to create models.

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_3,

doc	A string describing this variable
domain	A set that is used to validate variable values
name	The variable name

Table 3.2 Attributes of Var objects.

Two types of models can be created in Pyomo. Concrete models are formulated using the 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. The ConcreteModel class immediately initializes the modeling components that are added to it.

Abstract models are declared in a similar fashion, but offer the advantage of additional flexibility at the cost of computational effort and memory. The AbstractModel is used for applications where the model's components are created before the model data is available. Abstract models include additional components for managing the declaration of model data, as well as logic for data initialization and validation. Nevertheless, while the declaration and use of model components is similar for both abstract and concrete models, are a few differences are described throughout this chapter.

This chapter describes the Var, Objective, and Constraint components, which define the decision variables and algebraic equations used in a model. Additionally, this chapter describes components for specialized classes of constraints. Chapter 4 describes the Set and Param components, which define the data used to construct concrete and abstract models. Although a complete description of Set components is deferred until Chapter 4, the examples below illustrate how Set objects are used to index variable, objective, and constraint components. Finally, Chapter 5 describes other components included in Pyomo that support model debugging and error checking, along with Pyomo utility functions that simplify the formulation of Pyomo models.

3.2 Variables

A variable is a numerical value that is determined during optimization. Pyomo variables are managed with the Var class, which can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables can have initial values, and the value of a variable can be retrieved and set. The following sections describe the syntax for variable declarations, initialization, and validation. Table 3.2 describes the attributes of a Var object.

3.2.1 Var Declarations

A simple declaration of a `Var` object declares a single variable:

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

A variable array can also be specified by providing sets as options to the `Var` object. Multi-dimensional variable arrays can be declared by simply including a list of sets as options to the `Var` object:

```
B = [1.5, 2.5, 3.5]
model.u = Var(B)
model.C = Set()
model.t = Var(B, model.C)
```

The domain of a variable is specified with either the `domain` or `within` options:

```
model.A = Set(initialize=[1,2,3])
model.y = Var(within=model.A)
model.r = Var(domain=Reals)
model.w = Var(within=Boolean)
```

If the domain is not specified, the default is `Any`. The most commonly used domains are the virtual sets shown in Table 4.2 (see page 44).

Similarly, variable bounds can be explicitly specified with the `bounds` option:

```
model.a = Var(bounds=(0.0,None))

lower = {1:2, 2:4, 3:6}
upper = {1:3, 2:4, 3:7}
def f(model, i):
   return (lower[i], upper[i])
model.b = Var(model.A, bounds=f)
```

The `bounds` option can specify a 2-tuple with lower and upper values. Alternatively, it can specify a function that returns a 2-tuple for each variable index. Note that `None` can be used to specify that a bound is not enforced.

3.2.2 Variable Initialization

Variable values are typically determined during optimization. However, the initial value of variables can be set with the `initialize` option. This option can specify a numerical value used to initialize a variable or variable array:

```
model.za = Var(initialize=9, within=NonNegativeReals)
model.zb = Var(model.A, initialize={1:1, 2:4, 3:9})
model.zc = Var(model.A, initialize=2)
```

Additionally, this option can use a function that accepts the variable indices and model and returns the value of that variable element:

```
def g(model, i):
   return 3*i
model.m = Var(model.A, initialize=g)
```

The `reset` method of a variable sets the variable's value to the initial value:

```
model.za = 8.5
print model.za.value # 8.5
model.za.reset()
print model.za.value # 9
```

Variable values can also be set using the equality operator:

```
model.za = 8.5
model.zb[2] = 7
```

The equality operator is sometimes important as an alternative to `initialize` for providing initial values to be passed to a solver. For example, variables can also be given values in optimization scripts, as illustrated in Chapter 10.3.3.

3.2.3 Working with Variables

Variable objects have a variety of helper functions and utility methods that facilitate their use. The `float` function can be used to force a `Var` object into a floating point value:

```
print float(model.za) # 8.5
print float(model.zb[2]) # 7.0
```

Similarly, the `value` function can be used to coerce a `Var` object into its natural numerical value:

```
print value(model.za) # 8.5
print value(model.zb[2]) # 7
```

Finally, the `len` function returns the number of variables in a variable array:

```
print len(model.za) # 1
print len(model.zb) # 3
```

It is often useful to directly interact with `Var` objects. These objects have a variety of useful attributes that can be queried and set:

```
print model.zb[2].value # 7
print model.zb[2].initial # 4
print model.za.lb # 0
print model.za.ub # None
print model.zb[2].fixed # False
```

The `value` attribute is the current value of the variable, and `initial` is the value that the variable was initialized with. The values of `lb` and `ub` are the lower and

upper bounds on the variable, respectively; if possible, these values are inferred from the domain of the variable. If `fixed` is true, then this variable has a fixed value. Pyomo may preprocess these values out before calling the optimizer.

The `value`, `initial`, `lb`, `ub`, and `fixed` attributes are not included in Table 3.2 because they are only attributes of the `Var` object for singleton variables. For indexed variables, which are implemented as variable arrays, these are attributes of the object that is returned by indexing a `Var` object. E.g., `model.za.value` makes sense because `model.za` is a singleton and `model.zb[2].value` is defined because the index has been specified for the indexed variable `zb`; however, `model.zb.value` is not defined because `zb` is not a singleton.

The distinction between the component object and the component value objects is a common design pattern within Pyomo. Singleton components combine these two classes, but for indexed components the component object is distinct from the component value objects that are accessed with an index, which are implemented as `_VarValue` objects.

3.3 Objectives

3.3.1 Simple Declarations

An objective is a function involving data and variables that is either minimized or maximized by a solver. The solver searches for values of the variables that result in the best possible value of the objective function. Constraints may place restrictions on the values that the variables can take, but the objective function determines how the values of the variables will be evaluated.

Most solvers can be applied to optimization models with a single objective. A simple declaration of a `Objective` object declares a single objective:

```
model.a = Objective()
```

Some solvers can perform multi-objective optimization with two or more objectives. These objectives can either be declared separately, or with an array of objectives:

```
model.b = Objective()
model.c = Objective([1,2,3])
```

These examples are incomplete since they do not include a declaration of the functional expression for these objectives.

An `Objective` object defines the properties of the objective function such as the functional expression and an indication of whether it is to be minimized or maximized. Suppose that `model.x[1]` and `model.x[2]` have been declared as variables. The following declaration defines an objective for the model with a function that is the sum of the first variable plus twice the second:

```
model.d = Objective(expr=model.x[1] + 2*model.x[2])
```

By default, the declaration of an `Objective` object indicates that the objective is to be minimized. The `sense` option can also be used to indicate an objective that is maximized:

```
model.e = Objective(expr=model.x[1], sense=maximize)
```

3.3.2 Declarations with Rule Functions

In these examples, the `expr` option is used to specify the the function to be optimized. An alternative method for initializing an `Objective` object is to write a rule function that creates the expression that defines the objective function. The `rule` option is used to specify the name of the rule function. The following two declarations of `Objective` objects are equivalent:

```
model.f = Objective(expr=model.x[1] + 2*model.x[2])

def TheObjective(model):
    return model.x[1] + 2*model.x[2]
model.g = Objective(rule=TheObjective)

def gg_rule(model):
    return model.x[1] + 2*model.x[2]
model.gg = Objective()
```

Note that any name can be used for the rule function. However, the `rule` option can be omitted altogether if the name of the rule function equals the objective name appended with `_rule`.

When the `Objective` object is declared with a set as an argument, Pyomo iterates over all members of the set during object construction, passing each set member to the function given as the argument to the `rule` keyword.

```
def h_rule(model, i):
    return i*model.x[1] + i*i*model.x[2]
model.h = Objective([1, 2, 3, 4])
```

If multiple sets are specified in an `Objective` declaration, then Pyomo iterates over the cross product of all input sets, providing an index for each set to the rule function. The model being constructed is passed as the first argument to the rule function.

Note that the `expr` option cannot be used to initialize the `h` objective defined in this last example. The `rule` option is needed to define functions for each of the objectives. In some situations the `rule` option is preferable even for single objective declarations. Specifically, a rule function provides control over how the objective is formed or used to build up the expression. For example, the following rule function illustrates how an expression is constructed incrementally:

```
def m_rule(model):
   expr = model.x[1]
   expr += 2*model.x[2]
   return expr
model.m = Objective()
```

Similarly, the following rule function illustrates the use of scripting to control the creation of the expression:

```
p = 0.6
def n_rule(model):
   if p > 0.5:
      return model.x[1] + 2*model.x[2]
   else:
      return model.x[1] + 3*model.x[2]
   return expr
model.n = Objective()
```

In the interest of completeness, we note that this objective could also be done in a concrete model without a rule function in the following way:

```
p = 0.6
if p > 0.5:
   model.p = Objective(expr=model.x[1] + 2*model.x[2])
else:
   model.p = Objective(expr=model.x[1] + 3*model.x[2])
```

Examples of more sophisticated use of the `rule` option are given later in this book.

3.4 Constraints

A constraint defines expressions that place limits on the values of variables. The `bounds` option for a variable can also place limits on variable values, but constraints generalize this idea by allowing for expressions that place limits on many interacting variables simultaneously. The declaration of constraint expressions is similar to the declaration of objective function expressions. However, constraints differ from objectives in that the expressions are augmented with auxiliary information to specify equalities or inequalities. While objectives can be indexed, this feature is infrequently used. In contrast, constraints are commonly indexed, allowing for access to indexed parameters and variables when constructing the constraint expression.

3.4.1 Declarations with Logical Expressions

Suppose that `model.x[1]` and `model.x[2]` have been declared as variables. A simple declaration of a single, non-indexed `Constraint` object is as follows:

```
model.Diff= Constraint(expr=model.x[2]-model.x[1] <= 7.5)
```

As is the case when declaring an objective function, the expression specified by the `expr` keyword can alternatively be generated with a rule function. Thus, the `Diff` constraint can also be constructed as follows:

```
def Diff_rule(model):
   return model.x[2] - model.x[1] <= 7.5
model.Diff = Constraint()
```

Constraints can be indexed, and those indices can be used to refer to specific elements of indexed parameters and variables when constructing expressions. Consider the following code fragment for constructing a model:

```
N = [1,2,3]

a = {1:1, 2:3.1, 3:4.5}
b = {1:1, 2:2.9, 3:3.1}

model.y = Var(N, within=NonNegativeReals, initialize=0.0)

def CoverConstr_rule(model, i):
   return a[i] * model.y[i] >= b[i]
model.CoverConstr= Constraint(N)
```

The specification of indexed constraints is analogous to specification of indexed objectives. Pyomo iterates over the cross product of all input sets, providing an index for each set to the rule function, and the model being constructed is passed as the first argument to the rule function.

The previous example implements the following mathematical model:

$$\begin{array}{ll} a_i y_i \geq b_i & \forall i \in \{1,2,3\} \\ y_i \geq 0 & \forall i \in \{1,2,3\} \end{array} \tag{3.1}$$

If this constraint is part of a larger, complete model, the following constraints are generated and passed to a solver:

$$\begin{aligned} y[1] &\geq 1 \\ 3.1 \cdot y[2] &\geq 2.9 \\ 4.5 \cdot y[1] &\geq 3.1 \end{aligned}$$

Example 2.A.5 gives a more complete model that illustrates constraints that are indexed sets.

3.4.2 Declarations with Expression Tuples

Thus far, we have only considered the specification of constraints as logical expressions, either equalities (using the == operator) or inequalities (using the <= and >= operators). Alternatively, constraint expressions can be specified with a tuple (l, f, u) that is equivalent to

$$l \leq f \leq u$$

or with a tuple (l, f) that is equivalent to

$$l == f.$$

An expression involving constants, parameters, and variables can be given for f, while l and u must (with very few exceptions) contain only parameters or constants. When the tuple is given, either l or u can be specified using the Python keyword `None` to indicate a one-sided constraint. The `None` value cannot be simultaneously specified for both l and u (or for l in the case of a tuple that defines an equality constraint). This would indicate an unbounded constraint expression, which is not a constraint!

Using tuples is the most computationally efficient way to specificy constraints in Pyomo. The tuple representation is used internally within `Constraint` objects, so constraints specified as equalities and inequalities must often be re-structured to conform to the tuple representation. This re-structuring can require non-trivial computation. In general, we recommend specifying constraints using equalities and inequalities, as this approach is more intuitive to most users. If performance is an issue (as identified via profiling), then the constraints can be re-expressed as tuples.

The following constraint declaration illustrates how a tuple value is interpreted as inequalities:

```
def CapacityIneq_rule(model, i):
   return (0.25, (a[i] * model.y[i])/b[i], 1.0)
model.CapacityIneq = Constraint(N)
```

This constraint captures the mathematical concept

$$0.25 \leq \frac{a_i x_i}{b_i} \leq 1 \qquad \forall i \in \{1,2,3\} \tag{3.2}$$

which is the same as saying (after re-arranging terms) a times x must between one-fourth b and b. Similarly, the code fragment

```
def CapacityEq_rule(model, i):
   return (0, a[i] * model.y[i] - b[i])
model.CapacityEq = Constraint(N)
```

specifies an equality constraint using the tuple syntax, encoding the mathematical concept:

$$0 \leq a_i x_i - b_i \leq 0 \qquad \forall i \in \{1,2,3\} \tag{3.3}$$

or, after rearranging terms, that a times x must equal b.

Then the rule function is called for each member in the cross product of the index sets that are used to declare a constraint. In some optimization models, the constraint may not be defined for all indices. For example, particular indices may not be physically realizable. The rule function can return the constant value `Constraint.Skip` (or `Constraint.NoConstraint`) to indicate that no constraint is associated with a particular index. This technique is used in the following example, which defines a notional task scheduling constraint:

```
TimePeriods = [1,2,3,4,5]
LastTimePeriod = 5

model.StartTime = Var(TimePeriods, initialize=1.0)

def Pred_rule(model, t):
   if t == LastTimePeriod:
      return Constraint.Skip
   else:
      return model.StartTime[t] <= model.StartTime[t+1]

model.Pred = Constraint(TimePeriods)
```

The alternative is to construct a sparse index set that specifies only the valid indices in the constraint.

Several special constant values are recognized as return values for constraint rules. The value `Constraint.Skip` indicates that no constraint is being generated. When used within a constraint rule with one or more indices, those index values are skipped. The value `Constraint.Feasible` indicates that the constraint generated for the specified index is always feasible. Consequently, that constraint does not need to be generated, and it is skipped. Finally, the value `Constraint.Infeasible` indicates that the constraint generated by the specified index is infeasible. For this value, Pyomo raises an exception to inform the user, because this typically indicates an error in the model. The following example illustrates the use of these return values:

```
def C1_rule(model, i):
   if i == 1:
      return Constraint.Feasible
   if i == 2:
      return Constraint.Infeasible
   return Constraint.Skip

try:
   model.C1 = Constraint([1,2,3])
except:
   pass
```

Note that the try/except block is used to catch the exception raised when `Constraint.Infeasible` is returned.

The `simple_constraint_rule` decorator allows for a simpler semantics for constraint rules, by allowing return values `None`, `True`, and `False`. This decorator is a Python function that translates these values into the constant values

`Constraint.Skip`, `Constraint.Feasible`, and `Constraint.Infeasible`. The following example illustrates the use of this decorator:

```
model.EMPTY = Set()
model.z = Var(model.EMPTY)

@simple_constraint_rule
def C2_rule(model, i):
   if i == 1:
      return summation(model.z) < 1
   if i == 2:
      return summation(model.z) < -1
   return None

try:
   model.C2 = Constraint([1,2,3])
except:
   pass
```

The decorator is declared with the `@` symbol immediately before the rule that it modifies. The logical behavior of rule `C2_rule` is identical to `C1_rule`. However, in `C2_rule` the values `True` and `False` are returned from attempts to formulate expressions. The summation of `model.z` is zero, since the set `EMPTY` contains no elements. Thus, when `i == 1`, the return value is the expression `0 < 1`, which is `True`. Similarly, when `i == 2`, the return value is the expression `0 < -1`, which is `False`. Section 3.6 provides further discussion of simple rule decorators.

3.5 Constraint Lists

In simple applications, the model constraints may not need to be distinguished. That is, they can be imagined as belonging to a single list of constraints. This simplifies the specification of a model, because separate constraint declarations are not needed.

For example, consider Example 2.A.2. This example includes separate declarations of `Constraint` objects. The `ConstraintList` component can be used to simplify this model by combining the constraint declarations into a single component:

```
from coopr.pyomo import *

model = ConcreteModel()
model.x = Var([1,2], within=NonNegativeReals)
model.obj = Objective(expr=model.x[1] + 2*model.x[2])
model.con = ConstraintList()
model.con.add(3*model.x[1] + 4*model.x[2]>=1)
```

Similarly, Example 2.A.4 can be simplified to eliminate the explicit indexing of the constraints. The following model uses a `ConstraintList` with a rule function:

```
from coopr.pyomo import *

N = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}

model = ConcreteModel()
model.x = Var(N, within=NonNegativeReals)
model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N))

def con_rule(model, m):
   if m == 3:
      return ConstraintList.End
   return sum(a[i,m]*model.x[i] for i in N) >= b[m]
```

The first argument of the rule function is the constraint index, which starts at one. The rule function is called with increasing index values until the rule function returns `ConstraintList.End`.

The `simple_constraintlist_rule` decorator can also be used to simplify this last rule function. When this decorator is used, the constraint rule can return `None` to terminate the constraint generation:

```
@simple_constraintlist_rule
def con_rule(model, m):
   if m == 3:
      return None
   return sum(a[i,m]*model.x[i] for i in N) >= b[m]
model.con = ConstraintList(rule=con_rule)
```

3.6 Discussion

Using Rule Functions

As noted previously, there are some contexts where objective and constraint declarations need to specify the `rule` option instead of the `expr` option. Both the `rule` and `expr` options can be employed for concrete and abstract models. However, in most abstract models the `rule` option is necessary because set and parameter data in an abstract model cannot be used to construct an expression that is passed to the `expr` option. These components are not initialized when they are constructed. Consequently, their values are not defined when an objective or constraint component is constructed. However, they *are* defined when at the point where the rule function is executed.

It is generally recommended that rule functions be used to initialize objective and constraint components. The only context where the `expr` option can be used in an abstract model is where it depends on global Python data. However, this is an unusual context that does not reflect best practices for formulating a model, and it increases the changes for bugs.

Decorators for Simple Rules

Rule decorators such as `simple_constraint_rule` are a relatively new addition to Pyomo. Rule decorators support a simple, intuitive semantics for rules. Until the release of Coopr 3.1, this was the default semantics for rule functions. However, diagnosing errors in these rules has often proven difficult, even for expert Pyomo users. One serious complication is that Python functions return `None` when no return value is specified. Thus, a user can easily create a constraint rule where no return value is specified, which can be extremely difficult to diagnose in large, complex models. We have seen similar simple coding errors in which constraint rules erroneously return `True` or `False`.

In Coopr 3.1, rule functions were changed to only support specified return values. At the same time, rule decorators were introduced for backwards compatibility, and to support users who are more comfortable with the simpler syntax that they allow. Although this limits Pyomo's ability to create diagnostic error messages, users are making this choice when they apply a rule decorator.

Component Documentation

Pyomo components uniformly support the specification of the `doc` option. This option is used to specify a comment that can be used in model summaries.

SOS Constraints and Piecewise-linear Expressions

Support for special ordered set (SOS) constraints and piecewise-linear expressions is relatively new, and these capabilities have not been widely used. Thus, we do not expect these features to be as robust as other capabilities in Pyomo. Consequently, we decided not to document these modeling components.

Chapter 4
Model Components: Sets and Parameters

Abstract A Pyomo model is populated with modeling components that define model data, as well as model structures like objectives and constraints. This chapter describes the Pyomo classes for set and parameter components that define model data. These classes can be used in both concrete and abstract models, but they are particularly vital to abstract models.

4.1 Set Data

A set is a collection of data, possibly include numeric data (e.g., integer values) as well as symbolic data (e.g., strings). Pyomo set objects either contain concrete data, or they are *virtual* sets that do not contain data, but which support operations like set iteration and/or set membership validation. Several classes can be used to define sets in Pyomo models:

`Set`	A generic set declaration class
`RangeSet`	A set class that defines a range of numbers

The following subsections describe the syntax for set declarations, initialization and set validation. Table 4.1 describes the attributes of a `Set` class after it has been constructed. Table 4.2 describes the predefined *virtual* sets in Pyomo, which are principally used for validating set elements.

4.1.1 Set Declarations

A simple declaration of a `Set` object declares an unordered set of arbitrary data:

```
model.A = Set()
```

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_4,

`doc`	A string describing this set
`domain`	A super-set of this set, which is used to validate set membership
`dimen`	The *dimension* of the data in this set. Each set member is either a singleton, or a tuple with length `dimen`
`name`	The set name
`ordered`	A boolean value that indicates whether this set is ordered
`validate`	A function that a user can specify to validate new set data
`virtual`	A boolean value that indicates whether this set contains concrete set elements

Table 4.1 Attributes of `Set` objects.

`Any`	The set of all possible values
`Reals`	The set of floating point values
`PositiveReals`	The set of strictly positive floating point values
`NonPositiveReals`	The set of non-positive floating point values
`NegativeReals`	The set of strictly negative floating point values
`NonNegativeReals`	The set of non-negative floating point values
`PercentFraction`	The set of floating point values in the interval [0,1]
`Integers`	The set of integer values
`PositiveIntegers`	The set of positive integer values
`NonPositiveIntegers`	The set of non-positive integer values
`NegativeIntegers`	The set of negative integer values
`NonNegativeIntegers`	The set of non-negative integer values
`Boolean`	The set of boolean values, which can be represented as False/True, 0/1, 'False'/'True' and 'F'/'T'
`Binary`	The same as 'Boolean'

Table 4.2 Predefined virtual sets in Pyomo.

A set array can also be specified by providing a set as an option to the `Set` object, and multi-dimensional set arrays can be declared by including a list of sets as options to the `Set` object:

```
model.B = Set()
model.C = Set(model.A)
model.D = Set(model.A,model.B)
```

Similarly, standard Python types can be used to define a set index:

```
model.E = Set([1,2,3])
f = set([1,2,3])
model.F = Set(f)
```

A Python data type that is used to define an index must be iterable, and Pyomo duplicates this data in an index set within the model.

Set declarations can also use standard set operations to declare a set in a constructive fashion:

```
model.G = model.A | model.B # set union
model.H = model.B & model.A # set intersection
model.I = model.A - model.B # set difference
model.J = model.A ^ model.B # set exclusive-or
```

Also, set cross-products can be specified with the multiplication operator:

```
model.K = model.A * model.B
```

Note that this is different from the following, which specifies that L is a subset of a cross-product:

```
model.L = Set(within=model.A * model.B)
```

The declaration of L defines a condition for membership in L, but the members of L are not defined with this declaration.

4.1.2 Set Initialization

By default, a set declaration defines an abstract set in a model.

```
model.A = Set()
model.A.add(1,4,9)
```

The initialize option can also be used to specify the elements in a set:

```
model.B = Set(initialize=[2,3,4])
model.C = Set(initialize=[(1,4),(9,16)])
```

The rule option acts the same as initialize. The value of the initialize option can be a Python iterator or any Python object that supports iteration.

```
model.D = Set(initialize=set([1,4,9]))
model.E = Set(initialize=(i for i in model.B if i%2 == 0))
```

Set E is an example where a Python generator expressions is used, which allows a simple inline expression for the definition of set elements. The initialize option can also be used to specify the elements in a set array. These values are defined in a dictionary, which specifies how each array element is initialized:

```
F_init = {}
F_init[2] = [1,3,5]
F_init[3] = [2,4,6]
F_init[4] = [3,5,7]
model.F = Set([2,3,4],initialize=F_init)
```

Several types of functions can be used with the initialize option. If the function accepts a model object, then this function is expected to return an iterator containing data for the set:

```
def I_init(model):
   return ((a,b) for a in model.A for b in model.B)
model.I = Set(initialize=I_init, dimen=2)
```

The `initialize` option can also specify a function that is used to provide elements for a set array. The function accepts the array indices and model and returns the set for that array index:

```
def J_init(model, i, j):
   return range(0,i*j)
model.J = Set(model.B,model.B, initialize=J_init)
```

The previous function examples returned an iterator either for a specific set or for the specified set array index. Additionally, the `initialize` option can specify a function that is applied sequentially to generate set members. For a simple set, if the function accepts a set element number and model, then this function is expected to return the set element associated with that number:

```
def K_init(model, i):
   if i > 10:
      return Set.End
   return 2*i+1
model.K = Set(initialize=K_init)
```

In this example, the i-th set element has value $2*i+1$. The element numbers start at 1 and continue sequentially higher until the initialization function returns `Set.End`. Consequently, the previously defined set elements can be used to recursively define set elements. For example:

```
def L_init(model, z):
   if z==6:
      return Set.End
   if z==1:
      return 1
   else:
      return model.L[z-1]*(z+1)
model.L = Set(initialize=L_init)
```

The semantics of this rule can be simplified using the `simple_set_rule` decorator, which allows the rule to return `None` instead of `Set.End`:

```
@simple_set_rule
def LL_init(model, z):
   if z==6:
      return None
   if z==1:
      return 1
   else:
      return model.LL[z-1]*(z+1)
model.LL = Set(initialize=LL_init)
```

For an indexed set, the same logic is followed when the function accepts array indices along with a set element number:

```
def M_init(model, z, i, j):
   if z > 5:
      return Set.End
   return i*j+z
model.M = Set(model.B,model.B, initialize=M_init)
```

Similarly, the set element computation can employ element values for previously defined set array indices:

```
def N_init(model, z, i):
   if z==5:
      return Set.End
   if i == 5:
      return Set.End
   if z==1:
      if i==1:
         return 1
      else:
         return (z+1)
   return model.N[i][z-1]+z
model.N = Set(RangeSet(1,4), initialize=N_init, ordered=True)
```

Note that this example requires that the index set be ordered as well. The `simple_set_rule` decorator can also be used for rule functions that accept array indices.

The previous examples illustrate how data can be specified or dynamically generated to initialize a set. However, there are some contexts where it is simpler to specify what set elements should be omitted. The `filter` option can be used to specify a function that returns `True` when a element belongs in a set, and `False` otherwise. For example:

```
model.P = Set(initialize=[1,2,3,5,7])
model.Q = Set(initialize=xrange(1,10),
          filter=lambda model, x:not x in model.P)
```

Here, set `P` contains prime values, and set `Q` is the set of all numbers except for the members of `P`.

Finally, note that a set object is not immediately initialized with set data unless it is used within a concrete model. In abstract models, initialization functions like `K_init` are called during the construction of a model instance. Furthermore, these values may be overridden by later construction steps in a Pyomo model. For example, data specified in an input file will override the data specified by the `initialize` options.

4.1.3 Set Data Validation

Validation of set data is supported in two different ways. First, a superset can be specified with the `within` option:

```
model.B = Set(within=model.A)
```

When set `B` is constructed, the set elements are checked to confirm that they belong to the specified superset.

Similarly, validation of set data can be performed with the `validate` option, which is a function that returns `True` if an element belongs in this set. For example,

the following C_validate function mimics the within option:

```
def C_validate(model, value):
   return value in model.A
model.C = Set(validate=C_validate)
```

Although the within option is convenient, it can force the creation of a temporary set. For example, consider the declaration

```
model.D = Set(within=model.A*model.B)
```

In this example, the cross-product of sets A and B is needed to validate the members of set C. Pyomo creates this set implicitly and uses it for validation. By contrast, a simple validation function could be used in this example, though with a less intuitive syntax:

```
def E_validate(model, value0, value1):
   return value0 in model.A and value1 in model.B
model.E = Set(validate=E_validate, dimen=2)
```

The within option also supports validation of a set array. The elements of all sets in the array must be in this set:

```
model.F = Set([1,2,3], within=model.A)
```

Validation of set arrays can also be performed with the validate option. This is applied to all sets in the array:

```
def G_validate(model, value):
   return value in model.A
model.G = Set([1,2,3], validate=G_validate)
```

Finally, note that if both the within and validate options are specified, then the logic for both of these options are applied to validate set elements.

4.1.4 Set Options

By default, sets are unordered. That is, the internal representation may place the set elements in any order. In some cases, we need to know the order in which set elements are declared. In such cases, we can declare a set to be ordered with the ordered option:

```
model.A = Set(ordered=True)
```

Sets may contain data elements that are either singletons or k-tuples. The dimen option is used to specify the expected dimension of the data. The default value is 1, indicating that the set will contain singleton data. In some cases, the appropriate value of the dimension can be determined from other option values. Specifically, if the within option is specified, it is used to define the dimension of the data. Also, if the initialize option is specified with a function and the function object has a dimen attribute, then that value defines the dimension of the data. But in general the user is required to specify this option for tuple set data.

In unusual circumstances, the `name` option can be used to specify the set name. In general, this is specified automatically by Pyomo. The `doc` option can be used to document the role of a set in a model.

If the `virtual` option is `True`, then the set does not contain elements. Instead, the validation functions of these sets are typically used by concrete sets to validate membership. When a virtual set is used to specify the value of the `within` option, then the virtual set's validation function is used to check the elements that are added to that set. Predefined virtual sets are shown in Table 4.2.

Ordered sets may have first and last values. The `bounds` option can be used to specify a 2-tuple that defines upper and lower bounds for a set.

4.1.5 RangeSet

The `RangeSet` class is an ordered set that represents a sequence of integer or floating point values. This sequence is defined by a start value, a final value, and a step size. If a `RangeSet` object is defined with a single argument, then that value defines the final value. The start value defaults to 1 and the step size defaults to 1. For example, the following defines a sequence of integers from 1 to 10:

```
model.A = RangeSet(10)
```

In general, the final value represents the maximum allowable value in the sequence. However, the final value is only in the range set if it can be expressed as an integer multiple of the step size plus the start value. For example, the following defines a sequence of integers from 1 to 10:

```
model.B = RangeSet(10.5)
```

Here, the final value is a non-integer, so the maximum value in the sequence is the first integer preceding it.

If a `RangeSet` object is defined with two arguments, then the first is the start value and the second is the final value. For example, the following defines a sequence of integers from 5 to 10:

```
model.C = RangeSet(5,10)
```

Finally, if a `RangeSet` object is defined with three arguments, then they are the start value, final value and step size respectively. For example, the following defines a sequence of floating point values from 2.5 to 10.0 with step 1.5:

```
model.D = RangeSet(2.5,11,1.5)
```

4.1.6 Discussion of Index Sets

Sets can be used to index a wide variety of Pyomo components, including parameters, variables, objectives, and constraints. To prepare for subsequent sections, we review the use of sets as indices. Sets can have multi-dimensional indices, and they also can have multi-dimensional members. The dimension of the members is determined by the `dimen` attribute, while the dimension of the indices into a set array are determined by the dimension of the data in the sets that are given as index arguments.

There are a variety of ways to create and use multi-dimensional sets. Here are a few:

```
model.K = model.A * model.B
model.K2 = Set(dimen=2)

model.DFirst = Set(model.A, model.B)
model.DSecond = Set(model.K)
model.D2 = Set(model.K2)
```

In this example, we know that the members of set `model.K2` have dimension two and therefore `model.D2` is a two-dimensional array of sets. If both `model.A` and `model.B` have one-dimensional elements, then `model.K` has two-dimensional elements and the `model.DFirst` and `model.DSecond` are both two-dimensional arrays of sets. Higher dimensional set members, and therefore higher dimensional index sets, can be obtained by using the `dimen` set parameter, assigning higher dimensional data to sets, or by cross products of higher dimensional sets.

Often, a modeler would like to have an index set that is *sparse* in some sense. This enables creation of an object that is sparse, by which we mean that it is not a full cross product of sets. For example, a set that is a subset of model.A * model.B might be used as an index set in a situation where not all values of `model.B` are needed for all values of `model.A`. This is accomplished by declaring a set whose members are restricted to be `within=model.A * model.B` and whose actual members are defined using an `initialize` argument as described in Section 4.1.2 or by using methods described in Chapter 6 or Chapter 10.

Consider an example where there are three floors in a building to be modeled with rooms of interest on each floor. To provide indices for parameters and variables, a modeler may wish to form an index set with a set array as follows:

```
def FloorRoom_init(model, i):
   if i==1:
      return ['Lecture Hall']
   elif i==2:
      return ['Conference Room', 'Coffe Room']
   elif i==3:
      return range(301, 313)
model.FloorRoom = Set([1,2,3], initialize=FloorRoom_init)
```

doc	A string describing this parameter
domain	A set that is used to validate parameter values
name	The parameter name
nochecking	If true, various checks regarding valid domain and index values are skipped. This is intended for Pyomo developers
repn_type	Gives the name of the underlying representation
validate	A function that a user can specify to validate new parameter data

Table 4.3 Attributes of Param objects.

Alternatively, a two dimensional set might be desired instead. One way to construct such as set is via an initialize function:

```
def Floor_Room_init(model):
   retval = [(1, 'Lecture Hall'), (2, 'Conference Room')]
   retval = retval + [(2, 'Coffee Room')]
   retval = retval + [(3, i) for i in range(301, 313)]
   return retval
model.Floor_Room = Set(dimen=2, initialize=Floor_Room_init)
```

4.2 Parameter Data

A parameter is a numerical or symbolic value that is used to formulate constraints and objectives in a model. Pyomo parameters are managed with the Param class, which can denote a single, independent value, or an array of values. The following sections describe the syntax for parameter declarations, initialization and parameter validation. Table 4.3 describes the attributes of a Param object.

4.2.1 Param Declarations

A simple declaration of Param declares a single value:

```
model.Z = Param()
```

By default, a Param object can have any value (string, floating point, etc). The within option can be used to specify a set that defines the space of valid parameter values. This set can be a predefined Pyomo virtual set, as well as a set that is defined within the model. For example:

```
model.A = Set(initialize=[1,2,3])
model.Y = Param(within=model.A)
model.X = Param(within=Reals)
model.W = Param(within=Boolean)
```

Parameter Y has a value in set A, parameter X is a real value, and parameter W is a boolean value (i.e. True or False).

A parameter array can also be specified by providing sets as options to the `Param` object:

```
model.B = Set()
model.U = Param(model.B)
model.C = Set()
model.T = Param(model.A, model.C)
```

The declaration for parameter `T` illustrates that a multi-dimensional parameter array is declared by simply including a list of sets as options to the `Param` object.

The `Param` class may use different internal representations that have different trade-offs between memory usage and the expressiveness of the data structures. The underlying representation used by a `Param` object is controlled by the `repn` option. The `pyomo_dict` representation is the standard internal representation for a parameter; this representation is the default so it is obtained without specifying the option.

In concrete models, the `pyomo_dict` representation is fairly efficient. However, in abstract models this representation offers error checking and other protections at the expense of computer time and memory. An alternative is the `sparse_dict`, which is uses less memory but provides less error checking when parameters values are changed from their initial values.

4.2.2 Parameter Initialization

By default, a parameter object refers to an abstract parameter (or parameter array) in a model. The `initialize` option can be used to specify the value of a parameter:

```
model.Z = Param(initialize=32)
```

The value of the `initialize` option is simply a valid value for this parameter. Note that the `rule` option is a synonym for the `initialize` option.

Pyomo assumes that parameter values are specified with a sparse representation. For example, the `Param` object `T` declares a parameter indexed over sets `A` and `B`:

```
model.T = Param(model.A, model.B)
```

However, not all of these values are necessarily defined in a model. For example:

```
model.B = Set(initialize=[1,2,3])
w={}
w[1] = 10
w[3] = 30
model.W = Param(model.B, initialize=w)
```

Parameter `W` is defined for indices 1 and 3, but the index set `B` includes 1, 2, and 3. If `W[2]` is accessed, an error occurs and a Python exception is thrown.

The `default` option can be used to specify parameter values when the parameter is not initialized. For singletons, this option is equivalent to simply using the `initialize` option. For parameter arrays, this provides a value for all valid

indices that have not been explicitly initialized. For example, we can define a parameter array that represents a diagonal matrix as follows:

```
u={}
u[1,1] = 10
u[2,2] = 20
u[3,3] = 30
model.U = Param(model.A, model.A, initialize=u, default=0)
```

Note that the parameter data is stored with a sparse representation, even if the default value is specified. This option simply avoids generating an exception and instead provides a value for unspecified indices.

The `initialize` option can also specify a function that returns the value of each parameter. For a singleton parameter, this function accepts a model object and it returns a single value:

```
def Y_init(model):
   return 2.0
model.Y = Param(initialize=Y_init)
```

The `initialize` option can also specify a function that is used to provide elements for a parameter array. The function accepts the model and array indices, and it returns the parameter value for that array index:

```
def X_init(model, i, j):
   return i*j
model.X = Param(model.A, model.A, initialize=X_init)
```

If ordered sets are used define the index for a parameter array, then a recursive initialization function can be employed to define parameter values:

```
def XX_init(model, i,j):
   if i==1 or j==1:
      return i*j
   return i*j+model.XX[i-1,j-1].value
model.XX = Param(model.A, model.A, initialize=XX_init)
```

4.2.3 Data Validation

Validation of parameter data is supported in two ways. First, the domain of feasible parameter values can be specified using the `within` option:

```
model.Z = Param(within=Reals)
```

The default domain for parameters is `Any`, so by default no domain validation is performed. Validation of parameter data can also be performed with the `validate` option, which specifies a function that returns `True` if a parameter value is valid. The following example uses the `validate` option to mimic the behavior of the `within` option:

```
def Y_validate(model, value):
    return value in Reals
model.Y = Param(validate=Y_validate)
```

Validation of parameter arrays is performed similarly. The `within` option can be specified, which applies to all values for all parameter indices. The `validate` option specifies a function whose arguments are the parameter value, the parameter indices and the model:

```
model.A = Set(initialize=[1,2,3])
def X_validate(model, value, i):
    return value > i
model.X = Param(model.A, validate=X_validate)
```

Finally, note that if both the `within` and `validate` options are specified, then the logic for both of these options are applied to validate parameter values.

4.3 Discussion

Set Expressions

Section 4.1.1 describes set operations that can be used to define a set from other set data. Unfortunately, Pyomo does not currently have general support for expressions that contain set operations. Consequently, these set operations are limited to use in concrete models, where you can assume that the set data is available before the set operator is executed.

Set and Parameter Initialization

The `Set` and `Param` classes can be initialized with a variety of different standard Python data types. However, the `initialize` parameter for these classes allows modelers to perform initialization from arbitrary data types. For example, initializa-

tion of parameters from numpy arrays is a simple matter of defining an initialization rule that returns the value of the numpy array:

```
vec = numpy.array([1,2,4,8])
model.C = Set(initialize=range(len(vec)))
def V_rule(model, i):
   return vec[i]
model.V = Param(model.C, initialize=V_rule)
```

Chapter 5
Miscellaneous Model Components and Utility Functions

Abstract A Pyomo model is populated with modeling components that define model data, and with model structures like objectives and constraints. This chapter describes the Pyomo classes for modeling components that supplement the core components described in the previous two chapters. This chapter also includes a description of component indexing, including the specification of dynamic index sets. Finally, this chapter describes utility functions that simplify the formulation of Pyomo models.

5.1 Miscellaneous Components

Several Pyomo components that support the development of abstract models, but which are not used to define a part of the model itself. The `BuildAction` component is used to inject arbitrary actions into the model construction process. Similarly, the `BuildCheck` component is used to perform a test during the model construction process. These components are added to a model in the same manner as other components, but their role is to allow a user to insert diagnostics into the model construction process.

The `BuildAction` and `BuildCheck` components can be used in both concrete and abstract models, but they are most interesting for abstract models. Consider Examples 5.A.1 and 5.A.2. Example 5.A.1 is a Python script that defines a function that constructs a concrete model. Because this model is concrete, each component is initialized as it is constructed. Thus, we can naturally perform error checks and print diagnostic output while the model is being constructed.

In Example 5.A.1, the function creates a simple network flow model for a clique of size `N`. An error check is performed to confirm that the edge set contains the expected number of elements and the script prints out the edge weights in sorted order. The function returns the constructed model and, consequently, this function provides the same type of encapsulation of model construction that is supported by an abstract model.

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_5,

Example 5.A.2 is a corresponding abstract model for this network flow problem. In this model, the parameter N is not given a value in the model defintion file; its value must be supplied in a data file. The setup of the components in this abstract model is otherwise quite similar to the component definitions in the concrete model. The main difference is that in the abstract model the component rules reference the parameter N, while the rules in the concrete model use the Python variable N that is passed into the create function.

Example 5.A.2 illustrates the use of the BuildAction and BuildCheck components. These components define actions that are executed during the construction of the abstract model. These actions provide the same functionality as the error checking and print diagnostics in Example 5.A.1, but use rules that are not executed immediately when they are constructed. Instead, these rules are executed while creating the model instance from the abstract model. The BuildAction component executes the associated rule. The BuildCheck component executes its rule and wil terminate the construction of the model instance if the rule returns False.

As with other components, the BuildAction and BuildCheck components can be indexed, which allows actions and checks to be customized based on specific data. For example, the BuildAction component in Example 5.A.2 can be replaced with the following component declaration:

```
def action_rule(m, i, j):
   # A debugging statement
   print i,j, value(m.w[i,j])
model.action = BuildAction(model.edges, rule=action_rule)
```

This action prints an edge weight for each element of the edge set.

5.2 Advanced Component Indexing

A key aspect of component indexing that was not discussed in previous chapters concerns how Pyomo handles multi-dimensional index sets. The core data structures in Pyomo allow for multi-dimensional indexing of components in a uniform manner. Regardless of the dimensionality of the index sets, components are indexed with Python tuples that are a representation representation of the indexing sets.

Consider the following simple example:

```
model.A = Set(initialize=[1,2,3])
model.B = Set(initialize=[(1,1),(2,2),(3,3)])

model.z = Param(model.B, initialize=1.0)
model.y = Param(model.A, model.B, initialize=1.0)
```

In this example, parameter z is indexed by set B, which contains 2-tuples. Consequently, parameter z is a two-dimensional parameter that is indexed by 2-tuples. Similarly, parameter y is indexed by sets A and B. Since A contains singleton elements, the total dimension of the indexing sets for y is three. Thus, parameter y is

a three-dimensional parameter. Internally, the elements from A and B are combined to form 3-tuples, which are used to index the y parameter.

This process of flattening the indexing set has an important consequence: temporary sets can be used to define component indexes. For example, consider the definition of parameter x:

```
model.x = Param(['a','b'], model.B, initialize=1.0)
```

As with parameter y, parameter x is a three-dimensional parameter. Although the declaration of x relies on a Python list, this data is integrated with the B set data to form an independent index for x. Note that this creates a temporary set within the Pyomo model; the generation of large temporary sets may slow down the model generation.

Another strategy for specifying a temporary set is to define an indexing rule:

```
def C_index(model):
   for k in model.A:
      if k % 2 == 0:
         yield k
model.w = Param(C_index, model.B, initialize=1.0)
```

An indexing rule returns a set, iterator, or generator that represents a set index; the function C_index is a generator in this example. This function is passed as an argument to a component constructor in the same way as any other indexing set, and the indexing rule is called when the component is initialized.

Indexing rules can also define multi-dimensional set data:

```
def D_index(model):
   for i in model.B:
      k = sum(i)
      if k % 2 == 0:
         yield (k,k)
D_index.dimen=2
model.v = Param(D_index, model.B, initialize=1.0)
```

Note that this requires specifying the dimen attribute for the indexing rule in order to set up the proper dimensionality of the component index *before* the component is initialized.

5.3 Functions to Support Modeling

Pyomo includes a variety of functions that support the construction and evaluation of model expressions. Functions that are used to construct nonlinear expressions are described in Chapter 8 (see Table 8.2 on page 107). Table 5.1 summarizes the utility functions that support the concise expression of modeling concepts. These functions are described further in this section.

`display`	Display the properties of models and model components
`dot_product`	Compute a generalized dot product
`sequence`	Compute a sequence of integers
`summation`	Compute a generalized dot product
`value`	Compute the value of a model component
`xsequence`	Compute a sequence of integers

Table 5.1 Pyomo utility functions that support the construction and evaluation of model expressions.

5.3.1 Generalized Dot Products

The `summation` function is a utility function that computes a generalized dot product; the `dot_product` is a synonym for this function. This function creates an expression that represents the sum of elements of one or more indexed components. We use the following components in our examples:

```
model.N = Set(initialize=[1,2,3])
model.M = Set(initialize=[1,3])

model.a = Param(model.N, initialize={1:1, 2:3.1, 3:4.5})

model.x = Var(model.N, within=NonNegativeReals)
model.y = Var(model.N, within=NonNegativeReals)
model.z = Var(model.M, within=NonNegativeReals)
```

In the simplest case, `summation` creates an expression that represents the sum of the elements of an indexed component. For example,

```
summation(model.x)
```

represents the sum $\sum_{i=1}^{3} x_i$. This function provides a convenient shorthand for defining expressions in objectives and constraints. For example, the following constraint uses the Python `sum` function and a Python generator expression to define the constraint body:

```
model.c1 = Constraint(expr=
             sum(model.x[i] for i in model.N) <= 0)
```

This constraint can be rewritten in a more concise format with the `summation` function:

```
model.c2 = Constraint(expr=summation(model.x) <= 0)
```

More generally, the `summation` function can be used to create an expression for dot products between two arrays of parameters and variables in addition to generalized dot products and sums of subexpressions. For example, the following objective computes a simple dot product between `a` and `x`:

```
model.o1 = Objective(expr=summation(model.a, model.x))
```

The `denom` option is used to create terms that are fractions. This option specifies one or more components that are divided into each term of the sum:

```
model.o2 = Objective(expr=summation(model.x, denom=model.y))
```

The objective in this expression represents the sum $\sum_{i=1}^{3} x_i/y_i$. Similarly, two or more terms can be included in the denominator with the following syntax:

```
model.o3 = Objective(expr=
            summation(model.x, denom=(model.a, model.y)))
```

The previous examples have constructed sums from components with the same index set. When components have different index sets, the index set is inferred from the component arguments. If one or more components are specified for the numerator, then the last component specified defines the index for the sum. For example, consider the following:

```
model.o4 = Objective(expr=
            summation(model.x, model.z, denom=model.a))
```

This sum represents the polynomial $x_1 z_1/a_1 + x_3 z_3/a_3$. The index for component `z` defines the index for this sum, since it occurs last. Similarly, if no numerator components are specified, then the last component specified with the `denom` option defines the index for the sum. For example, consider the following:

```
model.o5 = Objective(expr=
            summation(denom=(model.x, model.z)))
```

This sum represents the polynomial $\frac{1}{x_1 z_1} + \frac{1}{x_3 z_3}$.

Finally, the `index` option allows the explicit specification of an index set. For example:

```
model.o6 = Objective(expr=
            summation(model.x, model.y, index=model.M))
```

This sum represents $x_1 y_1 + x_3 y_3$.

5.3.2 Generating Sequences

The function `sequence([start,] stop[, step])` returns a list that is an arithmetic progression of integers. With a single argument, `sequence` returns a sequence that starts with `1`. Thus, `sequence(i)` returns `[1, 2, ..., i]`. With two arguments, `sequence(i, j)` returns `[i, i+1, i+2, ..., j]`. The third argument, when given, specifies the increment (or decrement if negative).

Note that `sequence` is simply a wrapper around the Python `range` function. The main difference is that the lists are shifted by one. The lists returned by `range` start at `0`, and the lists returned by `sequence` start at `1`. Thus, `sequence` has a functionality that is more familiar to mathematical modelers.

The function `xsequence` returns a Python generator for the list that is created by `sequence`. A generator constructs the numbers in the sequence on demand. For

looping, this is slightly faster and more memory efficient.

5.3.3 Helper Functions

Pyomo includes several helper functions that aid in the interrogation of model objects. Pyomo models and model components support a `display` method, which summarizes the model. The `display` function is a helper function that can be called directly to generate this summary. For example, consider the following simple model:

```
model = ConcreteModel()

model.x = Var(initialize=1.0, bounds=(0,1))

model.y = Var(initialize=3.0, bounds=(2,4))

model.o = Objective(expr=model.x+model.y)
```

The command `display(model)` displays the entire model:

```
Model unknown

  Variables:
    Variable x : Size=1 Domain=Reals
      Value=1.0
    Variable y : Size=1 Domain=Reals
      Value=3.0

  Objectives:
    Objective o : Size=1
      Value=4.0

  Constraints:
    None
```

and the `display(model.x)` command displays the setup of the `x` variable:

```
Variable x : Size=1 Domain=Reals
  Value=1.0
```

Various components of a Pyomo model have a *value* that represents The `value` function provides a wrapper for accessing and computing the value of Pyomo components. For example, `value(model.x)` returns the value of `x` variable:

```
1.0
```

and `value(model.o)` compute the value of the expression for the `o` objective:

```
4.0
```

5.4 Discussion

A variety of modeling components are defined in Pyomo. The `pyomo` command can be used to list the available modeling components:

```
pyomo --help-components
```

Specifically, the `AbstractModel` and `ConcreteModel` *objects* are also modeling *components*. These are subclasses of the `Block` and `Model` components, which represent more fundamental classes that are used to organize the components in a Pyomo model. Although Pyomo allows these components to be added to a Pyomo model, the result of this behavior is not well-defined. Instead, these components are supported for advanced modeling packages in Coopr, such as `coopr.gdp`, which leverage these components to explicitly manage different blocks of components.

5.A Examples

5.A.1 Error Checks in a Concrete Model

A Python script that illustrates the use of error checks and diagnostic output in the construction of a concrete model.

```
def create(N):
   #
   # Create a maxflow problem on a random graph
   #
   model = ConcreteModel()

   def edges_rule(m):
      return [(i,j) for i in sequence(N)
                    for j in sequence(N)
                    if i != j]
   model.edges = Set(dimen=2, rule=edges_rule)

   if len(model.edges) != N*(N-1):
      raise RuntimeError, "Check failed"

   def w_rule(m,i,j):
      return random.randint(1,10)
   model.w = Param(model.edges, initialize=w_rule)

   print "Edge weights"
   for e in sorted(model.edges):
      print e, value(model.w[e])

   model.x = Var(model.edges, within=NonNegativeReals)

   def obj_rule(m):
      return sum(m.x[i,N] for i in sequence(N)
```

```
                    if (i,N) in m.edges)
   model.obj = Objective(sense=maximize)

   def flow_rule(m, i):
      return sum(m.x[j,i] for j in sequence(N-1)
                    if (j,i) in m.edges) == \
             sum(m.x[i,j] for j in sequence(2,N)
                    if (i,j) in m.edges)
   model.flow = Constraint(RangeSet(2,N-1))

   def limit_rule(m, i, j):
      return m.x[i,j] <= m.w[i,j]
   model.limit = Constraint(model.edges)

   return model

model = create(4)
```

5.A.2 Error Checks in an Abstract Model

A Python script that illustrates the use of `BuildAction` and `BuildCheck` components to define error checks and diagnostic output in an abstract model.

```
#
# Create a maxflow problem on a random graph
#
model = AbstractModel()

model.N = Param(within=Integers)

def edges_rule(m):
   return [(i,j) for i in sequence(m.N)
              for j in sequence(m.N)
              if i != j]
model.edges = Set(dimen=2, rule=edges_rule)

def check_rule(m):
   # An error check
   return len(m.edges) == m.N*(m.N-1)
model.check = BuildCheck(rule=check_rule)

def w_rule(m,i,j):
   return random.randint(1,10)
model.w = Param(model.edges, initialize=w_rule)

def action_rule(m):
   # A debugging statement
   print "Edge weights"
   for e in sorted(m.edges):
      print e, value(m.w[e])
```

```
model.action = BuildAction(rule=action_rule)

model.x = Var(model.edges, within=NonNegativeReals)

def obj_rule(m):
   N = value(m.N)
   return sum(m.x[i,N] for i in sequence(m.N)
                  if (i,N) in m.edges)
model.obj = Objective(sense=maximize)

def flow_rule(m, i):
   return sum(m.x[j,i] for j in sequence(m.N-1)
                  if (j,i) in m.edges) == \
          sum(m.x[i,j] for j in sequence(2,m.N)
                  if (i,j) in m.edges)
model.flow = Constraint(RangeSet(2,model.N-1))

def limit_rule(m, i, j):
   return m.x[i,j] <= m.w[i,j]
model.limit = Constraint(model.edges)
```

Chapter 6
Initializing Abstract Models with Data Command Files

Abstract Data command files allow users to define set and parameter data that can be used to construct a model instance from an abstract Pyomo model. This chapter discusses the format of these data commands, as well as the various data declarations that Pyomo supports. Pyomo's data commands include both direct specifications of data, as well as specifications that indicate how data is to be extracted from a variety of different sources: ASCII table files, CSV files, spreadsheets, and databases.

6.1 Model Data

The `Set` and `Param` components of a Pyomo model define the data that is used to construct constraints and objectives. By default, the `Set` and `Param` components define abstract data declarations. For example, the following declarations specify no data for set `A` and parameter `p`:

```
model.A = Set(within=Reals)
model.p = Param(model.A, within=Integers)
```

Note that a `Set` or `Param` component is abstract under several conditions:

- A `Set` component is abstract if the `initialize` option is not specified.
- A `Param` component is abstract if both the `initialize` and `default` options are not specified.
- A `Set` or `Param` component is abstract if it is indexed by an abstract `Set` component.

There are two ways to initialize the abstract data declarations in a Pyomo model: (1) with a *data command file* and (2) with a `ModelData` object. This chapter focuses on the use of data command files, whose syntax closely resembles the syntax of AMPL's data commands [4]. The use of a `ModelData` object is illustrated in Chapter 10.

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_6,

A data command file consists of a sequence of commands that directly specify set and parameter data, or specify where such data is to be obtained from external sources. Several commands can be used to declare data:

- The `set` command declares set data.
- The `param` command declares a table of parameter data, which can also include the declaration of the set data used to index parameter data.
- The `import` command defines how set and parameter data is imported from external data sources, including ASCII table files, CSV files, ranges in spreadsheets, and database tables.

The following commands can also be used in data command files:

- The `include` command specifies a data command file that is to be processed immediately.
- The `data` and `end` commands do not perform any actions, but they provide compatibility with AMPL scripts that define data commands.

The `namespace` declaration allows data commands to be organized into named groups that can be enabled or disabled during model construction.

Note that Pyomo's data commands do *not* exactly correspond to AMPL data commands. The `set` and `param` commands are designed to exactly match AMPL's syntax and semantics. Nevertheless, it is not possible to support other AMPL commands because Pyomo treats these commands as data declarations while AMPL treats these commands as part of its scripting language. For example, the syntax of the AMPL `table` command allows the user to specify complex mappings from table data values to corresponding model parameters and sets. The corresponding Pyomo `import` command supports much simpler mappings. Complex mappings are accomplished in Pyomo via scripting.

The remaining sections of this chapter describe the syntax associated with the different Pyomo data file commands. The syntax of data commands can be quite varied, and the goal of this chapter is to provide detailed examples that illustrate these commands. Note that all Pyomo data commands are terminated with a semicolon, and the syntax of data commands does not depend on whitespace. Thus, data commands can be broken across multiple lines, newlines and tab characters are ignored, and they can be formatted with whitespace with few restrictions.

6.2 The `set` Command

6.2.1 Simple Sets

The `set` data command explicitly specifies the members of either a single set or an array of sets, i.e., an indexed set. A single set is specified with a list of data values that are included in this set. The formal syntax for the set data command is given as follows:

```
set <setname> := [<value>] ... ;
```

The data values in a set consist of either numeric values, simple strings or quoted strings:

- *Numeric values* are any string that can be evaluated by Python as a numeric value, e.g., integer, float, scientific notation, or boolean.
- *Simple strings* are sequences of alpha-numeric characters.
- *Quoted strings* are simple strings that are included in a pair of single or double quotes. A quoted string can include quotes within the quoted string.

There is no restriction on the values in a set declaration. A set may be empty, and it may contain any combination of numeric and non-numeric string values. Validation of set data is performed when constructing a Pyomo model, not while parsing a data command file. For example, the following are valid set commands:

```
# An empty set
set A := ;

# A set of numbers
set A := 1 2 3;

# A set of strings
set B := north south east west;

# A set of mixed types
set C :=
0
-1.0e+10
'foo bar'
infinity
"100"
;
```

Note that numeric values are automatically converted to Python integer or floating point values when the set data specification is parsed. A quoted string can be used to define a string value that contains a numeric value. However, if the string strictly specifies a numeric value, it will be converted by Python to a numeric type. For example, the string “100” is included in set C, but this value is converted to a numeric value.

6.2.2 Sets of Tuple Data

The `set` data command can also specify tuple data with the standard notation for tuples. For example, suppose that the set `A` contains 3-tuples:

```
model.A = Set(dimen=3)
```

The following `set` data command then specifies that `A` is the set containing the tuples (1,2,3) and (4,5,6):

```
set A := (1,2,3) (4,5,6) ;
```

Alternatively, set data can simply be listed in the order that the tuple is represented:

```
set A := 1 2 3 4 5 6 ;
```

Obviously, the number of data elements specified using this syntax should be a multiple of the set dimension.

Sets with 2-tuple data can also be specified in a matrix denoting set membership. For example, the following `set` data command declares 2-tuples in `A` using + to denote valid tuples and − to denote invalid tuples:

```
set A : A1 A2 A3 A4 :=
   1 + - - +
   2 + - + -
   3 - + - - ;
```

This data command declares the following four 2-tuples: ('A1',1), ('A1',2), ('A2',3), ('A3',2), ('A4',1).

Finally, a set of tuple data can be concisely represented with tuple *templates* that represent a *slice* of tuple data. For example, suppose that the set `A` contains 4-tuples:

```
model.A = Set(dimen=4)
```

The following `set` data command declares groups of tuples that are defined by a template and data to complete this template:

```
set A :=
   (1,2,*,4) A B
   (*,2,*,4) A B C D ;
```

A tuple template consists of a tuple that contains one or more $\star$ symbols instead of a value. These represent indices where the tuple value is replaced by the values from the list of values that follows the tuple template. In this example, the following tuples are in set `A`:

```
(1, 2, 'A', 4)
(1, 2, 'B', 4)
('A', 2, 'B', 4)
('C', 2, 'D', 4)
```

6.2.3 Set Arrays

The set data command can also be used to declare data for a set array. Each set in a set array must be declared with a separate set data command with the following syntax:

```
set <set-name>[<index>] := [<value>] ... ;
```

Because set arrays can be indexed by an arbitrary set, the index value may be a numeric value, a non-numeric string value, or a comma-separated list of string values.

Suppose that a set A is used to index a set B, as follows:

```
model.A = Set()
model.B = Set(model.A)
```

Then set B is indexed using the values declared for set A:

```
set A := 1 aaa 'a b';

set B[1] := 0 1 2;
set B[aaa] := aa bb cc;
set B['a b'] := 'aa bb cc';
```

6.3 The param Command

Simple or non-indexed parameters are declared in an obvious way, as shown by these examples:

```
param A := 1.4;
param B := 1;
param C := abc;
param D := true;
param E := 1.0e+04;
```

Most parameter data is indexed over one or more sets, and there are a number of ways the param data command can be used to specify indexed parameter data.

Parameters can be defined with numeric and string data. Numeric data is defined with a string that can be evaluated by Python as a numeric value, which includes integer, floating point, scientific notation, and boolean. Boolean values can be specified with a variety of strings: TRUE, true, True, FALSE, false, and False. Note that parameters cannot be defined without data, so there is no analog to the specification of an empty set.

6.3.1 One-dimensional Parameter Data

One-dimensional parameter data is indexed over a single set. Suppose that the parameter `B` is a parameter indexed by the set `A`:

```
model.A = Set()
model.B = Param(model.A)
```

A `param` data command can specify values for `B` with a list of index-value pairs:

```
set A := a c e;

param B := a 10 c 30 e 50;
```

Because whitespace is ignored, this example data command file can be reorganized to specify the same data in a tabular format:

```
set A := a c e;

param B :=
a 10
c 30
e 50
;
```

Multiple parameters can be defined using a single `param` data command. For example, suppose that parameters `B`, `C`, and `D` are one-dimensional parameters all indexed by the set `A`:

```
model.A = Set()
model.B = Param(model.A)
model.C = Param(model.A)
model.D = Param(model.A)
```

Values for these parameters can be specified using a single `param` data command that declares these parameter names followed by a list of index and parameter values:

```
set A := a c e;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The values in the `param` data command are interpreted as a list of sublists, where each sublist consists of an index followed by the corresponding numeric value.

Note that parameter values do not need to be defined for all indices. For example, the following data command file is valid:

```
set A := a c e g;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The index g is omitted from the param command, and consequently this index is not valid for the model instance that uses this data. More complex patterns of missing data can be specified using the "." character to indicate a missing value. This syntax is useful when specifying multiple parameters that do not necessarily have the same index values:

```
set A := a c e;

param : B C D :=
a . -1 1.1
c 30 . 3.3
e 50 -5 .
;
```

This example provides a concise representation of parameters that share a common index set that indexes each parameter differently.

Note that this data file specifies the data for set A twice: (1) when A is defined and (2) implicitly when the parameters are defined. An alternate syntax for param allows the user to concisely specify the definition of an index set along with associated parameters:

```
param : A : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

Finally, we note that default values for missing data can also be specified using the default keyword:

```
set A := a c e;

param B default 0.0 :=
c 30
e 50
;
```

Note that default values can only be specified in param commands that define values for a single parameter.

6.3.2 Multi-Dimensional Parameter Data

Multi-dimensional parameter data is indexed over either multiple sets or multi-dimensional sets. Suppose that parameter `B` is a parameter indexed by set `A` that has dimension 2:

```
model.A = Set(dimen=2)
model.B = Param(model.A)
```

The syntax of the `param` data command remains essentially the same when specifying values for `B` with a list of index and parameter values:

```
set A := a 1 c 2 e 3;

param B :=
a 1 10
c 2 30
e 3 50;
```

Missing and default values are also handled in the same way with multi-dimensional index sets:

```
set A := a 1 c 2 e 3;

param B default 0 :=
a 1 10
c 2 .
e 3 50;
```

Similarly, multiple parameters can defined with a single `param` data command. Suppose that parameters `B`, `C`, and `D` are one-dimensional parameters indexed over set `A` that has dimension 2:

```
model.A = Set(dimen=2)
model.B = Param(model.A)
model.C = Param(model.A)
model.D = Param(model.A)
```

These parameters can be defined with a single `param` command that declares these parameter names followed by a list of index and parameter values:

```
set A := a 1 c 2 e 3;

param : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```

The following param data command defines the index set along with the parameters:

```
param : A : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```

The param command also supports a matrix syntax for specifying the values in a parameter that has a 2-dimensional index. Suppose parameter B is indexed over set A that has dimension 2:

```
model.A = Set(dimen=2)
model.B = Param(model.A)
```

The following param command defines a matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B : a c e :=
1 1 2 3
2 4 5 6
3 7 8 9
;
```

Additionally, the following syntax can be used to specify a transposed matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B (tr) : 1 2 3 :=
a 1 4 7
c 2 5 8
e 3 6 9
;
```

This functionality facilitates the presentation of parameter data in a natural format. In particular, the transpose syntax may allow the specification of tables for which the rows comfortably fit within a single line. However, a matrix may be divided column-wise into shorter rows since the line breaks are not significant in Pyomo's data commands.

For parameters with three or more indices, the parameter data values must be specified as a series of slices. Each slice is defined by a template followed by a list of index and parameter values. Suppose that parameter B is indexed over set A that has dimension 4:

```
model.A = Set(dimen=4)
model.B = Param(model.A)
```

The following `param` command defines a matrix of parameter values with multiple templates:

```
set A := (a,1,a,1) (a,2,a,2) (b,1,b,1) (b,2,b,2);

param B :=

  [*,1,*,1] a a 10 b b 20
  [*,2,*,2] a a 30 b b 40
;
```

The `B` parameter consists of four values: `B[a,1,a,1]=10`, `B[b,1,b,1]=20`, `B[a,2,a,2]=30`, and `B[b,2,b,2]=40`.

6.4 The `import` Command

The `import` command provides a mechanism for loading data from an external data source such as a relational database. This command loads a *relational table* that represents set and parameter data in a Pyomo model. A relational table consists of a matrix of numeric string values, simple strings, and quoted strings. All rows have the same lengths, all columns have the same length, and the first row represents labels for the column data.

The `import` command can load data from a variety of different external data sources, including CSV files, ASCII table files, spreadsheets, and relational databases. This command uses a *data manager* that coordinates how data is extracted from a specified *data source*. In this way, the `import` command provides a generic mechanism that enables Pyomo models to interact with standard data repositories that are maintained in an application-specific manner.

In the following section, we illustrate the syntax of the `import` command when used to load data from ASCII table files. Next, we discuss the command syntax that is used to import data from ASCII table and CSV files; these file formats are very similar, so we discuss them together. In Section 6.4.4 we provide corresponding examples for spreadsheets and relational databases. This section also describes advanced features that are specific to databases, including the specification of SQL queries to collect data.

NOTE: ASCII table files and CSV files can both be used to represent a relational table, and the difference between ASCII table and CSV files is their formatting. Elements in an ASCII table are separated by space and tab characters. The format of each row is not sensitive to the use of spaces and tab characters, but each row is terminated with a newline. The following is an example of a ASCII table file with four columns and three rows:

```
A B C D
Data1 Data2 1.0 1e-2
'Data 3' 'Data 4' 2.0 1e+2
100 FOO 3.0 0
```

Elements in a CSV file are separated by commas, and each row is terminate with a newline. The following is an example of a CSV file that corresponds to the previous ASCII table file:

```
A,B,C,D
Data1,Data2,1.0,1e-2
Data 3,Data 4,2.0,1e+2
100,FOO,3.0,0
```

6.4.1 Simple Import Examples

The simplest illustration of the `import` command is specifying data for an indexed parameter. Consider the file `Y.tab`:

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

This file specifies the values of 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 filename. The options after the colon indicate how the table data is mapped to model data. Option `[A]` indicates that set `A` is used as the index, and option `Y` indicates the parameter that is initialized.

Similarly, the following import command loads both the parameter data as well as the index set `A`:

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

The difference is the specification of the index set, `A=[A]`, which indicates that set `A` is initialized with the index loaded from the ASCII table file.

Set data can also be loaded from a ASCII table file that contains a single column of data:

```
A
A1
A2
A3
```

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

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

Note that this allows for specifying set data that contains tuples. Consider file `C.tab`:

```
A B
A1 1
A1 2
A1 3
A2 1
A2 2
A2 3
A3 1
A3 2
A3 3
```

A similar `import` syntax will load this data into set `C`:

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

Note that this example requires that `C` be declared with dimension two.

6.4.2 Import Syntax Options

The syntax of the `import` command is broken into two parts. The first part ends with the colon, and it begins with a filename, database URL, or DSN (data source name). Additionally, this first part can contain option value pairs. The following options are recognized:

`format`	A string that denotes how the relational table is interpreted
`password`	The password that is used to access the data source (for databases)
`query`	The query that is used to request data from a database
`range`	The subset of a spreadsheet that is requested
`user`	The user name that is used to access the data source
`using`	The data manager that is used to process the data source (for databases)

The `format` option is the only option that is required for all data managers. This option specifies how a relational table is interpretted to represent set and parameter data. A complete set of examples for this option is provided in Section 6.4.3. If the `using` option is omitted, then the filename suffix is used to select the data manager. The remaining options are specific to spreadsheets and relational databases, and these are discussed further in Section 6.4.4.

The second part of the import command consists of the specification of column names for indices and data. The remainder of this section describes different specifications and how they define how data is loaded into a model. Suppose file ABCD.tab defines the following relational table:

```
A B C D
A1 B1 1 10
A2 B2 2 20
A3 B3 3 30
```

There are many ways to interpret this relational table. It could specify a set of 4-tuples, a parameter indexed by 3-tuples, two parameters indexed by 2-tuples, and so on. Additionally, we may wish to select a subset of this table to initialize data in a model. Consequently, the import command provides a variety of syntax options for specifying how a table is interpreted.

A simple specification is to interpret the relational table as a set:

```
import ABCD.tab format=set : Z ;
```

Note that Z is a set in the model that the data is being loaded into. If this set does not exist, an error will occur while loading data from this table.

Another simple specification is to interpret the relational table as a parameter with indexed by 3-tuples:

```
import ABCD.tab : [A,B,C] D ;
```

Again, this requires that D be a parameter in the model that the data is being loaded into. Additionally, the index set for D must contain the indices that are specified in the table. The import command also allows for the specification of the index set:

```
import ABCD.tab : Z=[A,B,C] D ;
```

This specifies that the index set is loaded into the Z set in the model. Similarly, data can be loaded into another parameter than what is specified in the relational table:

```
import ABCD.tab : Z=[A,B,C] Y=D ;
```

This specifies that the index set is loaded into the Z set and that the data in the D column in the table is loaded into the Y parameter.

This syntax allows the import command to provide an arbitrary specification of data mappings from columns in a relational table into index sets and parameters. For example, suppose that a model is defined with set Z and parameters Y and W:

```
model.Z = Set()
model.Y = Param(model.Z)
model.W = Param(model.Z)
```

Then the following command defines how these data items are imported using columns B, C and D:

```
import ABCD.tab : Z=[B] Y=D W=C;
```

As noted earlier, when the using option is omitted the data manager is inferred from the filename suffix. However, in some contexts the filename suffix does not

reflect the format of the data it contains. For example, suppose the file `ABCD.txt` contains a relational table in CSV format:

```
A,B,C,D
A1,B1,1,10
A2,B2,2,20
A3,B3,3,30
```

Then we can specify the `using` option to import from this file into parameter `D` and set `Z`:

```
import ABCD.txt using=csv : Z=[A,B,C] D ;
```

The following are valid options for the `using` option:

`csv` Data manager for CSV files
`tab` Data manager for ASCII table files
`xls` Data manager for Excel spreadsheet files

6.4.3 Interpreting Relational Tables

By default, a relational table is interpreted as columns of one or more parameters with associated index columns. The `format` option can be used to specify other interpretations of a table. For example, we have illustrated that the option value `set` is used to interpret a table as a set of values or tuples. The following option values are supported the `format` option:

`array`	The table is a matrix representation of a two dimensional parameter.
`param`	The data is a singleton parameter value.
`set`	Each row is a set element.
`set_array`	The table is a matrix representation of a set of 2-tuples.
`transposed_array`	The table is a transposed matrix representation of a 2-D parameter.

We have previously illustrated the use of the `set` format value to interpret a relational table as a set of values or tuples. The following examples illustrate the other format values.

A relational table with a single value can be interpreted as a singleton parameter using the `param` format value. Suppose that `Z.tab` contains the following "table":

```
B A1 A2 A3
1 + - -
2 - + -
3 - - +
```

The following import command then loads this value into parameter `p`:

```
import Z.tab format=param: p;
```

Sets with 2-tuple data can be represented with a matrix format that denotes set membership. The `set_array` format value interprets a relational table as a matrix that defines a set of 2-tuples where + denotes a valid tuple and − denotes an invalid tuple. Suppose that `D.tab` contains the following relational table:

```
B A1 A2 A3
1 + - -
2 - + -
3 - - +
```

Then the following import command loads data into set B:

```
import D.tab format=set_array: B;
```

This command declares the following 2-tuples: ('A1',1), ('A2',2), and ('A3',3).

Parameters with 2-tuple indices can be interpreted with a matrix format that where rows and columns are different indices. Suppose that U.tab contains the following table:

```
I A1 A2 A3
I1 1.3 2.3 3.3
I2 1.4 2.4 3.4
I3 1.5 2.5 3.5
I4 1.6 2.6 3.6
```

Then the following import command loads this value into parameter U with a 2-dimensional index using the array format value.

```
import U.tab format=array: A=[X] U;
```

The transpose_array format value also interprets the table as a matrix, but it loads the data in a transposed format:

```
import U.tab format=transposed_array: A=[X] U;
```

Note that these format values do not support the initialization of the index data.

6.4.4 Importing from Spreadsheets and Relational Databases

Many of the options for the import command are specific to spreadsheets and relational databases. The range option is used to specify the range of cells that are imported from a spreadsheet. The range of cells represents a relational table; the first row of cells defines the column names for the table.

Suppose that file ABCD.xls contains the range ABCD that is shown in Figure 6.1. The following command imports this data to initialize parameter D and index Z:

```
import ABCD.xls range=ABCD : Z=[A,B,C] Y=D ;
```

Thus, the syntax for importing data from spreadsheets only differs from CSV and ASCII text files by the use of the range option.

When importing from a relational database, the data source specification is a filename or data connection string. Access to a database may be restricted, and thus the specification of username and password options may be required. Alternatively, these options can be specified within a data connection string.

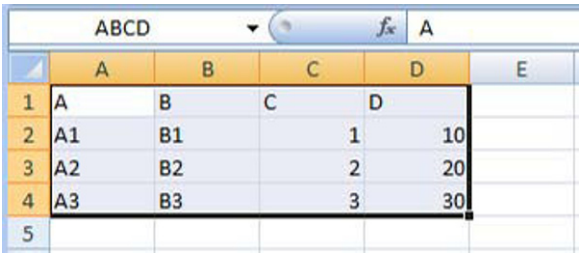

Fig. 6.1 A snapshot of the `ABCD.xls` spreadsheet, which defines a relational table in the `ABCD` range.

A variety of database interface packages are available within Python. The `using` option is used to specify the database interface package that will be used to access the database. The following database interface packages are currently recognized by Pyomo:

pyodbc Provides an ODBC interface to most databases.
pymysql Provides a Python API to a MySQL database.

See Appendix A for directions for installing database interface packages and other third-party software tools that can be leveraged by Pyomo.

The following command imports data from the Excel spreadsheet `ABCD.xls` using the `pyodbc` interface. The command imports this data to initialize parameter `D` and index `Z`:

```
import ABCD.xls using=pyodbc table=ABCD : Z=[A,B,C] Y=D ;
```

The `using` option specifies that the `pyodbc` package will be used to connect with the Excel spreadsheet. The `table` option specifies that the table `ABCD` is loaded from this spreadsheet. Similarly, the following command specifies a data connection string to specify the ODBC driver explicitly:

```
import "Driver={Microsoft Excel Driver (*.xls)}; Dbq=ABCD.xls;"
   using=pyodbc
   table=ABCD : Z=[A,B,C] Y=D ;
```

ODBC drivers are generally tailored to the type of data source that they work with. This syntax illustrates how the `import` command can be tailored to the details of the data source that a user is working with.

The previous examples specified the `table` option, which declares the name of the relational table in the data source. Relational tables can be defined in this manner in both spreadsheets and databases. Some databases also support Structured Query

Language (SQL), which can be used to compose a relational table in a complex manner.

Example 6.A.2 includes a Pyomo model and data files for the classic diet problem. In this example, a customer is faced with the task of minimizing the cost for a meal at a fast food restaurant – they must purchase a sandwich, side, and a drink for the lowest cost. The file diet1.py contains a Pyomo model for this problem, and the file diet1.mdb is a Microsoft Jet (Access) database containing the following data in the Food table:

FOOD	cost
Cheeseburger	1.84
Ham Sandwich	2.19
Hamburger	1.84
Fish Sandwich	1.44
Chicken Sandwich	2.29
Fries	0.77
Sausage Biscuit	1.29
Lowfat Milk	0.60
Orange Juice	0.72

In addition, the Food table has two additional columns, f_min and f_max, with no data for any row. These columns exist to match the structure for the parameters used in the model.

We can solve the diet1 model using the Python definition in diet1.py and the data from this Jet database by embedding a SQL query in an ODBC import line:

```
import "Driver={Microsoft Access Driver (*.mdb)};DBQ=diet.mdb"
   using=pyodbc
   query="SELECT FOOD,cost,f_min,f_max FROM Food":
   [FOOD] cost f_min f_max;
```

The PyODBC driver module will pass the SQL query through an Access ODBC connector, extract the data from the diet1.mdb file, and return it to Pyomo. The Pyomo ODBC handler can then convert the data received into the proper format for solving the model internally. More complex SQL queries are possible, depending on the underlying database and ODBC driver in use. However, the name and ordering of the columns queried are specified in the Pyomo data file; using SQL wildcards (e.g., SELECT *) or column aliasing (e.g., SELECT f AS FOOD) may cause errors in Pyomo's mapping of relational data to parameters.

6.5 The `include` Command

The `include` command allows a data command file to execute data commands from another file. For example, the following command file executes data commands from files `ex1.dat` and then `ex2.dat`:

```
include ex1.dat;
include ex2.dat;
```

Pyomo is sensitive to the order of execution of data commands, since data commands can redefine set and parameter values. The `include` command respects this data ordering; all data commands in the included file are executed before the remaining data commands in the current file are executed.

6.6 Data Namespaces

The `namespace` keyword is not a data command, but instead it is used to structure the specification of Pyomo's data commands. Specifically, a namespace declaration is used to group data commands and to provide a group label. Example 6.A.1 shows data commands for the abstract Pyomo model in Example 2.A.7.1. This data file defines two namespaces: `data1` and `data2`.

By default, data commands contained within a namespace are ignored during model construction. However, model construction can be customized to specify the namespaces that are used during construction. See Section 7.4 for an example of how namespaces are selected with the `pyomo` command.

6.7 Discussion

Support for Pyomo's data command files was initially motivated by a desire to facilitate conversion of AMPL models into Pyomo models. The close correspondence between the `set` and `param` data commands in AMPL and Pyomo allows many AMPL models to be reformulated with Pyomo commands without additional changes to the data specification. Although most other AMPL data commands have not been supported in Pyomo, we have attempted to retain similar functionality. For example, the AMPL `table` command is not supported, but the `import` command supports a subset of its functionality with a simpler syntax.

The syntax for Pyomo's data commands also differ from from AMPL's syntax in order to support additional functionality. For example, namespaces are used to allow the specification of alternate data sets. This construct is particularly useful for defining scenario data for stochastic programming models (e.g., see Chapter 9).

We expect support of data command files to be a core feature of Pyomo in the future. This command language simplifies loading data from relational tables (e.g.,

databases), and in the future we expect that it will be extended to support the exporting of data as well. Although this can be done directly within Python scripts, Pyomo data command files simplify the construction and analysis of abstract models, which is a core feature of Pyomo.

6.A Examples

6.A.1 Namespace Data Commands

Data commands for the abstract Pyomo model in Example 2.A.7.1 that are used to generate the concrete model in Formulation (2.2) This data file illustrates the use of namespaces that define collections of data commands that are grouped together.

```
namespace data1 {
   set N := 1 2 ;

   set M := 1 2 ;

   param c :=
   1 1
   2 2 ;

   param a :=
   1 1 3
   2 1 4
   1 2 2
   2 2 5 ;

   param b :=
   1 1
   2 2 ;
}

namespace data2 {
   set N := 3 4 ;

   set M := 5 6 ;

   param c :=
   3 10
   4 20 ;

   param a :=
   3 5 3
   4 5 4
   3 6 2
   4 6 5 ;

   param b :=
   5 1
   6 2 ;
}
```

6.A.2 The Diet Problem

A Pyomo model for the classic diet problem . The goal of this problem is to minimize cost while ensuring that the diet meets certain requirements.

```
#
# Imports
#

import sys
import os
from os.path import abspath, dirname
sys.path.insert(0, \
    dirname(dirname(dirname(dirname(abspath(__file__))))))
from coopr.pyomo import *

infinity = float('inf')

#
# Model
#

model = AbstractModel()

model.FOOD = Set()

model.cost = Param(model.FOOD, within=PositiveReals)

model.f_min = Param(model.FOOD, within=NonNegativeReals, \
    default=0.0)

MAX_FOOD_SUPPLY = 20.0 # There is a finite food supply
def f_max_validate (model, value, j):
   return model.f_max[j] > model.f_min[j]
model.f_max = Param(model.FOOD, validate=f_max_validate, \
    default=MAX_FOOD_SUPPLY)

# Unneeded vars - they're in the .dat file, so we list them\
     here
model.NUTR = Set()
model.n_min = Param(model.NUTR, within=NonNegativeReals, \
    default=0.0)
model.n_max = Param(model.NUTR, default=infinity)
model.amt = Param(model.NUTR, model.FOOD, \
    within=NonNegativeReals)

# -------------------------------------------------------

def Buy_bounds(model, i):
   return (model.f_min[i], model.f_max[i])
model.Buy = Var(model.FOOD, bounds=Buy_bounds, \
    within=NonNegativeIntegers)
```

```
# ------------------------------------------------------

def Total_Cost_rule(model):
   return sum(model.cost[j] * model.Buy[j] for j in \
       model.FOOD)
model.Total_Cost = Objective(rule=Total_Cost_rule, \
    sense=minimize)

# ------------------------------------------------------

def Entree_rule(model):
   entrees = ['Cheeseburger', 'Ham Sandwich', 'Hamburger', \
       'Fish Sandwich', 'Chicken Sandwich']
   return sum(model.Buy[e] for e in entrees) >= 1
model.Entree = Constraint(rule=Entree_rule)

def Side_rule(model):
   sides = ['Fries', 'Sausage Biscuit']
   return sum(model.Buy[s] for s in sides) >= 1
model.Side = Constraint(rule=Side_rule)

def Drink_rule(model):
   drinks = ['Lowfat Milk', 'Orange Juice']
   return sum(model.Buy[d] for d in drinks) >= 1
model.Drink = Constraint(rule=Drink_rule)
```

A Pyomo data file that defines data for a simple diet problem.

```
param: FOOD: cost f_min f_max :=
  "Cheeseburger" 1.84 . .
  "Ham Sandwich" 2.19 . .
  "Hamburger" 1.84 . .
  "Fish Sandwich" 1.44 . .
  "Chicken Sandwich" 2.29 . .
  "Fries" .77 . .
  "Sausage Biscuit" 1.29 . .
  "Lowfat Milk" .60 . .
  "Orange Juice" .72 . . ;

param: NUTR: n_min n_max :=
      Cal 2000 .
      Carbo 350 375
      Protein 55 .
      VitA 100 .
      VitC 100 .
      Calc 100 .
      Iron 100 . ;

param amt (tr):
                        Cal Carbo Protein VitA VitC Calc \
                            Iron :=
  "Cheeseburger" 510 34 28 15 6 30 20
  "Ham Sandwich" 370 35 24 15 10 20 20
  "Hamburger" 500 42 25 6 2 25 20
```

```
"Fish Sandwich" 370 38 14 2 0 15 10
"Chicken Sandwich" 400 42 31 8 15 15 8
"Fries" 220 26 3 0 15 0 2
"Sausage Biscuit" 345 27 15 4 0 20 15
"Lowfat Milk" 110 12 9 10 4 30 0
"Orange Juice" 80 20 1 2 120 2 2 ;
```

A Pyomo data file that imports data from the Access data base file `diet.mdb`.

```
import "Driver={Microsoft Access Driver (*.mdb)};DBQ=diet.mdb"
   using=pyodbc
   query="SELECT FOOD,cost,f_min,f_max FROM Food":
   [FOOD] cost f_min f_max;
```

Chapter 7
The Pyomo Command-line Interface

Abstract This chapter describes Pyomo's command-line interface, which can be used to construct a model, perform optimization, and summarize the results. Although these steps can be easily executed within a Python script, this command simplifies this process so users can focus on modeling.

7.1 Overview

The Pyomo software distribution includes the `pyomo` command, which was introduced in Section 2.5.1. This command can be used to construct a Pyomo model, convert an abstract model into a concrete instance with user-supplied data, apply an optimizer, and summarize the results. The model construction step requires a Pyomo model file, which is a Python file that defines a Pyomo model object. Thus, the `pyomo` command can be viewed as a generic script for analyzing a model defined by a Pyomo model file.

For example, the following command-line optimizes a model defined in `concrete1.py` using the `glpk` LP solver:

```
pyomo --solver=glpk concrete1.py
```

Similarly, the following command-line optimizes a model defined in `abstract5.py` using data in `abstract5.dat`, also using `glpk`:

```
pyomo --solver=glpk abstract5.py abstract5.dat
```

The `pyomo` command automatically executes the following steps:

1. Construct a model.
2. Read the instance data (if applicable).
3. Generate a model instance (if the model is abstract).
4. Apply simple preprocessors to the model instance.
5. Apply a solver to the model instance.
6. Load the results into the model instance.

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_7,

7. Display the solver results.

The `pyomo` command has a variety of optional command-line arguments that are used to customize the optimization process; documentation of the various available options is available by specifying the `--help` option.

The following sections describe different uses of the `pyomo` command. These examples illustrate the use of the command-line options for `pyomo`, as well as the flexibility of Pyomo's framework for managing models and performing optimization. More advanced uses of options for the `pyomo` command are discussed in Chapter 10.

7.2 Building a Model Instance

The default behavior of the `pyomo` command is to construct a model and then perform optimization. However, while developing a Pyomo model it is often useful to simply construct a model to validate that the modeling logic is correct. The `--instance-only` option directs `pyomo` to simply create a model instance and then terminate. For example, this can be used with the `--verbose` (or `-v`) option to print a summary of the model construction steps:

```
pyomo --instance-only concrete1.py
```

which prints

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.00] Creating model
```

Additionally, the `--save-model` option is used to save the model in a file. The suffix of the filename specifies the file format. For example, the command:

```
pyomo --instance-only --save-model=concrete1.lp concrete1.py
```

creates the file `concrete1.lp` which represents the model from `concrete1.py` in the `LP` file format. Although there is no standard `LP` format, the `LP` files generated by Pyomo are compatible with a variety of linear and integer programing solvers. Similarly, the command:

```
pyomo --instance-only --save-model=concrete1.nl concrete1.py
```

creates the file `concrete1.nl` which represents the model from `concrete1.py` in the `NL` file format. This format is used by solvers compatible with the AMPL modeling software; see Section 7.9 for further discussion of these file formats.

7.3 Specifying the Model Object

A *Pyomo model file* is a Python file that defines a Pyomo model object. Note that a Pyomo model file is not restricted in the type of Python statements that it includes. Within the `pyomo` command, a model file is executed with a Python import command, and thus it is interpreted like any other Python file.

In the simplest case, a Pyomo model file contains Python commands that create a model object that is stored in the `model` variable. For example, consider the simple LP problem described in Equation (2.1) that was introduced in Chapter 2. Example 2.A.7.1 illustrates an abstract Pyomo model for this problem, which is stored in the `model` variable.

If a user defines their model with a different variable name, then the `--model-name` option can be used to direct `pyomo` to select that name. Example 7.A.1 adapts Example 2.A.7.1 to store the model in `Model`, which can be optimized with the following command:

```
pyomo -q --model-name=Model abstract6.py abstract6.dat
```

Aside from supporting greater flexibility for the user, this option allows users to define multiple models in a Pyomo model file and then select the model that is optimized when the `pyomo` command is executed.

7.4 Selecting Data with Namespaces

Section 6.6 introduces the `namespace` command in Pyomo data files. This command is used to define blocks of data commands that are integrated optionally into a model. The `pyomo` command provides the `--namespace` option to specify one or more namespaces that are used to construct an instance of an abstract model; the `--ns` is a shorter alias for this option. For example, the command

```
pyomo --solver=glpk --ns=data1 abstract5.py \
                    abstract5-ns1.dat
```

creates and optimizes the abstract model in Example 2.A.7.1 using the data commands from Example 6.A.1. This command specifies the `data1` namespace, which has an optimal solution of 0.8. Similarly, the command

```
pyomo --solver=glpk --ns=data2 abstract5.py \
                    abstract5-ns1.dat
```

creates and optimizes the same model using the `data2` namespace, which has an optimal solution of 8. A different index set is used in the `data2` data, as well as different objective coefficients.

The previous example illustrates how namespaces allow the user to specify different data sets within a single data command file. Note that a model can be constructed from data commands using multiple namespaces, including data that is not

in a namespace. Consider the data commands in Example 7.A.2, which includes four namespaces and data commands outside of a namespace. The command

```
pyomo --solver=glpk --ns=c1 --ns=data2 abstract5.py \
                  abstract5-ns2.dat
```

creates and optimizers the abstract model in Example 2.A.7.1 using data commands from the `c1` and `data2` namespaces, as well as the data command for `N`, which is outside of any namespace. Note that if multiple namespaces contain data commands for the same component, then the component is initialized with the data from first namespace that contains the corresponding data command. If there is not a namespace containing a corresponding data command, then the data commands outside of namespaces are used to initialize the component.

7.5 Customizing Pyomo's Workflow

The different steps that are executed by the `pyomo` command represent a generic workflow for model construction and optimization. This workflow can be customized using a variety of call-back functions that are defined within a Pyomo model file. These call-back functions allow the user to define additional analysis steps, as well as replace some of the default steps in the workflow.

Table 7.1 summarizes the call-back functions and the functionality that they support. Each call-back function takes one or more keyword arguments in the form `keyword = value`. Consequently, there are two different ways that a call-back function can be defined in Python. Consider the `pyomo_print_results` call-back function, which takes three arguments: `options`, `instance`, and `results`. This function can be defined with default values that are ignored:

```
def pyomo_print_results(options=None, instance=None,
                                      results=None):
   """A call-back with dummy default values"""
   print options
```

Alternatively, Python allows functions to be defined that accept arbitrary arguments and keywords. The keyword arguments are passed in as a dictionary as follows:

```
def pyomo_print_results(**kwds):
   """A call-back with arbitrary keyword arguments"""
   print kwds.get('options',None)
```

In the second example, the dictionary for keyword arguments is used to explicitly reference a function argument. The call-back functions will always pass in their expected arguments, so no additional error checking is required.

There are several standard arguments for the call-back functions described in Table 7.1. The `options` argument is an enhanced Python dictionary that contains the command-line options sent to the `pyomo` command. The `model` argument is the Pyomo model object, and the `instance` argument is the model instance that is constructed from this model. In the case where the user defines a model using

Function	Description
pyomo_preprocess	Perform a preprocessing step before model construction
pyomo_create_model	Construct and return a model object
pyomo_create_modeldata	Construct and return a `ModelData` object
pyomo_print_model	Output model object information
pyomo_modify_instance	Modify the model instance
pyomo_print_instance	Output model instance information
pyomo_save_instance	Save the model instance
pyomo_print_results	Print the optimization results
pyomo_save_results	Save the optimization results
pyomo_postprocess	Perform a postprocessing step after optimization

Table 7.1 Call-back functions that can be used in a Pyomo model file to customize the workflow in the `pyomo` command.

`ConcreteModel`, then the `model` and `instance` arguments are the same object. Other arguments are described with their associated call-back functions.

pyomo_preprocess

This call-back function is executed before model construction to perform preprocessing steps. This function has one argument: `options`. For example, the following call-back function simply prints the command-line options:

```
def pyomo_preprocess(options=None):
   print "The following options were specified:\n%s" % \
                                    options
```

pyomo_create_model

This call-back function is used to construct a model. This function has two arguments: `options` and `model_options`. The latter argument contains the options for constructing the model, which are specified with the `--model-options` command-line option. The return value of this function must be the model object that has been created, which may be either an abstract or concrete model. For example, the following call-back function creates a model by importing the `abstract6.py` file and then returning the `Model` object:

```
def pyomo_create_model(options=None, model_options=None):
   abstract6 = __import__('abstract6')
   return abstract6.Model
```

pyomo_create_modeldata

This call-back function creates a model data object that is used to create a model instance. Model data objects are useful in contexts where a set of different data sources need to be specified for model constructions. This function has two arguments: `options` and `model`. The return value must be a `ModelData` object. For example, the following call-back function creates a `ModelData` object from the file `abstract6.dat`:

```
def pyomo_create_modeldata(options=None, model=None):
   data = ModelData()
   data.add('abstract6.dat')
   data.read(model)
   return data
```

pyomo_print_model

This call-back function prints an abstract model before a model instance is created. This function has two arguments: `options` and `model`. The following example calls the `pprint` method to print detailed information about an abstract model:

```
def pyomo_print_model(options=None, model=None):
   if options.debug:
      model.pprint()
```

pyomo_modify_instance

This call-back function modifies the model instance after it has been constructed. This function has three arguments: `options`, `model`, and `instance`. The following call-back fixes a variable after the model is constructed:

```
def pyomo_modify_instance(options=None, model=None,
                          instance=None):
   instance.x[1].fixed = True
```

pyomo_print_instance

This call-back function prints the Pyomo model instance. This function is used to print the concrete model instance rather than the abstract model. This function has two arguments: `options` and `instance`. The following example calls the `pprint` method to print detailed information about a model instance:

```
def pyomo_print_instance(options=None, instance=None):
   if options.debug:
      instance.pprint()
```

pyomo_save_instance

This call-back function saves the Pyomo model instance. This function has two arguments: `options` and `instance`. Note that Pyomo does not specify how the model is saved. However, a convenient mechanism would be to use Python's pickle mechanism:

```
def pyomo_save_instance(options=None, instance=None):
   OUTPUT = open('abstract7.pyomo','w')
   print >>OUTPUT, pickle.dumps(instance)
   OUTPUT.close()
```

pyomo_print_results

This call-back function prints the results generated from optimization. This function has three arguments: `options`, `instance`, and `results`. The `results` object supports a generic summary of optimization solutions, solver statistics, etc. in both the JSON or YAML formats. Thus, this call-back function can simply print this data:

```
def pyomo_print_results(options=None, instance=None,
                                  results=None):
   print results
```

However, the `pyomo` command includes the `--print-results` option, which performs this operation. More generally, this call-back function is included to allow users to provide problem-specific summaries of their optimization results.

pyomo_save_results

This call-back function is used to save the results generated from optimization. This function has three arguments: `options`, `instance`, and `results`. This call-back function can simply print the results to a file:

```
def pyomo_save_results(options=None, instance=None,
                                  results=None):
   OUTPUT = open('abstract7.results','w')
   print >>OUTPUT, results
   OUTPUT.close()
```

The `pyomo` command includes the `--save-results` option, which performs this operation. More generally, this call-back function is included to allow users to save problem-specific summary of their optimization results.

pyomo_postprocess

This call-back function is executed after optimization to perform postprocessing steps. This function has three arguments: `options`, `instance`, and `results`. For example, the following function prints a simple summary of the optimization results:

```
def pyomo_postprocess(options=None, instance=None,
                                    results=None):
   print "Solution found with value", \
                results.Solution[0].Objective.obj.value
```

7.6 Customizing Solver Behavior

The generic workflow supported by the `pyomo` command includes the execution of a solver to optimize (or otherwise analyze) a model. A variety of command-line options are used to control solver behavior. The `--solver` option is used to specify the name of the solver that is constructed.

The default behavior of the `pyomo` command is to execute the solver, wait for termination, and then collect the results. Remote and asynchronous execution of solvers can also be enabled by selecting an appropriate solver manager with the `--solver-manager` options.

The `--solver` option can specify two classes of solvers: the names of command-line executables that are on the user's path, and predefined solver interfaces. Command-line executables are assumed to perform I/O using NL files. Thus, command-line executables can be optimized with any solver executable that is built with the AMPL solver library.

Table C.2 summarizes the solver interfaces that are recognized by Pyomo. Solver interfaces provide a generic interface to a solver that can be executed with different types of I/O. The `--solver-io` option can be used to select the type of I/O that is used. See Chapter C.2 for further discussion of the trade-offs between different types of I/O. The `--help-solvers` option provides information about the solvers that can be used.

Solver options can be specified in a generic manner using the `--solver-options` option. This specifies a string that is interpreted as one or more option-value pairs. For example, the following option passes the `mipgap` option to the glpk solver:

```
pyomo --solver=glpk --solver-options='mipgap=1' concrete1.py
```

Additionally, the `--timelimit` option can be used to specify the maximum runtime of the solver. This is typically passed into the solver, and thus this timelimit is enforced in a solver-dependent manner.

Solver results are generated from solution information provided by the solver, and optionally a logfile of output from the solver. By default, Pyomo captures in-

formation about the variable values that are selected by the solver. However, there is often additional information that a user may wish to collect, such as dual values for constraints in a linear program. For performance reasons, this data is not automatically collected by the `pyomo` command, but the `--solver-suffixes` option is used to specify the names of the data that is desired. A *suffix* is simply data for a constraint or variable that results from the application of a solver. Suffixes can be specified by name, or with a regular expression. For example, the following command specifies that all suffixes generated by the solver are requested:

```
pyomo --solver=glpk --solver-suffixes='.*' concrete2.py
```

The following suffixes are currently supported within Pyomo:

- `dual` - constraint dual values
- `rc` - reduced costs
- `slack` - constraint slack values

Note that a given solver may provide only a subset of these suffixes.

The `--tempdir` and `--keepfiles` options can be used to archive the temporary files that Pyomo uses. By default, Pyomo uses temporary files that are automatically generated in system temporary directories. The `--tempdir` option is used to specify the directory that these files are created in. By default, temporary files are deleted after optimization is completed. The `--keepfiles` options disables this deletion, which allows the user to see the data that Pyomo sends to the optimizer.

7.7 Analyze Solver Results

The `--postprocess` option can be used to specify a Python module that is executed after the solver has executed. A typical use of this option is to specify post-processing steps that interpret the solver results in a problem-dependent manner.

Post-processing steps can be defined by declaring a `pyomo_postprocess` function in the Python modules that are used in post processing. Figure /ref-fig:command:postprocess provides an example of a post-processing function that writes the final solutions to a file in the CSV format.

7.8 Managing Diagnostic Output

The `pyomo` command includes a variety of options that control the generation of diagnostic output and other information that useful to learn more about the workflow that this is executed.

The default output of the `pyomo` command is a terse summary of the major steps that are executed. The `--log` and `--stream-output` options are used to print the solver output. The `--log` option is used to print the solver output after the solver

```
import csv

def pyomo_postprocess(options=None, instance=None,
                                 results=None):
   #
   # Collect the data
   #
   vars = set()
   data = {}
   f = {}
   for i in range(len(results.solution)):
      data[i] = {}
      for var in results.solution[i].variable:
         vars.add(var)
         data[i][var] = \
            results.solution[i].variable[var]['Value']
      f[i] = results.solution[i].objective['obj']['Value']
   #
   # Write a CSV file, with one row per solution.
   # The first column is the function value, the remaining
   # columns are the values of nonzero variables.
   #
   rows = []
   vars = list(vars)
   vars.sort()
   rows.append(['obj']+vars)
   for i in range(len(results.solution)):
      row = [f[i]]
      for var in vars:
         row.append( data[i].get(var,None) )
      rows.append(row)
   print "Creating results file results.csv"
   writer = csv.writer(open('results.csv', 'wb'))
```

Fig. 7.1 A post-processing plugin that writes final solutions in a CSV file.

has terminated, and the `--stream-output` option is be used to print the solver output as it is generated. Similarly, the `--summary` and `--show-results` options print different summaries of the optimization results. The `--summary` command prints a summary of the Pyomo model, after the results are loaded.

The `--show-results` prints the final results. If the `PyYAML` package is installed, then the default results format is YAML and the final results are stored in the file `results.yml`. Otherwise, the default results format is JSON and the final results are stored in the file `results.json`. The `--jsonpyomo command@pyomo command!argument, --json` option can be used to specify the JSON results format when the `PyYAML` package is installed. The `--save-results` option can be used to specify an alternative results file.

Pyomo uses a standard Python logging system to manage the printing of logging messages for the underlying software in Coopr and PyUtilib. By default, logging

messages that represent Coopr errors and warnings are always printed, and all PyUtilib logging messages are suppressed. The `--quiet` option suppresses all log messages except for those that refer to errors. The `--warning` option enables warning messages for both Coopr and PyUtilib. The `--info` option enables informative, warning and error log messages for Coopr and PyUtilib.

The `--verbose` option enables debugging log messages for Coopr and PyUtilib. This option can be specified multiple times to enable logging messages for different parts of Coopr and PyUtilib: (1) debugging for just Pyomo, (2) debugging for all Coopr packages, and (3) debugging for all Coopr and PyUtilib packages. The `--debug` option enables debugging logging, and it allows exceptions to trigger a failure in which the program stack is printed.

7.9 Discussion

The `pyomo` command applies many optimizers by generating a temporary file in a standard problem format that is understood by the optimizer. Currently, two different problem formats can be generated from Pyomo models: the `NL` format that is recognized by solvers used with the AMPL modeling tool, and the `LP` file format that is used by a variety of commercial and open source integer programming solvers. Taken together, these two problem formats can express a wide variety of linear and nonlinear optimization problems.

It is often useful to generate these problem files, both to diagnose issues with a model as well as to directly manage the execution of a solver. Pyomo includes two additional commands that can be used to generate these problem files: `pyomo2nl` and `pyomo2lp`. These commands share many options with the `pyomo` command, but they simply generate the problem file and terminate.

7.A Examples

7.A.1 Model Object with Non-Default Name

An abstract Pyomo model for Formulation (2.1) that is stored in the `Model` variable.

```
from coopr.pyomo import *

Model = AbstractModel()

Model.N = Set()
Model.M = Set()
Model.c = Param(Model.N)
Model.a = Param(Model.N, Model.M)
Model.b = Param(Model.M)

Model.x = Var(Model.N, within=NonNegativeReals)

def obj_rule(Model):
   return sum(Model.c[i]*Model.x[i] for i in Model.N)
Model.obj = Objective(rule=obj_rule)

def con_rule(Model, m):
   return sum(Model.a[i,m]*Model.x[i] for i in Model.N) \
               >= Model.b[m]
Model.con = Constraint(Model.M, rule=con_rule)
```

7.A.2 Pyomo Data Commands with Multiple Namespaces

Data commands for the abstract Pyomo model in Example 2.A.7.1 that are used to generate the concrete model in Formulation (2.2) This data file illustrates the use of namespaces that define collections of data commands that are grouped together.

```
set N := 1 2;

namespace c1 {
   param c :=
   1 1
   2 2 ;
}

namespace c2 {
   param c :=
   1 10
   2 20 ;
}

namespace data1 {
   set M := 1 2 ;

   param a :=
   1 1 3
   2 1 4
   1 2 2
   2 2 5 ;

   param b :=
   1 1
   2 2 ;
}

namespace data2 {
   set M := 5 6 ;

   param a :=
   1 5 3
   2 5 4
   1 6 2
   2 6 5 ;

   param b :=
   5 1
   6 2 ;
}
```

Chapter 8
Nonlinear Programming with Pyomo

Abstract The modeling components that are described earlier in the book can be used within nonlinear models. This chapter describes the nonlinear programming capabilities of Pyomo. It presents all the nonlinear expressions and functions that are supported, and it provides some tips for formulating and solving nonlinear programming problems. Pyomo makes use of the interface provided by the AMPL Solver Library to provide efficient expression evaluation and automatic differentiation. Use of the AMPL Solver Library means that any AMPL-enabled solver should be usable as a solver within the Pyomo framework. This chapter also provides several real-world examples to illustrate formulating and solving nonlinear programming problems.

8.1 Introduction

Several powerful tools exist for solving linear programming problems. Whenever possible, one should strive to formulate continuous problems as an LP. Nevertheless, there are times when it is not possible to adequately represent the true problem without of nonlinear expressions. Fortunately, Pyomo has the ability to represent general nonlinear programming problems.

Pyomo supports straightforward modeling of NLP problems; however, the solution of this class of problems presents several challenges that do not exist for LP problems. For example, most modern, efficient NLP solvers require derivatives of the constraints and the objective function. Since the functions are nonlinear, this requires accurate numerical evaluation of these derivatives at a given trial point. Furthermore, in the case of non-convex problems, multiple local minima may exist (due to the shape of the objective function or the constraints), and specifying a suitable starting point may be critical.

In Section 8.2, we present the supported nonlinear expressions and illustrate how to build a basic nonlinear problem formulation within Pyomo using the Rosenbrock function as an example. In Section 8.3, we discuss the solver interface that allows for

W.E. Hart et al., *Pyomo—Optimization Modeling in Python,*
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_8,

any AMPL-enabled solver to be used with Pyomo. We also give a few tips to help with effectively formulate nonlinear programming problems. Finally, we close this chapter with a number of small, but real-world nonlinear programming examples.

8.2 Building Nonlinear Programming Formulations

Pyomo supports the following general nonlinear programming formulation:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & d^L \leq d(x) \leq d^U \\ & x^L \leq x \leq x^U. \end{aligned}$$

The allowable form of the objective function $f(x)$, the vector of equality constraints $c(x)$, and the vector of inequality constraints $d(x)$ depends entirely on the solver that is selected to provide a solution. However, Pyomo has been tested with local and global solvers that typically assume that these functions are continuous and smooth, with continuous first (and possibly second) derivatives. The development of nonlinear extensions for Pyomo has focused on this broad problem class.

8.2.1 Nonlinear Expressions

Formulating nonlinear optimization problems in Pyomo is no different from formulating linear or mixed-integer examples like those previously discussed in this book. All the Pyomo modeling components that we have described throughout the book are used in the same way (e.g., `Var`, `Constraint`) except that they may include nonlinear expressions.

Table 8.1 lists the operators that are currently supported with examples where `x` and `y` are Pyomo `Var` objects. In addition to these operators, Pyomo supports a number of nonlinear functions as described in Table 8.2. Unfortunately, not every mathematical function that is available in Python is available to write nonlinear programming problems in Pyomo. Behind the scenes, Pyomo creates a nonlinear expression tree for each nonlinear expression in the objectives and constraints.

8.2.2 The Rosenbrock Example

This section will present a short example to describe how to create and solve a nonlinear Pyomo model using a formulation for the unconstrained minimization of the

Operation	Operator	Example
multiplication	`*`	`expr = model.x * model.y`
division	`/`	`expr = model.x / model.y`
exponentiation	`**`	`expr = (model.x+2.0)**model.y`
in-place multiplication[1]	`*=`	`expr *= model.x`
in-place division[2]	`/=`	`expr /= model.x`
in-place exponentiation[3]	`**=`	`expr **= model.x`

[1] The example given for in-place multiplication is equivalent to `expr = expr * model.x`.
[2] The example given for in-place multiplication is equivalent to `expr = expr / model.x`.
[3] The example given for in-place multiplication is equivalent to `expr = expr ** model.x`.

Table 8.1 Python operators that have been redefined to generate Pyomo expressions.

Operation	Function	Example
arccosine	`acos`	`expr = acos(model.x)`
hyperbolic arccosine	`acosh`	`expr = acosh(model.x)`
arcsine	`asin`	`expr = asin(model.x)`
hyperbolic arcsine	`asinh`	`expr = asinh(model.x)`
arctangent	`atan`	`expr = atan(model.x)`
hyperbolic arctangent	`atanh`	`expr = atanh(model.x)`
cosine	`cos`	`expr = cos(model.x)`
hyperbolic cosine	`cosh`	`expr = cosh(model.x)`
exponential	`exp`	`expr = exp(model.x)`
natural log	`log`	`expr = log(model.x)`
log base 10	`log10`	`expr = log10(model.x)`
sine	`sin`	`expr = sin(model.x)`
square root	`sqrt`	`expr = sqrt(model.x)`
hyperbolic sine	`sinh`	`expr = sinh(model.x)`
tangent	`tan`	`expr = tan(model.x)`
hyperbolic tangent	`tanh`	`expr = tanh(model.x)`

Table 8.2 Functions supported by Pyomo for the definition of nonlinear expressions.

two-variable Rosenbrock function. The Rosenbrock function is shown in Figure 8.1. This is a classic problem that is frequently used as an example for discussion of unconstrained nonlinear optimization algorithms (see, for example, [44]). The minimization problem is defined as,

$$\min_{x,y} \; f(x,y) = (1-x)^2 + 100\left(y-x^2\right)^2,$$

and the solution is in the bottom of the banana shaped valley at the point x=1 and y=1. Example 8.A.1 provides a Pyomo model for this problem.

The Pyomo model in Example 8.A.1 begins like any other model; first the necessary packages are imported, and then a model object is created.

```
from coopr.pyomo import *

model = AbstractModel()
```

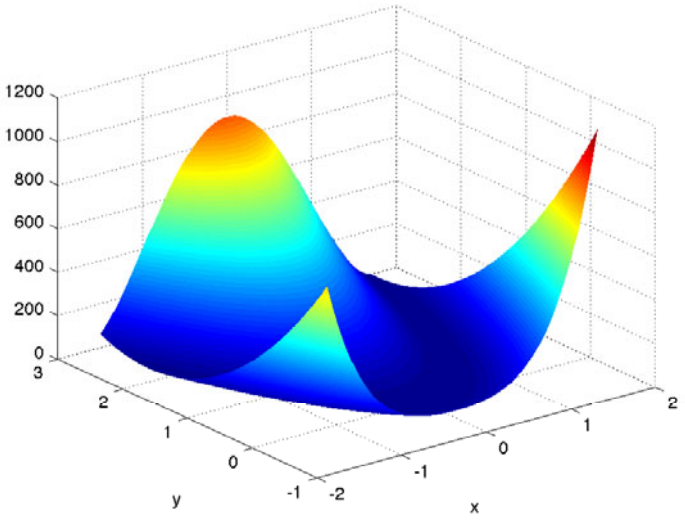

Fig. 8.1 Contours of the Rosenbrock function $f(x,y)=(1-x)^2+100\left(y-x^2\right)^2$. The minimum is in the bottom of a banana shaped valley at the point x=1, y=1.

Next, the model creates two variables `x` and `y` and initializes each of them to a value of 1.5.

```
model.x = Var(initialize = 1.5)
model.y = Var(initialize = 1.5)
```

Notice that this declaration is identical to that used in linear examples. There is no need to provide any indication that the variable will later appear in a nonlinear expression. This will be deduced by Pyomo before solving the problem.

With the two variables defined, it is now straightforward to define the rule for the objective function and add it to the model.

```
def rosenbrock(model):
    return (1.0-model.x)**2 \
        + 100.0*(model.y - model.x**2)**2
model.obj = Objective(rule=rosenbrock, sense=minimize)
```

Again, it is clear that defining the nonlinear model is really no different from defining linear models. One can make use of a variety of nonlinear expressions, as we will see in the next section. Also notice that the python operator for an exponent is `**`, rather than `^`.

We can solve this nonlinear optimization problem with the following `pyomo` command. Here, we have assumed that the solver IPOPT is available on the path:

```
pyomo --solver=ipopt --summary Rosenbrock.py
```

This will produce the following output:

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.00] Creating model
[ 0.00] Applying solver
[ 0.02] Processing results
   Number of solutions: 1
   Solution Information
     Gap: <undefined>
     Status: optimal
   Solver results file: results.yml

==========================================================
Solution Summary
==========================================================

Model unknown

  Variables:
    Variable x : Size=1 Domain=Reals
      Value=1.0
    Variable y : Size=1 Domain=Reals
      Value=1.0

  Objectives:
    Objective obj : Size=1
      Value=7.01364595134e-25

  Constraints:
    None

[ 0.02] Applying Pyomo postprocessing actions
[ 0.02] Pyomo Finished
```

In this output, we see that the problem is correctly solved to a value of $x=y=1.0$, with an objective value of essentially zero. While this example only has a single nonlinear objective and two scalar variables, all the modeling components that were discussed in earlier chapters are available. As we have just seen, formulating a nonlinear programming problem in Pyomo is as easy as formulating linear or mixed-integer problems.

8.3 Solving Nonlinear Programming Formulations

Pyomo enforces a clear separation between the modeling language used to formulate the problem and the numerical package that is used to find a solution. As is the case

NOTE: The `pyomo` command may result in the following output:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
[    0.01] Applying solver
ERROR: Unexpected exception while running model Rosenbrock.py
        Problem constructing solver 'ipopt'
```

This occurs when the nonlinear solver (in this case `ipopt`) is not available on the path. See Section 8.3 for more information.

with the linear and mixed-integer formulations, a user must first obtain and install an appropriate nonlinear solver before Pyomo can be used with nonlinear programming formulations.

8.3.1 Nonlinear Solvers

Nonlinear programming solvers require the modeling framework to evaluate the nonlinear objective and constraints at candidate points in x. However, most efficient nonlinear solvers also require evaluation of first (and often second) derivatives at candidate points as well. Nevertheless, it is often problematic to require users to provide expressions for the first and second derivatives for objectives and constraints. This transformation is both tedious and error prone. Because of this, many modeling languages and solvers have interfaced with tools for automatic differentiation (AD). AD tools can be used to provide accurate and efficient numerical evaluation of the first and second derivatives without any user involvement.

Pyomo currently relies on the solver interface provided by the AMPL Solver Library (ASL)[26] for solving NLP problems. This has several benefits. First, there are a wealth of existing nonlinear programming solvers that have already been interfaced with AMPL through the ASL. By supporting all the ASL solvers, these packages are already available for use with Pyomo without any need to develop another interface. Second, the ASL provides efficient numerical evaluation of first and second derivative information.

Pyomo generates an `.nl` file that describes a nonlinear problem (complete with expression trees for nonlinear objectives and constraints). The format of `.nl` files is described by Gay [27]. These files are then read by the solver packages through the interface provided in the ASL. The ASL has methods that allow the solver to evaluate the objective, constraints, and derivative information as needed. Through the ASL interface, solvers also write the solution values to a `.sol` file that is read by Pyomo to obtain the solution. Further details concerning these file formats are provided in Gay et al. [23, 26].

NOTE: Because Pyomo uses the ASL interface, *Pyomo should work with any AMPL-based solver*. Thus a number of competitive, commercial and open-source packages can be used to solve Pyomo models. A user must install an ASL solver and make sure that solver is available on the path before running the `pyomo` command. In all the examples in this chapter, we have made use of IPOPT[37], an open-source package available from the COIN-OR foundation. The instructions for obtaining and installing the IPOPT solver are available at the COIN-OR website: http://www.coin-or.org/Ipopt/.

8.3.2 Tips for Nonlinear Programming

Effective formulation and solution of nonlinear programming problems can be significantly more challenging than linear programming problems. In this section, we provide a few basic tips to help with the transition from linear programming to nonlinear programming.

Variable Initialization

Probably the most important difference when solving nonlinear programming problems is the need for effective initialization of the problem variables. You will notice that all the examples in this chapter specify initial values for the variables. If initial values are not specified, then Pyomo assumes that the initial values are zero. This can be problematic because of domain violations, as discussed below. However, effective initialization is important for another reason.

For the general nonconvex case, nonlinear programming problems can, and often do, have multiple local solutions. While academic and commercial solvers do exist that are mathematically guaranteed to find the global solution of a nonlinear programming problem, the current state-of-the-art for these packages limits the size of the problem that can be addressed. Consequently, one is often forced to look at packages that provide a guarantee of local optimality only. For these packages, it is often critical to initialize the problem near the desired local solution.

Sometimes, the undesired local solutions are not physically meaningful, and a sensible initialization with reasonable variable bounds is sufficient to ensure reliable progress to the desired solution. Other times, there may be several physically reasonable local solutions. The development of good nonlinear problem formulations often includes significant effort to provide a reasonable initialization strategy. Furthermore, while this is not a requirement of most solvers, some solvers can benefit from initialization at a point that is already feasible for the constraints.

Undefined Evaluations

Several nonlinear functions that are only well-defined over a specific domain (e.g., $log(x)$ is only valid for $x > 0$). Therefore, the modeler must take care to ensure that the problem formulation restricts the variable values to be within this domain. This is usually accomplished by setting reasonable bounds on the variables.

It is also important to note that many nonlinear solver packages require first (and sometimes second) derivative information for the objective function and the constraints. Therefore, one must restrict the variables to be within a valid domain of the nonlinear expressions and the derivatives of these expressions. For example, a common occurrence is to include `sqrt(x)` in an expression, along with the bounds $x \geq 0$. While $\sqrt{x}$ is valid at x=0, its derivative, $1/\sqrt{x}$ is not. This must be considered when setting reasonable variable bounds.

Finally, note that some nonlinear interior-point solvers (e.g., IPOPT) may relax the variable bounds slightly before solving the problem. While this has proven to be an effective strategy in most cases, this can sometimes cause a domain violation even if the modeler has specified reasonable variable bounds. One may need to disable this behavior in the solver, or apply more conservative bounds.

Model Singularities and Problem Scaling

Many nonlinear programming solvers have restrictions on the constraints (called constraint qualifications) that must be satisfied. In particular, it is often a good idea to ensure that the constraints are independent everywhere within the solution domain (i.e., the set of active constraint gradients are linearly independent). This further means that none of the gradients of the active constraints should be zero. Nocedal and Wright [44] discuss this issue further (see Chapter 12).

Unfortunately, a model that satisfies these restrictions in exact math may still exhibit problems when solved numerically. If the model is ill-conditioned, then many solvers can have difficulty converging or finding a solution efficiently. It is important to scale the model as much as possible to provide a well-conditioned Jacobian and Hessian. This can be as simple as linearly scaling the variables and the constraints. However, in difficult cases, the model may need to be reformulated.

8.4 Nonlinear Programming Examples

8.4.1 Variable Initialization in Minimization of Multimodal Function

In this simple example, we illustrate the importance of effective variable initialization. We will be minimizing the following multimodal function,

$$f(x) = (2 - \cos \pi x - \cos \pi y)\, x^2 y^2.$$

Example 8.A.2 shows a Pyomo script for this problem, which initializes the variables at x=y=0.25. If we solve this problem using the following `pyomo` command,

```
pyomo --solver=ipopt --summary multimodal_init1.py
```

we can see that IPOPT finds the solution that is close to our initial point x=$y = 0.0178$. However, if we change the problem and initialize the variables at x=y=2.0,

```
model.x = Var(initialize = 2.0, bounds=(0,4))
model.y = Var(initialize = 2.0, bounds=(0,4))
```

and rerun the problem, we can now see that IPOPT finds a different local solution at x=y=2.0.

8.4.2 Optimal Quotas for Sustainable Harvesting of Deer

Maintaining a healthy deer population relies on both effective habitat development and a sustainable harvesting policy. Among most hunters there is high demand for tags that allow them to take bucks. However, over harvesting of bucks within a population can be devastating to future population growth. The primary goal of this nonlinear programming formulation is to determine an optimal policy for deer harvesting that maximizes the value of the harvest while maintaining a strong and sustainable deer population.

We begin this example by presenting the model that describes the dynamics of the deer population. This example was adapted from Bailey [7]. The deer population in a given area can be divided into three sub-populations: bucks, does, and fawns. Aditionally, each year is divided into four periods: winter, breeding season, summer, and harvest. The model describing the population dynamics is based on the following assumptions:

- It is assumed that the sub-populations can be represented by continuous variables (i.e., population numbers are large enough that this is a good approximation).
- Each season, there is a reduction in the number of bucks, does, and fawns. This reduction is assumed to be due to natural causes and is proportional to the size of the sub-populations. This reduction is captured by specifying a fractional survival rate that depends on the period (winter, breeding, summer, harvest) and the sub-population in question (bucks, does, fawns).
- New fawns are born each year during the breeding season. Fawns are born from does and older fawns according to a birth rate that depends on the available amount of food. Half of them are assumed to be male and half are assumed to be female. After surviving one year, half of the remaining fawns become bucks and half become does.
- The total yearly food supply is constant and represents a constraint based on habitat management.
- All harvesting is based on hunting. Hunting quotas can be set for each sub-population, and these quotas are assumed to be completely filled (i.e., all hunters are successful).

The complete derivation of the sub-population model is given in [7], resulting in the following set of difference equations,

$$f_{y+1} = p_1 br_y \left(\frac{p_2}{10} f_y + p_3 d_y\right) + h_y^f \tag{8.1}$$

$$d_{y+1} = p_4 d_y + \frac{p_5}{2} f_y - h_y^d \tag{8.2}$$

$$b_{y+1} = p_6 b_y + \frac{p_5}{2} f_y - h_y^b \tag{8.3}$$

$$br_y = 1.1 + 0.8 \frac{p_s - c_y}{p_s} \tag{8.4}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y \tag{8.5}$$

where the value for parameters p_1 through p_9 are calculated from the various survival rates and food consumption rates. These values are given in Table 8.3. The variables f_y, d_y, and b_y represent the number of fawns, does, and bucks in year y, respectively. Likewise, h_y^f, h_y^d, and h_y^b are the unknown numbers of fawns, does, and bucks harvested in year y, respectively. The birth rate br_y for does is described by a nonlinear relationship where c_y is the amount of food consumed by the deer (in pounds) and p_s is the total available supply of food (again in pounds).

parameter	value	parameter	value
p_1	0.88	p_7	2700.0
p_2	0.82	p_8	2300.0
p_3	0.92	p_9	540.0
p_4	0.84	w_f	1.0
p_5	0.73	w_d	1.0
p_6	0.87	w_b	10.0
p_s	700 000		

Table 8.3 Parameter values used the the deer harvesting problem.

In the original reference, this set of difference equations was optimized in the formulation over a period of 20 years so that a sustainable steady-state policy could be deduced from the values at later years. Here, we instead include only one year and add the constraint that the number of fawns, does, and bucks at year $y+1$ is equal to those at y. This provides a steady-state solution with a formulation that is significantly smaller.

The objective is to maximize the value of the harvest, giving the following nonlinear programming formulation,

$$\max \; w_b h_y^b + w_f h_y^f + w_d h_y^d \tag{8.6}$$

$$f_y = p_1 br_y \left(\frac{p_2}{10} f_y + p_3 d_y \right) + h_y^f \tag{8.7}$$

$$d_y = p_4 d_y + \frac{p_5}{2} f_y - h_y^d \tag{8.8}$$

$$b_y = p_6 b_y + \frac{p_5}{2} f_y - h_y^b \tag{8.9}$$

$$br_y = 1.1 + 0.8 \frac{p_s - c_y}{p_s} \tag{8.10}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y \tag{8.11}$$

$$c_y \leq p_s \tag{8.12}$$

$$b_y \geq \frac{1}{5} (0.4 f_y + d_y) \tag{8.13}$$

where w_f, w_d and w_b represent the value of harvesting a fawn, doe, and buck, respectively. As can be seen in Table 8.3, it is assumed that the value of a buck tag is 10 times the value of a doe or fawn tag. Equation (8.12) ensures that the amount of consumed food cannot be more than the available supply, thereby restricting the overall size of the population. Equation (8.13) ensures that the number of bucks is large enough for effective, sustainable breeding.

Example 8.A.3 provides a Pyomo model for the optimal deer harvesting problem and a data file for the parameters in Table 8.3. The following command line can be used to optimizes this problem:

```
#!/bin/sh

pyomo --solver=ipopt --summary DeerProblem.py DeerProblem.dat
```

This produces the following output:

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.05] Creating model
[ 0.09] Applying solver
[ 0.17] Processing results
   Number of solutions: 1
   Solution Information
     Gap: <undefined>
     Status: unknown
   Solver results file: results.yml

============================================================
Solution Summary
============================================================

Model unknown

  Variables:
    Variable f : Size=1 Domain=PositiveReals
      Value=189.605592667
    Variable d : Size=1 Domain=PositiveReals
```

```
    Value=196.006401042
  Variable b : Size=1 Domain=PositiveReals
    Value=54.3697276124
  Variable hf : Size=1 Domain=PositiveReals
    Value=0.0
  Variable hd : Size=1 Domain=PositiveReals
    Value=37.8450171568
  Variable hb : Size=1 Domain=PositiveReals
    Value=62.1379767339
  Variable br : Size=1 Domain=PositiveReals
    Value=1.09999999201
  Variable c : Size=1 Domain=PositiveReals
    Value=700000.00699

 Objectives:
  Objective obj : Size=1
    Value=659.224784496

 Constraints:
  Constraint f_bal : Size=1
    Value=8.98748453437e-9
  Constraint d_bal : Size=1
    Value=7.1054273576e-15
  Constraint b_bal : Size=1
    Value=0.0
  Constraint food_cons : Size=1
    Value=-7.27595761418e-11
  Constraint supply : Size=1
    Value=700000.00699
  Constraint birth : Size=1
    Value=-2.00017780116e-12
  Constraint minbuck : Size=1
    Value=9.34788602081e-9

[ 0.17] Applying Pyomo postprocessing actions
[ 0.17] Pyomo Finished
```

In the solution summary, we can see that the quotas favor harvesting of bucks, but harvesting too many bucks would affect population growth. We can also see that the residual for the `minbuck` constraint is essentially zero (meaning that this constraint is active). Therefore, this constraint restricts the number of bucks that can be harvested. The optimal quota policy also allows for some harvesting of does, but no harvesting of fawns.

Obviously, this solution is a function of the parameter values that determine the value of fawns, does, and bucks in the objective function, as well as the parameters in model for the population dynamics. Because Pyomo is built with Python, it is straightforward to develop a script that determines the optimal solution as a function of different parameter values, enabling more advanced analysis of the system. Chapter 10 gives more discussion of this functionality and provides numerous examples.

8.4.3 Estimation of Parameters in Infectious Disease Models

Effective widespread vaccination programs in the modern era have significantly minimized the impact of many childhood diseases. However, childhood infectious diseases continue to be a concern in developing countries, and outbreaks of new disease strains pose challenges for public health policy makers. In this example, we simulate the outbreak of an infectious disease within a small community of 300 individuals (representing, for example, a small school). We derive a basic model to describe the spread of infection in the population and use a nonlinear programming formulation to estimate key parameters in this model using the simulated data.

We use a standard discrete time compartment model to represent the system. Individuals are separated into three compartments based on their status with respect to the disease: susceptible (S), infected (I), or recovered (R). We assume that once an individual has contracted the disease and recovered, they are immune from that point forward (i.e., they do not return to the susceptible pool). The discrete time model representing this systems is given by:

$$I_i = \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N}$$
$$S_i = S_{i-1} - I_i$$

These two difference equations describe the propagation of the disease in the population. As a generation-based model, it is assumed that all the individuals infected at time i have recovered by time $i+1$. I_i and S_i are the number of infected and susceptible individuals at time i, respectively. The population size is given by N, and β and α are model parameters.

In this example, we use least-squares to estimate the parameters from simulated data. The full problem formulation is given by,

$$\begin{aligned}
\min \sum_{i \in SI} \left(\varepsilon_i^I\right)^2 & \\
I_i &= \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N} \ \ \forall \ i \in SI \setminus 1 \\
S_i &= S_{i-1} - I_i \ \ \forall \ i \ \in SI \setminus 1 \\
C_i &= I_i + \varepsilon_i^I \\
0 &\leq I_i, \ Si \leq N \\
0.5 &\leq \beta \leq 70 \\
0.5 &\leq \alpha \leq 1.5
\end{aligned}$$

where SI is the set of indices for the serial intervals. In our example, we are estimating over one year, comprising 26 two-week serial intervals. The reported cases (known input) are given by C_i, and the variable ε_i^I is the residual between the measured and calculated cases.

Example 8.A.4 provides an abstract Pyomo model for this least-squares estima-

tion problem, along with a Pyomo data file that contains the data for an instance of this model. We can optimize this model with the following command line:

```
#!/bin/sh

pyomo --solver=ipopt --summary DiseaseEstimation.py DiseaseEstimation.dat
```

However, this command line generates a lot of information, which makes it difficult to find the optimized values for β and α.

As discussed in Chapter 7, the optimization process supported by the `pyomo` command can be customized with call-back functions that print the variables of interest from the results data. Consider the following call-back function:

```
def pyomo_postprocess(options=None, instance=None, \
                      results=None):
   print ' ***'
   print ' *** Optimal beta Value: ', instance.beta.value
   print ' *** Optimal alpha Value: ', instance.alpha.value
   print ' ***'
```

If this is added to the model file `DiseaseEstimation.py`, then the `pyomo` script will call this call-back function after solving the problem. If we execute the `pyomo` command without the `--summary` flag:

```
#!/bin/sh

pyomo --solver=ipopt DiseaseEstimationCallback.py DiseaseEstimation.dat
```

then the call-back produces a more manageable output that clearly shows the variables of interest:

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.00] Creating model
[ 0.05] Applying solver
[ 0.15] Processing results
   Number of solutions: 1
   Solution Information
     Gap: <undefined>
     Status: unknown
   Solver results file: results.yml
[ 0.17] Applying Pyomo postprocessing actions
***
*** Optimal beta Value: 1.98603420504
*** Optimal alpha Value: 1.00096449443
***
[ 0.17] Pyomo Finished
```

The data used in this example was generated using β=2 and α=1. We can see that our estimates are quite reasonable. This example will be further expanded in Chapter 10 where we will demonstrate how to plot the data and the solution.

8.4.4 Reactor Design

Chemical reactors are often the most important unit operations in a chemical plant. Reactors come in many forms, however two of the most common idealizations are the continuously stirred tank reactor (CSTR) and the plug flow reactor. The CSTR is often used in modeling studies, and it can be effectively modeled as a lumped parameter system. In this example, we will consider the following reaction scheme known as the Van de Vusse reaction:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$

$$2A \xrightarrow{k_3} D$$

A diagram of the system is shown in Figure 8.2, where F is the volumetric flowrate. The reactor is assumed to be filled to a constant volume, and the mixture is assumed to have constant density, therefore the volumetric flowrate into the reactor is equal to the volumetric flowrate out of the reactor. Since the reactor is assumed to be well-mixed, the concentrations in the reactor are equivalent to the concentrations of each component flowing out of the reactor, given by c_A, c_B, c_C, and c_D, respectively.

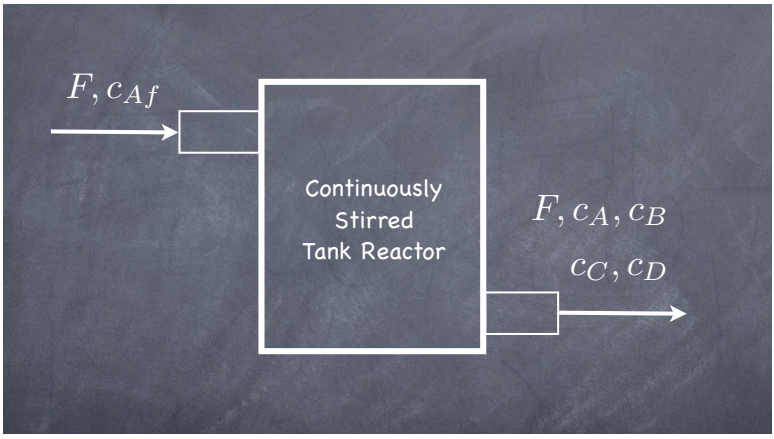

Fig. 8.2 Continuously stirred tank reactor system producing desired product B, and undesired products C, and D from A.

Consider the following reactor problem that was adapted from Bequette [8]. The goal is to produce product B from a feed containing reactant A. If we design a reactor that is too small, we will obtain insufficient conversion of A to the desired product B. However, given the above reaction scheme, if the reactor is too large (e.g., too much reaction is allowed to occur), a significant amount of the desired product B will be further reacted to form the undesired product C. Therefore, our goal in this

exercise will be to solve for the optimal reactor volume that produces the maximum outlet concentration for product B.

The steady-state mole balances for each of the four components are given by,

$$
\begin{aligned}
0 &= \frac{F}{V}c_{Af} - \frac{F}{V}c_A - k_1 c_A - 2k_3 c_A^2 \\
0 &= -\frac{F}{V}c_B + k_1 c_A - k_2 c_B \\
0 &= -\frac{F}{V}c_C + k_2 c_B \\
0 &= -\frac{F}{V}c_D + k_3 c_A^2
\end{aligned}
$$

The known parameters for the system are,

$$
c_{Af} = 10\ \frac{\text{gmol}}{\text{m}^3} \quad k_1 = \frac{5}{6}\ \text{min}^{-1} \quad k_2 = \frac{5}{3}\ \text{min}^{-1} \quad k_3 = \frac{1}{6000}\ \frac{\text{m}^3}{\text{mol min}}.
$$

Since the volumetric flowrate F always appears as the numerator over the reactor volume V, it is common to consider this ratio as a single variable, called the space-velocity (`sv`). Our optimization formulation will seek to find the space-velocity that maximizes the outlet concentration of the desired product B.

Example 8.A.5.1 provides a Pyomo model for this reactor design problem. In this example, we will illustrate the use of a `ConcreteModel` instead of the more common `AbstractModel`. The following pyomo command is used to solve this problem:

```
#!/bin/sh

pyomo --solver=ipopt --summary --stream-solver ReactorDesign.py
```

The following output is produced:

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.00] Creating model
[ 0.00] Applying solver
Ipopt 3.9.2:

******************************************************
This program contains Ipopt, a library for large-scale non
 Ipopt is released as open source code under the Eclipse P
       For more information visit http://projects.coin-o
******************************************************

This is Ipopt version 3.9.2, running with linear solver ma

Number of nonzeros in equality constraint Jacobian...:
Number of nonzeros in inequality constraint Jacobian.:
Number of nonzeros in Lagrangian Hessian.............:

Total number of variables............................:
```

```
                variables with only lower bounds:
             variables with lower and upper bounds:
                variables with only upper bounds:
Total number of equality constraints.................:
Total number of inequality constraints...............:
      inequality constraints with only lower bounds:
  inequality constraints with lower and upper bounds:
      inequality constraints with only upper bounds:

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg)
  0 -2.0000000e+03 3.17e+02 6.25e-01 -1.0 0.00e+00 -
  1 -1.0801475e+03 2.71e+01 2.46e+00 -1.0 1.39e+03 -
  2 -1.0763574e+03 4.43e+00 1.87e+02 -1.0 3.66e+02 -
  3 -1.0727252e+03 5.01e-01 5.26e+00 -1.0 9.35e+01 -
  4 -1.0726714e+03 1.02e-01 1.22e-01 -1.0 6.03e+01 -
  5 -1.0724371e+03 1.72e-05 2.93e-05 -2.5 4.19e-01 -
  6 -1.0724372e+03 4.04e-10 3.93e-09 -3.8 3.85e-03 -
  7 -1.0724372e+03 1.51e-12 2.69e-11 -5.7 2.24e-04 -
  8 -1.0724372e+03 4.55e-14 7.81e-14 -8.6 2.78e-06 -

Number of Iterations....: 8

                          (scaled)
Objective...............: -1.0724372001073814e+03 -1.07
Dual infeasibility......: 7.8080721909049831e-14 7.80
Constraint violation....: 4.5474735088646414e-14 4.54
Complementarity.........: 2.5059065225775025e-09 2.50
Overall NLP error.......: 2.5059065225775025e-09 2.50


Number of objective function evaluations = 9
Number of objective gradient evaluations = 9
Number of equality constraint evaluations = 9
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 9
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 8
Total CPU secs in IPOPT (w/o function evaluations) =
Total CPU secs in NLP function evaluations =

EXIT: Optimal Solution Found.
[ 0.10] Processing results
   Number of solutions: 1
   Solution Information
     Gap: <undefined>
     Status: unknown
   Solver results file: results.yml

============================================================
Solution Summary
============================================================

Model unknown
```

```
  Variables:
    Variable sv : Size=1 Domain=PositiveReals
      Value=1.34381176107
    Variable ca : Size=1 Domain=PositiveReals
      Value=3874.25886724
    Variable cb : Size=1 Domain=PositiveReals
      Value=1072.43720011
    Variable cc : Size=1 Domain=PositiveReals
      Value=1330.09353341
    Variable cd : Size=1 Domain=PositiveReals
      Value=1861.60519963

  Objectives:
    Objective obj : Size=1
      Value=1072.43720011

  Constraints:
    Constraint ca_bal : Size=1
      Value=-6.29370333627e-9
    Constraint cb_bal : Size=1
      Value=4.86625140184e-9
    Constraint cc_bal : Size=1
      Value=-3.57454155164e-9
    Constraint cd_bal : Size=1
      Value=-5.0026756071e-9

[ 0.11] Applying Pyomo postprocessing actions
[ 0.11] Pyomo Finished
```

There are a few important things to notice. First, the `--stream-solver` option is used to display the output produced by IPOPT while solving the problem. In many nonlinear examples, when the solver fails to find a solution or the behavior is unexpected, this output can often provide valuable information to correct the issue. In this case, the solver is successful at finding the optimal solution. Second, we also added the `--summary` option to print the solution to the screen after solving the problem. In this output, we see that the optimal space-velocity is `sv=1.34`, giving an outlet concentration for B of `cb=1072`.

We can further verify this solution using the Python script shown in Example 8.A.5.2. This script fixes the value of the space-velocity variable, and it solves the square problem repeatedly for different values of the space-velocity. We print a table of space-velocity versus the outlet concentration of B. Notice the new scripting commands at the end of the model definition that loop over the different space-velocity values and obtain the solution of the series of square problems.

This script can be executed simply using the `python` command:

```
#!/bin/sh

python ReactorDesignTable.py
```

This produces the following tabular results where we can see that the solution we obtained does in fact correspond to the local optimum:

```
         sv cb
       1.05 1060.84692138
        1.1 1064.77717388
       1.15 1067.78673119
        1.2 1069.97476354
       1.25 1071.42857143
        1.3 1072.22525762
       1.35 1072.43312302
        1.4 1072.11283896
       1.45 1071.31843739
        1.5 1070.09815137
       1.55 1068.49513205
        1.6 1066.54806288
       1.65 1064.29168809
        1.7 1061.75726893
       1.75 1058.97297921
        1.8 1055.96424923
       1.85 1052.75406573
        1.9 1049.36323446
       1.95 1045.81061049
```

This example illustrates the scripting capabilities of Pyomo. See Chapter 10 for more scripting examples and further description of these capabilities.

8.A Examples

8.A.1 Rosenbrock

`RosenBrock.py`: A Pyomo model for the Rosenbrock function.

```
from coopr.pyomo import *

model = AbstractModel()

model.x = Var(initialize = 1.5)
model.y = Var(initialize = 1.5)

def rosenbrock(MOD):
   return (1.0-MOD.x)**2 \
      + 100.0*(MOD.y - MOD.x**2)**2
model.obj = Objective(rule=rosenbrock, sense=minimize)
```

8.A.2 Multimodal

`multimodal_init1.py`: A Pyomo model that defines a simple multimodal test problem.

```
from coopr.pyomo import *
from math import pi

model = ConcreteModel()
model.x = Var(initialize = 0.25, bounds=(0,4))
model.y = Var(initialize = 0.25, bounds=(0,4))

def multimodal(m):
   return (2-cos(pi*m.x)-cos(pi*m.y)) * (m.x**2) * (m.y**2)
model.obj = Objective(rule=multimodal, sense=minimize)
```

8.A.3 Deer Harvesting

`DeerProblem.py`: An abstract Pyomo model for the formulation of the optimal sustainable deer harvesting problem.

```
from coopr.pyomo import *

#
# Model
#

model = AbstractModel()

model.p1 = Param();
model.p2 = Param();
model.p3 = Param();
model.p4 = Param();
model.p5 = Param();
model.p6 = Param();
model.p7 = Param();
model.p8 = Param();
model.p9 = Param();
model.ps = Param();

model.f = Var(initialize = 20, within=PositiveReals)
model.d = Var(initialize = 20, within=PositiveReals)
model.b = Var(initialize = 20, within=PositiveReals)

model.hf = Var(initialize = 20, within=PositiveReals)
model.hd = Var(initialize = 20, within=PositiveReals)
model.hb = Var(initialize = 20, within=PositiveReals)

model.br = Var(initialize=1.5, within=PositiveReals)
```

```
model.c = Var(initialize=500000, within=PositiveReals)


def obj_rule(amodel):
   return 10*amodel.hb + amodel.hd + amodel.hf
model.obj = Objective(rule=obj_rule, sense=maximize)

def f_bal_rule(amodel):
   return amodel.f == amodel.p1*amodel.br \
      *(amodel.p2/10.0*amodel.f + amodel.p3*amodel.d) \
      -amodel.hf
model.f_bal = Constraint(rule=f_bal_rule)

def d_bal_rule(amodel):
   return amodel.d == amodel.p4*amodel.d \
      + amodel.p5/2.0*amodel.f - amodel.hd
model.d_bal = Constraint(rule=d_bal_rule)

def b_bal_rule(amodel):
   return amodel.b == amodel.p6*amodel.b \
      + amodel.p5/2.0*amodel.f - amodel.hb
model.b_bal = Constraint(rule=b_bal_rule)

def food_cons_rule(amodel):
   return amodel.c == amodel.p7*amodel.b \
      + amodel.p8*amodel.d + amodel.p9*amodel.f
model.food_cons = Constraint(rule=food_cons_rule)

def supply_rule(amodel):
   return amodel.c <= amodel.ps
model.supply = Constraint(rule=supply_rule)

def birth_rule(amodel):
   return amodel.br == 1.1 + \
      0.8*(amodel.ps - amodel.c)/amodel.ps
model.birth = Constraint(rule=birth_rule)

def minbuck_rule(amodel):
   return amodel.b >= 1.0/5.0*(0.4*amodel.f + amodel.d)
model.minbuck = Constraint(rule=minbuck_rule)
```

`DeerProblem.dat`: The data file for the sustainable deer harvesting problem.

```
param p1 := 0.88;
param p2 := 0.82;
param p3 := 0.92;
param p4 := 0.84;
param p5 := 0.73;
param p6 := 0.87;
param p7 := 2700;
param p8 := 2300;
param p9 := 540;
param ps := 700000;
```

8.A.4 Disease Estimation

`DiseaseEstimation.py`: A Pyomo model for disease estimation.

```
from coopr.pyomo import *

model = AbstractModel()

model.S_SI = Set(ordered=True)

model.P_REP_CASES = Param(model.S_SI)
model.P_POP = Param()

model.I = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=1)
model.S = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=300)
model.beta = Var(bounds=(0.05, 70))
model.alpha = Var(bounds=(0.5, 1.5))
model.eps_I = Var(model.S_SI, initialize=0.0)

def _objective(model):
   return sum((model.eps_I[i])**2 for i in model.S_SI)
model.objective = Objective(rule=_objective, sense=minimize)

def _InfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.I[i-1]**model.alpha)/model.P_POP
   return Constraint.Skip

model.InfDynamics = Constraint(model.S_SI, \
    rule=_InfDynamics)

def _SusDynamics(model, i):
   if i != 1:
      return model.S[i] == model.S[i-1] - model.I[i]
   return Constraint.Skip
model.SusDynamics = Constraint(model.S_SI, \
    rule=_SusDynamics)

def _Data(model, i):
   return model.P_REP_CASES[i] == model.I[i]+model.eps_I[i]
model.Data = Constraint(model.S_SI, rule=_Data)
```

`DiseaseEstimation.dat`: A Pyomo data file that contains the set definitions, the population, and the case count data.

```
set S_SI := 1 2 3 4 5 6 7 8 9 10 11 12 13 14
        15 16 17 18 19 20 21 22 23 24 25 26 ;

param P_POP := 300.000000;

param P_REP_CASES :=
  1 1.000000
  2 2.000000
  3 4.000000
  4 8.000000
  5 15.000000
  6 27.000000
  7 44.000000
  8 58.000000
  9 55.000000
  10 32.000000
  11 12.000000
  12 3.000000
  13 1.000000
  14 0.000000
  15 0.000000
  16 0.000000
  17 0.000000
  18 0.000000
  19 0.000000
  20 0.000000
  21 0.000000
  22 0.000000
  23 0.000000
  24 0.000000
  25 0.000000
  26 0.000000
;
```

8.A.5 Reactor Design

8.A.5.1 Initial Design

`ReactorDesign.py`: A concrete Pyomo model for a simple reactor design problem.

```
from coopr.pyomo import *

# create the concrete model
model = ConcreteModel()

# set the data (native python data)
k1 = 5.0/6.0 # min^-1
k2 = 5.0/3.0 # min^-1
k3 = 1.0/6000.0 # m^3/(gmol min)
caf = 10000.0 # gmol/m^3

# create the variables
model.sv = Var(initialize = 1.0, within=PositiveReals)
model.ca = Var(initialize = 5000.0, within=PositiveReals)
model.cb = Var(initialize = 2000.0, within=PositiveReals)
model.cc = Var(initialize = 2000.0, within=PositiveReals)
model.cd = Var(initialize = 1000.0, within=PositiveReals)

# create the objective
model.obj = Objective(expr = model.cb, sense=maximize)

# create the constraints
model.ca_bal = Constraint(expr = (0 == model.sv * caf \
             - model.sv * model.ca - k1 * model.ca \
             - 2.0 * k3 * model.ca ** 2.0))

model.cb_bal = Constraint(expr=(0 == -model.sv * model.cb \
             + k1 * model.ca - k2 * model.cb))

model.cc_bal = Constraint(expr=(0 == -model.sv * model.cc \
             + k2 * model.cb))

model.cd_bal = Constraint(expr=(0 == -model.sv * model.cd \
             + k3 * model.ca ** 2.0))
```

8.A.5.2 Design Table

`ReactorDesignTable.py`: A script to iteratively solve the reactor design problem for different values of the space-velocity.

```
from coopr.pyomo import *
from pyutilib.misc import Options

# create the concrete model
model = ConcreteModel()

# set the data (native python data)
k1 = 5.0/6.0 # min^-1
k2 = 5.0/3.0 # min^-1
k3 = 1.0/6000.0 # m^3/(gmol min)
caf = 10000.0 # gmol/m^3

# create the variables
model.sv = Var(initialize = 1.0, within=PositiveReals)
model.ca = Var(initialize = 5000.0, within=PositiveReals)
model.cb = Var(initialize = 2000.0, within=PositiveReals)
model.cc = Var(initialize = 2000.0, within=PositiveReals)
model.cd = Var(initialize = 1000.0, within=PositiveReals)

# create the objective
model.obj = Objective(expr = model.cb, sense=maximize)

# create the constraints
model.ca_bal = Constraint(expr = (0 == model.sv * caf \
             - model.sv * model.ca - k1 * model.ca \
             - 2.0 * k3 * model.ca ** 2.0))

model.cb_bal = Constraint(expr=(0 == -model.sv * model.cb \
             + k1 * model.ca - k2 * model.cb))

model.cc_bal = Constraint(expr=(0 == -model.sv * model.cc \
             + k2 * model.cb))

model.cd_bal = Constraint(expr=(0 == -model.sv * model.cd \
             + k3 * model.ca ** 2.0))

# setup the solver options
options = Options()
options.solver = 'ipopt'
options.quiet = True

# run the sequence of square problems
instance = model.create()
instance.sv.fixed = True
sv_values = [1.0 + v * 0.05 for v in range(1, 20)]
print " ", 'sv'.rjust(10), 'cb'.rjust(10)
for sv_value in sv_values:
   instance.sv = sv_value
   results, opt = \
      scripting.util.apply_optimizer(options, instance)
   instance.load(results)
   print " ", str(instance.sv.value).rjust(10),\
      str(instance.cb.value).rjust(15)
```

Chapter 9
Stochastic Programming Extensions

Abstract This chapter describes PySP, a stochastic programming extension to Pyomo. PySP enables the expression of stochastic programming problems as extensions of deterministic models, which are often formulated first. To formulate a stochastic program in PySP, the user specifies both the deterministic base model and the scenario tree with associated uncertain parameters in Pyomo. Given these two models, PySP provides two paths for solving the corresponding stochastic program. The first alternative involves writing the extensive form and invoking a standard deterministic (mixed-integer) solver. For more complex stochastic programs, PySP includes an implementation of Rockefeller and Wets' Progressive Hedging algorithm, which provides an effective heuristic for approximating general multi-stage, mixed-integer stochastic programs. By leveraging the combination of a high-level programming language (Python) and the embedding of the base deterministic model in that language (Pyomo), PySP provides completely generic and highly configurable solver implementations.

9.1 Introduction

From the earliest days of using computers to optimization problems, it was recognized that input data is uncertain in most real-world decision problems [20]. In cases where parameter uncertainty is independent of the decisions and information becomes available in a few time stage stages, stochastic programming is an appropriate and widely studied mathematical framework to express and solve uncertain decision problems [11, 58, 39, 54]. Stochastic programming allows the user to explicitly account for the fact that some of the data is uncertain and that the values of parameters may become known over time.

Nevertheless, few solvers devoted to generic stochastic programs include mixed-integer linear or nonlinear problems. With few exceptions, modeling packages that support stochastic programming rely on the translation of problem into the *extensive form* – a deterministic mathematical programming representation of the stochastic

W.E. Hart et al., *Pyomo—Optimization Modeling in Python,*
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_9,

program in which all scenarios are explicitly represented and solved simultaneously. The extensive form can then be optimized with a standard solver. The number of scenarios, the size of the underlying model, the number of decision stages, and the presence of discrete decision variables are factors that affect the scale of the extensive form. Unfortunately, for real-world problems the direct solution of the extensive form is often (a) too difficult to solve or (b) too large to solve with available system memory.

Iterative decomposition strategies such as the L-shaped method [55] or Progressive Hedging [51] directly address both of these scalability issues, but these methods introduce fundamental parameters and algorithmic challenges. Other approaches include coordinated branch-and-cut procedures [3]. In general, the solution of difficult stochastic programs requires both experimentation with and customization of alternative algorithmic paradigms – thus necessitating for generic and configurable solvers.

PySP is a recently developed stochastic programming extension for Pyomo [59]. To express a stochastic program in PySP, the user specifies both the deterministic base model and the scenario tree with associated uncertain parameters in Pyomo. This separation of deterministic and stochastic problem components is similar to the mechanism proposed in SMPS [12, 25].

Once the deterministic and scenario tree models have been specified, PySP provides two paths for solving the corresponding stochastic program. The first alternative involves writing the extensive form and invoking a deterministic (mixed-integer) linear solver. For more complex stochastic programs, a generic implementation of Rockefeller and Wets' Progressive Hedging algorithm [51] can be applied. The development of PySP has focused on the use of Progressive Hedging as an effective heuristic for approximating general multi-stage, mixed-integer programs. PySP provides a completely generic and highly configurable solver implementation for Progressive Hedging by leveraging the combination of a high-level programming language (Python) and the embedding of the base deterministic model in that language (Pyomo). Additionally, PySP leverages Pyomo to provide access to the full range of solvers supported within Coopr. Consequently, a broad range of model types can be addressed by PySP.

9.2 Stochastic Programming: Definition and Notations

PySP is designed to express and solve stochastic programming problems, which we now briefly introduce. Readers with no background in stochastic programming will probably need to make use of more comprehensive introductions to both the theoretical foundations and the range of potential applications that can be found in Birge and Leouvaux [11], Shapiro et al. [54], and Wallace and Ziema [58].

We concern ourselves with stochastic optimization problems where uncertain parameters (data) can be represented by a finite set of scenarios $\mathscr{S}$, each of which specifies both (1) a full set of random variable realizations and (2) a corresponding

probability of occurrence. The random variables in question specify the evolution of uncertain parameters over time. We index the scenario set by s and refer to the probability of occurrence of s (or, more accurately, a realization "near" scenario s) as $\Pr(s)$. Let the number of scenarios be given by $|\mathscr{S}|$. The source of these scenarios does not concern us at this point, although we observe that they are frequently obtained via simulation or formed from historical data or expert opinions. We assume that the decision process of interest consists of a sequence of discrete time stages, the set of which is denoted $\mathscr{T}$. We index $\mathscr{T}$ by t, and denote the number of time stages by $|\mathscr{T}|$.

Because PySP can support some types of nonlinear constraints and objectives as well as linear and mixed-integer problems, we develop the notation in a fairly abstract way. For each scenario s and time stage t, $t \in \{1,\ldots,|\mathscr{T}|\}$, we are given a function $f_s(\cdot)$. For each $t \in \{1,\ldots,|\mathscr{T}|\}$, the decision variables in a stochastic program consist of a set of the vectors $x(s,t)$, one vector for each scenario $s \in \mathscr{S}$. Let $X(s)$ be $(x(s,1),\ldots,x(s,\mathscr{T}))$. We will use X as shorthand for the entire *solution system* of x vectors, i.e., $X = x(1,1),\ldots,x(|\mathscr{S}|,|\mathscr{T}|)$.

If we were prescient enough to know which scenario $s \in \mathscr{S}$ would be ultimately realized, our optimization objective would be to minimize

$$f_s(X(s)) \quad (\mathrm{P}_s)$$

subject to the constraint

$$X(s) \in \Omega_s.$$

We use Ω_s as an abstract notation to express all constraints for scenario s, including requirements that some decision vector elements are discrete or more general requirements.

We must obtain solutions that do not require foreknowledge and that will be feasible independent of which scenario is ultimately realized. We refer to solution systems that satisfy constraints for all scenarios as *admissible*. We refer to a system of solution vectors as *implementable* if for all pairs of scenarios, s and s', that are indistinguishable up to time t, $x_i(s,t') = x_i(s',t')$ for every i in every $N(t)$ for all $1 \le t' \le t$. We refer to the set of all implementable solution systems as $\mathscr{N}_{\mathscr{S}}$ for a given set of scenarios, $\mathscr{S}$.

To achieve admissible and implementable solutions, the expected-value minimization problem then becomes:

$$\min \sum_{s \in \mathscr{S}} \Pr(s) f_s(X(s)) \quad (\mathrm{P})$$

subject to

$$X(s) \in \Omega_s,\ s \in \mathscr{S}$$
$$X \in \mathscr{N}_{\mathscr{S}}.$$

Formulation (P) is known as a stochastic mathematical program. If all decision variables are continuous, we refer to the problem simply as a stochastic program. If

some of the decision variables are discrete, we refer to the problem as a stochastic mixed-integer program. We observe that, lacking prescience, only solutions that are implementable are useful. Solutions that are not admissible, on the other hand, may have some value because while some constraints may represent laws of physics, others may be violated slightly without serious consequence.

In practice, the parameter uncertainty in stochastic programs is often encoded via a *scenario tree*, in which every node is associated with a time stage and a list of scenarios whose parameter values are indistinguishable up to that time. Scenario trees are discussed in more detail in Section 9.3.2

9.3 Modeling in PySP

Pyomo allows non-specialists to easily formulate and solve deterministic mathematical programming models, avoiding the need for a deep understanding of the underlying algorithmic technologies that are used to analyze these models; PySP strives to provide similar capabilities for stochastic mathematical programming models.

In this section, we discuss the use of Pyomo to formulate and express stochastic programs in PySP. As a motivating example, we consider the well-known Birge and Louveaux [11] "farmer" stochastic program. Mirroring several other approaches to modeling stochastic programs (e.g., see Thénié et al. [56]), we require the specification of two related components: the deterministic base model and the scenario tree. In Section 9.3.1 we discuss the specification of the deterministic reference model and associated data; Section 9.3.2 details the model and data underlying PySP scenario tree specification. The mechanisms for specifying uncertain parameter data are discussed in Section 9.3.3. Finally, we briefly discuss the programmatic compilation of a scenario tree specification into objects appropriate for use by solvers in Section 9.3.4.

9.3.1 The Deterministic Reference Model

The starting point for developing a stochastic programming model in PySP is the specification of an abstract *reference* model, which describes the deterministic multi-stage problem for an arbitrary, canonical scenario. The reference model does not make use of, or describe, any information relating to parameter uncertainty or the scenario tree. Typically, it is simply the model that would be used in single-scenario analysis, i.e., the model that is commonly developed before stochastic aspects of an optimization problem are considered. PySP requires that the reference model – specified in Pyomo – is contained in a file named `ReferenceModel.py`. The complete reference model file for Birge and Louveaux's farmer problem is shown Example 9.A.1.

The reference model is independent of any stochastic components of the problem; however, PySP does require that the objective cost component for each decision stage of the stochastic program be assigned to a distinct singleton variable or an element of a variable array. In the reference model for the farmer problem, we simply label the first and second stage cost variables as `FirstStageCost` and `SecondStageCost`, respectively. The corresponding values are computed via the constraints
`ComputeFirstStageCost` and `ComputeSecondStageCost`.

To create a model instance from the abstract reference model, a reference data file must also be specified. The data can correspond to an arbitrary scenario, and must completely specify all parameters in the abstract reference model. The reference data file must be named `ReferenceModel.dat`. An example data file corresponding to the farmer reference model is shown in Example 9.A.2. Although Pyomo supports various data file formats, the example illustrates the use of a data command file.

9.3.2 The Scenario Tree

The second step in developing a stochastic program in PySP is to specify the scenario tree structure and associated parameter data. A PySP scenario tree specification supplies all information concerning the time stages, the mapping of decision variables to time stages, how various scenarios are temporally related to one another (i.e., scenario tree nodes and their inter-relationships), and the probabilities of various scenarios. As discussed below, the scenario tree does not directly specify uncertain parameter values; rather, it specifies references to data files containing such data.

The scenario tree is defined with a scenario tree model, which itself happens to be a Pyomo model. Example 9.A.3 shows the scenario tree model used by PySP, but note that the form of this model is fixed. The model is built into and distributed with PySP; the user does not edit this model. The user simply supplies values for each of the parameters specified in the scenario tree model.

The precise semantics for each of the parameters (or sets) indicated in the scenario tree model in Example 9.A.3 are as follows:

`Stages` An ordered set containing the names (specified as arbitrary strings) of the time stages. The order corresponds to the time order of the stages.

`Nodes` A set of the names (specified as arbitrary strings) of the nodes in the scenario tree.

`NodeStage` An indexed parameter mapping node names to stage names. Each node in the scenario tree must be assigned to a specific stage.

`Children` An indexed set mapping node names to sets of node names. For each non-leaf node in the scenario tree, a set of child nodes must be specified. This set implicitly defines the overall branching structure of the scenario tree. Using

this set, the parent nodes are computed internally to PySP. There can only be one node in the scenario tree with no parents, i.e., the tree must be singly rooted.

`ConditionalProbability` An indexed parameter mapping node names to their conditional probability, relative to their parent node. The conditional probability of the root node must be equal to 1, and for any node with children, the conditional probabilities of the children must sum to 1. Numeric values must be contained within the interval $[0,1]$.

`Scenarios` An ordered set containing the names (specified as arbitrary strings) of the scenarios. These names are used for two purposes: reporting and data specification (see Section 9.3.3). The ordering is provided as a convenience for the user, to organize reporting output.

`ScenarioLeafNode` An indexed parameter mapping scenario names to their leaf node name. This data facilitates linkage of the scenarios to their composite nodes in the scenario tree.

`StageVariables` An indexed set mapping stage names to sets of variable names in the reference model. The sets of variables names indicate those variables that are associated with the given stage. This implicitly defines the non-anticipativity constraints that should be imposed when generating and/or solving the PySP model.

`ScenarioBasedData` A boolean parameter specifying how the instances for each scenario are to be constructed. A value of `True`, which is the default, indicates that scenario data will be given in files that specify all data for each scenario, even the data that are not stochastic. A value of `False`, indicates that the data will be given in a file for each node of the scenario tree. See Section 9.3.3 for further details.

The data that is used to instantiate these parameters and sets must be provided in a file named `ScenarioStructure.dat`. The scenario tree structure specification for the farmer problem is shown in Example 9.A.4, specified using a data command file. This example illustrates how PySP provides a simple "slicing" syntax to specify subsets of indexed variables. The "*" character is used to match all values in a particular dimension of an indexed parameter. In more complex examples, variables are typically indexed by time stage. In these cases, the slice syntax allows for very concise specification of the stage-to-variable mapping.

Finally, we observe that PySP makes no assumptions regarding the linkage between time stages and variable index structure. In particular, the time stage need not explicitly be referenced within a variable's index set. While this is often the case in multi-stage formulations, the convention is not universal, e.g., as in the case of the farmer problem.

9.3.3 Scenario Parameter Specification

Data files specifying the (deterministic and stochastic) parameters for each of the scenarios in a PySP model can be specified in one of two ways. The simplest ap-

proach is "scenario-based", in which each scenario is defined by a separate data file that provides a *complete* parameter specification for the scenario. If the scenario is named `ScenarioX`, then the corresponding data file for the scenario must be named `ScenarioX.dat`. This approach is often expedient – especially if the scenario data are generated via simulation, which is often convenient in practice. However, there is redundancy in this encoding of the model parameters. Depending on the problem size and number of scenarios, this redundancy may become excessive in terms of disk storage and access. Scenario-based data specification is the default behavior in PySP, as indicated by the default value of the `ScenarioBasedData` parameter in Example 9.A.3. Note that the listing in Example 9.A.2 is an example of a scenario-based data specification.

Node-based parameter specification is provided as an alternative to the default scenario-based approach, principally to eliminate storage redundancy. With a node-based specification, parameter data specific to each `node` in the scenario tree is specified in a distinct data file. If the node is named `NodeX`, then the corresponding data file for the node must be named `NodeX.dat`. To create a scenario instance, data for all nodes associated with a scenario are accessed (via the `ScenarioLeafNode` parameter in the scenario tree specification and the computed parent node linkages). Node-based parameter encoding eliminates redundancy, although typically at the expense of a slightly more complex instance generation process. To enable node-based scenario initialization, a user needs to simply add the following line to `ScenarioStructure.dat`:

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

In the case of the farmer problem, all parameters except for `Yield` are identical across all scenarios. Consequently, these parameters can be placed in a file named `RootNode.dat`. Then, files containing scenario-specific `Yield` parameter values are specified for each second-stage leaf node in files with names:

- AboveAverageNode.dat
- AverageNode.dat
- BelowAverageNode.dat

9.3.4 Compilation of the Scenario Tree Model

The PySP scenario tree structure model is a declarative entity, specifying the data associated with a scenario tree. PySP uses the information contained in this model internally to construct a `ScenarioTree` object, which, in turn, is composed of `ScenarioTreeNode`, `Stage`, and `Scenario` objects. In aggregate, these Python objects allow programmatic navigation, query, manipulation, and reporting of the scenario tree structure. While hidden from the typical user, these objects are crucial in the processes of generating the extensive form (Section 9.4) and generic solvers (Section 9.5).

9.4 Generating and Solving the Extensive Form

The most straightforward method to solve a stochastic program involves generating the *extensive form* (also known as the *deterministic equivalent*) and then invoking a standard deterministic (mixed-integer) programming solver. The extensive form given as problem (P) in Section 9.2 completely specifies all scenarios and the coupling non-anticipativity constraints at each node in the scenario tree. Although more scalable solution techniques may be needed for large, real-world stochastic programs, the extensive form is usually the first method applied to solve a stochastic program.

PySP provides the `runef` command to both generate and solve the extensive form for a given stochastic program. The following are the primary command-line options for this command:

`--help`
: Display all command-line options, with brief descriptions.

`--verbose`
: Display verbose output to the standard output stream, above and beyond the usual status output. This generates a lot of output, so it is disabled by default.

`--model-directory=MODEL_DIRECTORY`
: Specifies the directory in which the reference model (`ReferenceModel.py`) is stored. The default is the current working directory.

`--instance-directory=INSTANCE_DIRECTORY`
: Specifies the directory in which the reference model (`ReferenceModel.dat`), the scenario structure (`ScenarioStructure.dat`) and scenario data files are stored. The default is the current working directory.

`--output-file=OUTPUT_FILE`
: Specifies the name of the LP format output file to which the extensive form is written. The default is `efout.lp`.

`--solve`
: Directs the command to solve the extensive form after writing it. This is disabled by default.

`--solver=SOLVER_TYPE`
: Specifies the type of solver for solving the extensive form, if a solve is requested. The default is `cplex`.

`--solver-options=SOLVER_OPTIONS`
: Specifies solver options in keyword-value pair format, if a solve is requested. These keywords and values are passed directly to the solver.

`--output-solver-log`
: Specifies that the output of the solver is to be echoed to the standard output stream. This is disabled by default. This is used to ascertain status for extensive forms with long solve times.

Note that all options begin with a double dash prefix. The full set of arguments for `runef` can be obtained using the `--help` option.

For example, to write and solve the farmer problem (provided with the Coopr installation, in the directory `coopr/examples/pysp/farmer`), the command is:

```
runef --model-directory=models \
     --instance-directory=scenariodata \
     --solve --solver=glpk
```

Following solver execution, the resulting solution is loaded and displayed. The solution output is split into two distinct components: variable values and stage/scenario costs. For the farmer example, the per-node variable values are given as

```
Tree Nodes:

      Name=AboveAverageNode
      Stage=SecondStage
      Parent=RootNode
      Variables:
            QuantitySubQuotaSold[CORN]=48.0
            QuantitySubQuotaSold[SUGAR_BEETS]=6000.0
            QuantitySubQuotaSold[WHEAT]=310.0

      Name=AverageNode
      Stage=SecondStage
      Parent=RootNode
      Variables:
            QuantitySubQuotaSold[SUGAR_BEETS]=5000.0
            QuantitySubQuotaSold[WHEAT]=225.0

      Name=BelowAverageNode
      Stage=SecondStage
      Parent=RootNode
      Variables:
            QuantitySubQuotaSold[SUGAR_BEETS]=4000.0
            QuantitySubQuotaSold[WHEAT]=140.0
            QuantityPurchased[CORN]=48.0

      Name=RootNode
      Stage=FirstStage
      Parent=None
      Variables:
            DevotedAcreage[CORN]=80.0
            DevotedAcreage[SUGAR_BEETS]=250.0
            DevotedAcreage[WHEAT]=170.0
```

Similarly, the per-node stage costs are given as

```
Tree Nodes:

      Name=AboveAverageNode
      Stage=SecondStage
      Parent=RootNode
      Conditional probability=0.3333
      Children:
            None
      Scenarios:
            AboveAverageScenario
      Expected node cost=-275900.0000

      Name=AverageNode
      Stage=SecondStage
      Parent=RootNode
      Conditional probability=0.3333
      Children:
            None
      Scenarios:
            AverageScenario
      Expected node cost=-218250.0000

      Name=BelowAverageNode
      Stage=SecondStage
      Parent=RootNode
      Conditional probability=0.3333
      Children:
            None
      Scenarios:
            BelowAverageScenario
      Expected node cost=-157720.0000

      Name=RootNode
      Stage=FirstStage
      Parent=None
      Conditional probability=1.0000
      Children:
            AboveAverageNode
            AverageNode
            BelowAverageNode
      Scenarios:
            AboveAverageScenario
            AverageScenario
            BelowAverageScenario
      Expected node cost=-108390.0000
```

and the per-scenario overall costs are

```
Scenarios:

      Name=AboveAverageScenario
      Probability=0.3333
      Leaf Node=AboveAverageNode
      Tree node sequence:
            RootNode
            AboveAverageNode
      Stage= FirstStage Cost=108900.0000
      Stage= SecondStage Cost=-275900.0000
      Total scenario cost=-167000.0000

      Name=AverageScenario
      Probability=0.3333
      Leaf Node=AverageNode
      Tree node sequence:
            RootNode
            AverageNode
      Stage= FirstStage Cost=108900.0000
      Stage= SecondStage Cost=-218250.0000
      Total scenario cost=-109350.0000

      Name=BelowAverageScenario
      Probability=0.3333
      Leaf Node=BelowAverageNode
      Tree node sequence:
            RootNode
            BelowAverageNode
      Stage= FirstStage Cost=108900.0000
      Stage= SecondStage Cost=-157720.0000
      Total scenario cost=-48820.0000
```

Currently, the extensive form is output for solution in the LP file format. In practice, this has not been a limitation, since this LP file format is compatible with the "LP" format supported by many commercial and open-source solvers.

Various other command-line options are available in the `runef` command, including those related to performance profiling and Python garbage collection. Further, the `runef` command is capable of writing and solving the extensive form augmented with a weighted Conditional Value at Risk term in the objective [53].

The `runef` command produces the extensive form by introducing master variables to use in explicit non-anticipativity constraints. Although more compact formulations could be generated, there do not appear to be compelling reasons for doing so. The presolvers in commercial packages such as CPLEX, GUROBI, or XpressMP (and those available with some open-source solvers) are able to quickly identify and eliminate most of the redundant variables and constraints that are generated in the extensive form generated by `runef`.

9.5 Progressive Hedging: A Generic Decomposition Strategy

We now describe a decomposition strategy for optimizing stochastic programs, which is often required in practice for large-scale instances with large numbers of scenarios, discrete variables, or decision stages. There are two broad classes of decomposition-based strategies: horizontal and vertical. *Vertical* strategies decompose a stochastic program by time stages; Van Slyke and Wets' L-shaped method is the primary method in this class [55]. In contrast, *horizontal* strategies decompose a stochastic program by scenario; Rockafellar and Wets' Progressive Hedging algorithm [51] and Caroe and Schultz's Dual Decomposition (DD) algorithm [14] are the two notable methods in this class.

Progressive Hedging (PH) was initially introduced as a decomposition strategy for solving large-scale stochastic linear programs [51]. PH is a horizontal, or scenario-based, decomposition technique, and it possesses theoretical convergence properties when all decision variables are continuous. In particular, the algorithm has a linear convergence rate given a convex reference scenario optimization model.

Despite its introduction in the context of stochastic linear programs, PH has proven to be a very effective heuristic for solving stochastic mixed-integer programs. PH is particularly effective in this context when there are computationally efficient techniques for solving the deterministic single-scenario optimization problems. A key advantage of PH in the mixed-integer case is the absence of requirements concerning the number of stages or the type of variables allowed in each stage – as is common for many proposed stochastic mixed-integer algorithms. A disadvantage is the current lack of provable convergence and optimality results. However, PH has been used as an effective heuristic for a broad range of stochastic mixed-integer programs [21, 19, 36, 41, 42]. For large, real-world stochastic mixed-integer programs, the determination of optimal solutions is generally not computationally tractable, so a heuristic solver like PH is quite useful and practical.

Research on computational aspects of PH is ongoing. Numerous ways to speed convergence are controlled by parameters in PySP, but there are not yet methods for setting their values automatically. Hence, an understanding of the PH algorithm is needed to effectively apply it in practice. The basic idea of PH for the linear case is as follows:

1. For each scenario s, solutions are obtained for the problem of minimizing, subject to the problem constraints, the deterministic f_s (Formulation P_s).
2. The variable values for an implementable – but likely not admissible – solution are obtained by averaging over all scenarios at a scenario tree node.
3. For each scenario s, solutions are obtained for the problem of minimizing, subject to the problem constraints, the deterministic f_s (Formulation P_s) plus terms that penalize the lack of implementability using a sub-gradient estimator for the non-anticipativity constraints and a squared penalty term.
4. If the solutions have not converged sufficiently and the allocated compute time is not exceeded, goto Step 2.
5. Post-process, if needed, to produce a fully admissible and implementable solution.

Progressive Hedging is a technique to iteratively and gradually enforce implementability, while maintaining admissibility at each step in the process. For each scenario s, approximate solutions are obtained for the problem of minimizing, subject to the constraints, the deterministic f_s plus terms that penalize the lack of implementability. These terms strongly resemble those found when the method of augmented Lagrangians is used [9]. The method makes use of a system of row vectors, w, that have the same dimension as the column vector system X, so we use the same shorthand notation. For example, $w(s)$ denotes$(w(s,1),\ldots,w(s,|\mathscr{T}|))$ in the multiplier system.

To provide a formal PH algorithm statement, we first formalize some of the scenario tree concepts. We use $\Pr(\mathscr{A})$ to denote the sum of $\Pr(s)$ over all s for scenarios emanating from node $\mathscr{A}$ (i.e., those s that are the leaves of the sub-tree having $\mathscr{A}$ as a root, also referred to as $s \in \mathscr{A}$). We use $t(\mathscr{A})$ to indicate the time index for node $\mathscr{A}$ (i.e., node $\mathscr{A}$ corresponds to time t). We use $X(t;\mathscr{A})$ on the left hand side of a statement to indicate assignment to the vector $(x_1(s,t),\ldots,x_{N(t)}(s,|\mathscr{T}|))$ for each $s \in \mathscr{A}$. We refer to vectors at each iteration of PH using a superscript; e.g., $w^{(0)}(s)$ is the multiplier vector for scenario s at PH iteration zero. The PH iteration counter is k.

Figure 9.1 provides a formal description of the PH algorithm (with step numbering that matches in the informal statement given above). Note that $\rho > 0$ is an algorithmic parameter. In addition to termination criteria based mainly on convergence, we must also allow for the use of time-based termination because non-convergence is a possibility. Iterations are continued until k reaches some pre-determined limit or the algorithm has *converged* – which we take to indicate that the set of scenario solutions s is sufficiently homogeneous. One possible definition requires the distance between solutions for each scenario sub-problem to be less than some parameter.

The value of the parameter ρ strongly influences the actual convergence rate of PH: if ρ is too small, the penalty coefficients will vary little between consecutive iterations. To achieve tractable PH runtimes, significant tuning and problem-dependent strategies for computing ρ are often required; mechanisms to support various strategies for setting ρ are described in Section 9.5.1.

9.5.1 The `runph` Script

Analogous to the `runef` command for generating and solving the extensive form, PySP provides a single point-of-entry command – `runph` – to solve and post-process stochastic programs via PH. In this section, we briefly describe the general usage of this command, followed by a discussion of some generally effective options to customize the execution of PH. A number of key options are shared with the `runef` command: `--verbose`, `--model-directory`, `--instance-directory`, and `--solver`. The `--model-directory` and `--instance-directory` options are used to specify the PySP problem in-

1. $k \longleftarrow 0$
2. For all scenario indexes, $s \in \mathscr{S}$:

$$X^{(0)}(s) \longleftarrow \text{argmin}\, f_s(X(s)) : X(s) \in \Omega_s \tag{9.1}$$

and

$$w^{(0)}(s) \longleftarrow 0$$

3. $k \longleftarrow k+1$
4. For each node, $\mathscr{A}$, in the scenario tree, and for all $t = t(\mathscr{A})$:

$$\overline{X}^{(k-1)}(t;\mathscr{A}) \longleftarrow \sum_{s \in A} \Pr(s) X(t;s)^{(k-1)} / \Pr(\mathscr{A})$$

5. For all scenario indices, $s \in \mathscr{S}$:

$$w^{(k)}(s) \longleftarrow w^{(k-1)}(s) + (\rho)\left(X^{(k-1)}(s) - \overline{X}^{(k-1)}\right)$$

and

$$X^k(s) \longleftarrow \text{argmin}\, f_s(X(s)) + w^{(k)}(s)X(s) + \rho/2 \left\| X(s) - \overline{X}^{k-1} \right\|^2 : X(s) \in \Omega_s. \tag{9.2}$$

6. If the termination criteria are not met (e.g., solution discrepancies quantified via a metric $g^{(k)}$), then go to Step 3.

Fig. 9.1 A formal description of the Progressive Hedging algorithm.

stance, while the `--solver` option is used to specify the solver applied to individual scenario sub-problems. The most general PH-specific options are:

`--max-iterations`=MAX_ITERATIONS
The maximal number of PH iterations, which defaults to 100.

`--default-rho`=DEFAULT_RHO
The global ρ scalar parameter value for all variables not given a ρ value by a configuration file. The default is 1, which may not work well for any particular instance.

`--termdiff-threshold`=TERMDIFF_THRESHOLD
The convergence threshold used to terminate PH (Step 6 of the pseudocode). This quantity is known as the *termdiff*. The default is 0.01, which is too low for most instances. The default convergence metric is the sum of deviations from the mean taken across all variables and scenarios:
$g^k = \sum_{s \in \mathscr{S}} \Pr(s) ||X^{(k)}(t;s) - \bar{X}^{(k)}(\mathscr{A})||$.

In general, the default values for the maximum allowable iteration count, ρ, and convergence threshold are likely to yield slow convergence of PH; for any real application, experimentation and analysis should be applied to obtain a more computationally effective configuration.

To illustrate the execution `runph` on a stochastic linear program, we again consider Birge and Louveaux's farmer problem. To solve the farmer problem with PySP, a user simply executes the following,

```
runph --model-directory=models \
      --instance-directory=scenariodata
```

which will result in eventual convergence to an optimal, admissible, and implementable solution – subject to the numerical tolerance issues. For the sake of brevity, we do not illustrate the output here; the final solution is reported in a format identical to that illustrated in Section 9.4. The quantity of information generated by PH can be significant, e.g., including the penalty weights and solutions for each scenario problem $s \in \mathscr{S}$ at each iteration. However, this information is not generated by default. Rather, simple summary information, including the value of $g^{(k)}$ at each PH iteration k, is output.

As is theoretically promised in the case of stochastic linear programs, `runph` does converge given a linear PySP input model. The exact number of iterations depends in part on the precise solver used; on our test platform, for example, convergence is achieved in 48 iterations using CPLEX 11.2.1. It should be noted that for many stochastic linear and small mixed-integer programs (including the farmer example), PH may solve *much more slowly* than the extensive form, primarily because of the overhead associated with communicating with solvers for each scenario, for each PH iteration. However, this overhead is negligible for larger and more difficult scenario problems, and larger numbers of scenarios. Perhaps more vexing, is that for large instances a bad value of ρ can result in non-convergence (or glacial rates of convergence).

Having described the basic functionality of `runph`, we now transition to a discussion of some issues with PH that can arise in practice, and their resolution via the `runph` command. More comprehensive configuration methods, to address more complex PH issues, are discussed in Section 9.6.

9.5.1.1 Variable-specific ρ

In many applications, a single value of ρ does not yield a computationally efficient PH configuration. Consider the situation in which the objective is to minimize expected investment costs in a spare parts supply chain, e.g., for maintaining an aircraft fleet. The acquisition cost for spare parts is highly variable, ranging from very expensive (engines) to very cheap (gaskets). If ρ values are too small, e.g., on the order of the price of a tire, PH will require large iteration counts to achieve changes – let alone convergence – in the decision variables associated with engine procurement counts. If ρ values are too high, e.g., on the order of the price of an engine, then the PH weights w associated with gasket procurement counts will converge too quickly, yielding sub-optimal variable values. Alternatively, PH sub-problem solves may "over-shoot" the optimal variable value, resulting in oscillation.

We have developed various strategies for computing variable-specific ρ val-

ues [60]. The following command-line option for `runph` is used to specify these strategies:

`--rho-cfgfile`=RHO_CFGFILE
The name of a configuration command to compute PH rho values. The default is None.

The configuration file is a piece of executable Python code that computes the desired ρ values. This allows the expression of arbitrarily complex formulas or procedures for computing ρ values. For example, the following configuration file is used in conjunction with the PySP `SIZES` example [38]:

```
model_instance = self._model_instance

for i in model_instance.ProductSizes:
 self.setRhoAllScenarios( \
       model_instance.ProduceSizeFirstStage[i], \
       model_instance.SetupCosts[i] * 0.001)
 self.setRhoAllScenarios( \
       model_instance.NumProducedFirstStage[i], \
       model_instance.UnitProductionCosts[i] * 0.001)
 for j in model_instance.ProductSizes:
  if j <= i:
    self.setRhoAllScenarios( \
       model_instance.NumUnitsCutFirstStage[i,j], \
       model_instance.UnitReductionCost * 0.001)
```

See the `rhosetter.cfg` file in the PySP SIZES examples directory. The `self` object in the script refers to the PH object itself, which, in turn, possesses an attribute `_model_instance`. The `_model_instance` attribute represents the instance of the deterministic reference model, from which the full set of problem variables can be accessed. The example script implements a simple cost-proportional ρ strategy, in which ρ is specified as a function of a variable's objective function cost coefficient. Once the appropriate ρ value is computed, the script invokes the `setRhoAllScenarios` method of the PH object, which distributes the computed ρ value to each of the scenario problem instances. It is also possible to set the ρ values on a per-variable, per-scenario basis.

The customization strategy underlying the PySP variable-specific ρ mechanism is a limited form of call-back function, in which the core PH code temporarily hands control back to a user script to set specific model parameters.

9.5.1.2 Linearization of the Proximal Penalty Terms

At each PH iteration $k \geq 1$, scenario sub-problems are solved with an augmented form of the original optimization objective, using both linear and quadratic penalty terms. The presence of the quadratic terms can cause significant practical difficulties. At present, no open-source linear or mixed-integer solvers currently support quadratic objective terms in an integrated, robust manner. While most commercial solvers can handle problems with quadratic objectives, but solver efficiency is often

dramatically worse relative to the linear case. We have consistently observed scenario sub-problem solve times an order of magnitude or larger on quadratic mixed-integer stochastic programs relative to their linearized counterparts.

To address this issue, the `runph` command supports automatic linearization of quadratic penalty terms in PH. We first observe that a linear expression results from the expansion of any quadratic penalty term involving binary variables. Consequently, the default behavior of `runph` is to linearize these terms for binary variables but to use quadratic penalty terms that involve continuous and general integer variables. The following options can be used to linearize these quadratic penalty terms:

`--linearize-nonbinary-penalty-terms`=BPTS
: Approximate the PH quadratic term for non-binary variables with a piece-wise linear function. The argument BPTS gives the number of breakpoints in the linear approximation. This defaults to 0, indicating that linearization is disabled.

`--breakpoint-strategy`=BREAKPOINT_STRATEGY
: Specify the strategy to distribute breakpoints on the [lb, ub] interval of each variable when linearizing. This defaults to 1.

To linearize a quadratic penalty term, `runph` requires that both lower and upper bounds (respectively denoted *lb* and *ub*) be specified for each variable in each scenario instance. This is most straightforwardly accomplished by specifying bounds or rules for computing bounds in each of the variable declarations appearing in the deterministic reference model. In reality, lower and upper bounds can be specified for all variables, even if trivially. If for some reason bounds are not easily specified in the deterministic reference model, the option `--bounds-cfgfile` option can be used, which functions in a fashion similar to the mechanism for setting variable-specific ρ described above. Note that if a breakpoint would be very close to a variable bound, then the breakpoint is omitted. In other words, the BPTS parameter serves as an upper bound on the number of actual breakpoints.

Three breakpoint strategies are provided. A value of 1 indicates a uniform distribution of the BPTS points between *lb* and *ub*. A value of 2 indicates a uniform distribution of the BPTS points between the current minimum and maximum values observed for the variable at the corresponding node in the scenario tree; segments between the node min/max values and *lb*/*ub* are also automatically generated. Finally, a value of 3 places the half of the BPTS breakpoints on either side of the observed variable average at the corresponding node in the scenario tree, with exponentially increasing distance from the mean.

Automatic linearization of the quadratic penalty term allows PySP to employ a wide variety of solvers within PH, and it enables a more efficient utilization of those solvers. In particular, it facilitates the use of open-source solvers – which can be critical in parallel environments where it may be impossible to procure large numbers of commercial solver licenses for concurrent use (see Section 9.7).

9.6 Progressive Hedging Extensions: Advanced Configuration

The previous sections have described ways of customizing PH that do not change the core behavior of the PH algorithm. The following sections describe more extensive and intrusive customization of the PySP PH behavior. In Section 9.6.1, we describe the interface to a PH extension that provides functionality that is often critical to achieving good performance on stochastic mixed-integer programs. The next two sections discuss other command-line options that are often used in PH practice. Finally, we discuss the programmatic facilities that PySP provides to users (typically programmers) that want to develop their own extensions.

9.6.1 Watson and Woodruff Extensions

The basic PH algorithm can converge slowly, even if appropriate values of ρ have been computed. Further, in the mixed-integer case, PH can exhibit cyclic behavior, preventing convergence. Consequently, PH implementations in practice are augmented with methods to both accelerate convergence and prevent cycling. Watson and Woodruff [60] describe and introduce many of these extensions.

The PySP implementation of PH provides these extensions in the form of a *plug-in*, i.e., a piece of code that extends the core functionality of the underlying algorithm, at well-defined points during execution. This "Watson-Woodruff" (WW) plug-in generalizes the accelerator and cycle-avoidance mechanisms described in Watson and Woodruff [60]. The Python module implementing this plug-in is named `wwextension.py`; general users do not need to understand the contents of this module.

The `runph` command provides three command-line options to control the execution of the Watson-Woodruff extensions plug-in:

`--enable-ww-extensions`
: Enable the Watson-Woodruff PH extensions plug-in. This defaults to False.

`--ww-extension-cfgfile`=WW_EXTENSION_CFGFILE
: The name of a configuration file for the Watson-Woodruff PH extensions plug-in. This defaults to "wwph.cfg".

`--ww-extension-suffixfile`=WW_EXTENSION_SUFFIXFILE
: The name of a variable suffix file for the Watson-Woodruff PH extensions plug-in. This defaults to "wwph.suffixes".

As discussed in Section 9.6.4, user-defined extensions can co-exist with the Watson-Woodruff extension.

Many aspects of the extensions are necessarily problem-specific. However, there are some general principles. Some of the main issues addressed by the Watson-Woodruff extensions are *convergence detection*, *cycle detection*, and *convergence-based sub-problem optimality thresholds*.

A detailed analysis of PH behavior on a variety of problems indicates that individual decision variables frequently converge to specific, fixed values across all scenarios in early PH iterations. Further, despite interactions among the variables, this value frequently does not change in subsequent PH iterations. Such variable-by-variable convergence behavior suggests a potentially powerful, albeit obvious, heuristic: once a particular variable has converged to an identical value across all scenarios for some number of iterations, fix it to that value. However, the strategy must be used carefully. In particular, for problems where the constraints effectively limit variables x from both sides, these methods may result in PH encountering infeasible scenario sub-problems even though the problem is ultimately feasible.

In the presence of integer variables, PH occasionally exhibits cycling behavior. 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 weight vectors w, heuristically implemented using a simple hashing scheme [61] 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 (using problem-specific, user-supplied knowledge) across scenarios in order to break the cycle.

A number of researchers have noted that it is unnecessary to solve scenario sub-problems to optimality in early PH iterations [34]. In these early iterations, the primary objective is to quickly obtain coarse estimates of the PH weight vectors, which (at least empirically) does not require optimal solutions to scenario sub-problems. Once coarse weight estimates are obtained, optimal solutions can then be pursued to tune the weight vectors in the effort to achieve convergence. Given a measure of scenario solution homogeneity (e.g., the convergence threshold $g^{(k)}$), a commonly used strategy is to set the solver *mipgap* – a termination threshold based on the difference in current lower and upper bounds – in proportion to this measure.

Fixing variables aggressively typically results in shorter run-times, but the strategy can also degrade the quality of the obtained solution. Furthermore, for some problems, aggressive fixing can result in infeasible sub-problems even though the extensive form is ultimately feasible. Many of the parameters discussed in the next subsections control fixing of variables.

9.6.1.1 Mipgap Control and Cycle Detection Parameters

The WW extension defines and exposes a number of key user-controllable parameters, each of which can be specified in the WW PH configuration file. The following parameters are supported in this configuration file:

`Iteration0MipGap` Specifies the mipgap for all PH scenario sub-problem solves in iteration 0. This defaults to 0, indicating that the solver default mipgap is used.

`InitalMipGap` Specifies the mipgap for all PH scenario sub-problem solves in iteration 1. This defaults to 0.1. A value equal to 0 indicates that the solver default mipgap is used. If a value $z > 0$ is specified, then no PH scenario sub-problem

solves will use a mipgap greater than z in iterations $k > 1$. Let $g(1)$ denote the value of the PH convergence metric (independent of the particular metric used) in iteration 1. To determine the mipgap for PH iterations $k > 1$, the value of the convergence metric $g(k)$ is used to interpolate between the `InitialMipGap` and `FinalMipGap` parameter values; the latter is discussed below. In cases where the convergence metric $g(k)$ increases relative to $g(k-1)$, the mipgap is thresholded to the value computed during iteration $k-1$.

`FinalMipGap` The target final value for all iteration k scenario sub-problem solves at PH convergence, i.e., when the value of the convergence metric $g(k)$ is indistinguishable from 0 (subject to tolerances). This defaults to 0.001. The value of this parameter must be less than or equal to the `InitialMipGap`.

`hash_hit_len_to_slam` Ignore possible cycles in the weight vector associated with a variable for which the cycle length is less than this value. Also, ignore cycles if any variables have been fixed in the previous `hash_hit_len_to_slam` PH iterations. This defaults to the number of problem scenarios $|\mathscr{S}|$. This default is often not a good choice. For many problems with numerous scenarios, fixed constant values (e.g., such as 10 or 20) typically lead to significantly better performance.

`DisableCycleDetection` A binary parameter, which defaults to `False`. If this is `True`, then cycle detection and the associated slamming logic (described below) are completely disabled. This parameter cannot be changed during PH execution, as the data structures associated with cycle detection storage and per-iteration computations are bypassed.

Users specify values for these parameters in the WW PH configuration file, which is loaded by specifying the `--ww-extension-cfgfile` command-line option for `runph`. The parameters are set directly using Python syntax, e.g., as follows:

```
self.Iteration0MipGap = 0.1
self.InitialMipGap = 0.2
self.FinalMipGap = 0.001
```

The contents of the configuration file are read by the WW extension following initialization, and the `self` identifier refers to the WW extension object that is initialized.

9.6.1.2 General Variable Fixing and Slamming Parameters

Variable fixing is often an empirically effective heuristic for accelerating PH convergence. Fixing strategies implicitly rely on strong correlation between the converged value of a variable across all scenario sub-problems in an intermediate PH iteration and the value of the variable in the final solution should no fixing be imposed. Variable fixing reduces scenario sub-problem size, accelerating solve times. However, depending on problem structure, the strategy can lead to either sub-optimal solutions (due to premature declarations of convergence) or the failure of PH to converge (due to interactions among the constraints). Consequently, careful and

problem-dependent tuning is typically required to achieve an effective fixing strategy. To facilitate such tuning, the WW PH extension allows for specification of the following parameters in the configuration file:

`Iter0FixIfConvergedAtLB` A binary parameter indicating whether discrete variables that are at their lower bound in all scenarios after iteration 0 will be fixed at that bound. This defaults to False.

`Iter0FixIfConvergedAtUB` A binary parameter that is analogous to the `Iter0FixIfConvergedAtLB` parameter, except applying to discrete variable upper bounds. This defaults to False.

`Iter0FixIfConvergedAtNB` A binary parameter that is analogous to the `Iter0FixIfConvergedAtLB` parameter, except it applies to discrete variable values that are not equal to either lower or upper bounds. This defaults to False.

`FixWhenItersConvergedAtLB` The number of consecutive PH iterations $k \geq 1$ over which discrete variables must be at their lower bound in all scenarios before they will be fixed at that bound. This defaults to 10. A value of 0 indicates that discrete variables will never be fixed at this bound.

`FixWhenItersConvergedAtUB` An integer parameter that is analogous to the `FixWhenItersConvergedAtLB` parameter, except that it applies to discrete variable upper bounds. This defaults to 10.

`FixWhenItersConvergedAtNB` An integer parameter that is analogous to the `FixWhenItersConvergedAtLB` parameter, except that it applies to discrete variable values that are not equal to either lower or upper bounds. This defaults to 10.

`FixWhenItersConvergedContinuous` The number of consecutive PH iterations $k \geq$ that continuous variables must be at the same, consistent value in all scenarios before they will be fixed at that value. This defaults to 0, indicating that continuous variables will not be fixed.

Fixing strategies at iteration 0 are typically distinct from those in subsequent iterations. For example, agreement in iteration 0 of acquisition quantities in a resource allocation problem to a value of 0 may (depending on the problem structure) indicate that no such resources are likely to be required. In general, fixing strategies for PH iterations $k \geq 1$ yield better solutions with longer delays, albeit at the expense of longer run-times; this trade-off is numerically illustrated in Watson and Woodruff [60]. Differentiation between fixing behaviors at lower bounds, upper bounds, or intermediate values are typically necessary due to the problem structure (e.g., variables being constrained from lower or upper bounds).

For many mixed-integer problems, PH can spend a disproportionately large number of iterations "fine-tuning" the values of a small number of variables in order to achieve convergence. Consequently, it is often desirable to force early agreement of these variables, even at the expense of sub-optimal final solutions. We refer to this mechanism as *slamming* [60]. Slamming is also used to break cycles detected through the mechanisms described above. The WW PH extension supports a number of configuration options to control variable slamming:

`SlamAfterIter` The PH iteration k after which variables will be slammed to force convergence. After this threshold is passed, one variable is slammed every other iteration to force convergence. This defaults to the number of scenarios $|\mathscr{S}|$.

`CanSlamToLB` A binary parameter indicating whether any discrete variable can be slammed to its lower bound. This defaults to False.

`CanSlamToUB` Analogous to the `CanSlamToUB` parameter, except that it applies to discrete variable upper bounds. This defaults to False.

`CanSlamToAnywhere` Analogous to the `CanSlamToLB` parameter, that the variable can be slammed to its current average value across scenarios. This defaults to False.

`CanSlamToMin` A binary parameter indicating whether any discrete variable can be slammed to its current minimum value across scenarios. This defaults to False.

`CanSlamToMax` Analogous to the `CanSlamToMin` parameter, except that it applies to discrete variable maximum values across scenarios. This defaults to False.

`PH_Iters_Between_Cycle_Slams` Controls the number of PH iterations to wait between variable slams imposed to break cycles. This defaults to 1, indicating a single variable will be slammed every iteration if a cycle is detected. A value of 0 indicates an unlimited number of variable slams can occur per PH iteration.

Slamming to the minimum and maximum scenario tree node values is often useful in resource allocation problems and other problems that we will call *one-sided diet problems* for historical reasons. For example it is frequently safe, with respect to feasibility, to slam a variable value to the scenario maximum in the case of one-side diet problems. In the event that multiple slamming options are available, the priority order is given as lower bound, minimum, upper bound, maximum, and anywhere.

For our purposes, we use the name one-sided diet problems to refer to group of linear problems defined mathematically to be of the form:

$$\min \sum_{j=1}^{n} c_j x_j$$

subject to:

$$\sum_{j=1}^{n} A_{i,j} x_j \geq b_i, \;\; i = 1, \ldots, m$$

and

$$x_j \geq 0, \;\; j = 1, \ldots, n$$

where the c is a vector of non-negative data of length n, A is a n by m matrix of non-negative data, and b is a vector of length m with non-negative data. The problem is to find values for the variable x, which is a vector of length n. We call it one-sided because given the non-negative data and the direction of the inequality, large

enough values of x are all feasible and furthermore, once a feasible solution has been found, it is clear that strictly increasing the value of some of the elements of x will also result in a feasible solution. For a stochastic problem, the data may vary from scenario to scenario, but when there is a feasible solution for every scenario then for any vector element it must be true that the maximum value across all scenarios must be feasible for all scenarios. This makes slamming to the maximum "safe" in some sense for such problems.

9.6.1.3 Variable-specific Fixing and Slamming Parameters

Global controls for variable fixing and slamming are generally useful, but for many problems more fine-grained control is required. For example, in one-sided diet problems, feasibility can be maintained during slamming by fixing a variable value at the maximum level observed across scenarios (assuming a minimization objective) [60]. Similarly, it is often desirable in a multi-stage stochastic program to fix variables appearing in early stages before those appearing in later stages, or to fix binary variables for siting decisions in facility location before discrete allocation variables associated with those sites.

The WW PH extension provides fine-grained, variable-specific control of both fixing and slamming using the concept of *suffixes*, which is similar to the mechanism employed by AMPL [4]. Global defaults are established using the mechanisms described in Section 9.6.1.2, while optional variable-specific overrides are specified via the suffix mechanism we now describe.

The specific suffixes recognized by the WW PH extension include the following, and have analogous (variable-specific) functionality to that provided by the parameters described in Section 9.6.1.2:

```
Iter0FixIfConvergedAtLB
Iter0FixIfConvergedAtUB
Iter0FixIfConvergedAtNB
FixWhenItersConvergedAtLB
FixWhenItersConvergedAtUB
FixWhenItersConvergedAtNB
CanSlamToLB
CanSlamToUB
CanSlamToAnywhere
CanSlamToMin
CanSlamToMax.
```

Additionally, we introduce the suffix `SlammingPriority`, which allows for prioritization of variables slammed during convergence acceleration; larger values indicate higher priority. The latter are particularly useful, for example, in the context of resource allocation problems in which early slamming of lower-cost items tends to yield lower-cost final solutions.

In general, suffixes are specified via (VARSPEC, SUFFIX, VALUE) triples, where VARSPEC indicates a variable slice (i.e., a template that matches one or more indices of a variable; if the variable is not indexed, only the variable name is specified), SUFFIX indicates the name of a suffix recognized by the WW PH extension, and VALUE indicates the quantity associated with the specified suffix. This quantity is expected to be consistent with the value expected by the suffix. If no suffix is associated with a given variable, then the global default parameter values are accessed.

Variable-specific suffixes are supplied to the WW PH extension in a file, which is specified using the the --ww-extension-suffixfile option. For example, the following suffix employs the variables used in the notional examples discussed above:

```
resource_quantity[*,Stage1] FixWhenItersConvergedAtUB 10
resource_quantity[*,Stage2] FixWhenItersConvergedAtUB 50

resource_quantity[TIRES, Stage1] SlammingPriority 50
resource_quantity[ENGINES, Stage1] SlammingPriority 10

resource_quantity[TIRES, *] CanSlamToMax True
resource_quantity[ENGINES, *] CanSlamToMax True
```

Within PySP, suffixes are trivially processed via Python's dynamic object attribute functionality. For each VARSPEC encountered, the index template (if it exists) is expanded and all matching variable value objects are identified. Then, for each variable value, a call to setattr is performed to attach the corresponding attribute/value pair to the object. The advantage of this approach is simplicity and generality: any suffix can be applied to any variable. The disadvantage is the lack of error-checking, in that suffixes unknown to the WW PH extension can inadvertently be specified, e.g., a capitalized SLAMMINGPRIORITY suffix.

9.6.2 Solving a Constrained Extensive Form

A common practice for using PH as a mixed-integer stochastic programming heuristic involves running PH for a limited number of iterations (e.g., via the --max-iterations option), fixing the values of discrete variables that appear to have converged, and then solving the significantly smaller extensive form that results [42]. The resulting compressed extensive form is generally far smaller and easier to solve than the original extensive form. This technique directly avoids issues related to the slow convergence of PH, which may be required to resolve relatively small remaining discrepancies in scenario sub-problem solutions. Any disadvantage stems from the variable fixing itself, since premature fixing of variables can lead to sub-optimal extensive form solutions.

The following options are used to write and solve the extensive form following PH termination of the runph command:

`--write-ef`
Upon termination, write the extensive form of the model. Disabled by default.
`--solve-ef`
Following write of the extensive form model, solve the extensive form and display the resulting solution. Disabled by default.
`--ef-output-file`=EF_OUTPUT_FILE
The name of the extensive form output file. Defaults to "efout.lp".

When writing the extensive form, all variables whose value is currently fixed in any scenario sub-problem are automatically preprocessed into constant terms in any referencing constraints or the objective. As noted in Section 9.4, PySP only supports output of the CPLEX LP file format. Solver selection is controlled with the `--solver` keyword, and it used the same way as when solving scenario sub-problems. However, the `runph` command does provide interfaces for solver options (including mipgap) that are specific to the extensive form solve.

9.6.3 Alternative Convergence Criteria

The implementation of PH in PySP supports a variety of convergence metrics, enabled via the following `runph` command-line options:

`--enable-termdiff-convergence`
Terminate PH based on the `termdiff` convergence metric, which is defined as the unscaled sum of differences between variable values and the mean (see Section 9.5.1). This Defaults to `True`.
`--enable-normalized-termdiff-convergence`
Terminate PH based on the normalized `termdiff` convergence metric. Each term in the `termdiff` sum is normalized by the average variable value. This defaults to `False`.
`--enable-free-discrete-count-convergence`
Terminate PH based on the free discrete variable count convergence metric. This defaults to `False`.
`--free-discrete-count-threshold`=
FREE_DISCRETE_COUNT_THRESHOLD
PH will terminate once the number of free discrete variables drops below this threshold.

The termination criterion associated with the free discrete variable count is particularly useful when deployed in in conjunction with the capability to solve restricted extensive forms described in Section 9.6.2.

9.6.4 User-Defined Extensions

The Watson-Woodruff PH extensions described in Section 9.6.1 rely on a simple, general call-back framework that allows user-defined extensions to enhance the functionality of the core PH algorithm. Most modelers and typical PySP users would not make use of this feature, programmers and algorithm developers can easily leverage this capability. The interface for user-defined PH extensions is defined in a PySP read-only file called `phextension.py`. This file defines an interface class that defines the points at which `runph` temporarily transfers control to the user-defined extensions:

```
class IPHExtension(Interface):

   def post_ph_initialization(self, ph):
      """ Called after PH initialization."""
      pass

   def post_iteration_0_solves(self, ph):
      """ Called after the iteration 0 solves."""
      pass

   def post_iteration_0(self, ph):
      """ Called after the iteration 0 solves, averages
         computation, and weight update."""
      pass

   def post_iteration_k_solves(self, ph):
      """ Called after the iteration k solves."""
      pass

   def post_iteration_k(self, ph):
      """ Called after the iteration k solves, averages
         computation, and weight update."""
      pass

   def post_ph_execution(self, ph):
      """ Called after PH has terminated."""
      pass
```

User-defined extensions are defined with a Python class that inherits from the class `SingletonPlugin` and implements the PH extension interface shown above:

```
from pyutilib.component.core import *
from coopr.pysp import phextension

class examplephextension(SingletonPlugin):

  implements(phextension.IPHExtension)

  def post_instance_creation(self, ph):
    print ``Done creating PH scenario instances!''
```

A more complete example of PH extensions is supplied with PySP, in the Python file `testphextension.py`. All Coopr user plug-ins are derived from a `SingletonPlugin` base class. The word "Singleton" indicates that that there cannot be multiple instances of each type of user-defined extension.

Each extension point (i.e., call-back) in the user-defined extension is supplied the PH object, which includes the current state of the scenario tree, reference instance, all scenario instances, PH weights, etc. User code can then be developed to modify the state of PH (e.g., current solver options) or variable attributes (e.g., fixing as in the case of the Watson-Woodruff extension).

To use a customized extension with `runph`, the user invokes the command-line option
`--user-defined-extension=EXTENSIONFILE`.
Here, `EXTENSIONFILE` is the Python module name, which is assumed to be either in the current directory or in some directory specified via the `PYTHONPATH` environment variable. Finally, note that both a user-defined extension and the Watson-Woodruff PH extension can co-exist. However, the Watson-Woodruff extension will be invoked before any user-defined extension.

9.7 Solving PH Scenario Sub-Problems in Parallel

PySP supports the distributed execution of the optimization solve from both the `runef` and `runph` commands. A simple client-server paradigm is supported, which leverages the general distributed solver capabilities that are provided in the Coopr library [33]. Coopr integrates the third-party, open-source Pyro (Python Remote Objects) package [50] to manage the communication between machines, and Coopr includes the mechanisms and scripts by which name servers (used to locate distributed objects) and solver servers (daemons capable of solving MIPs, for example) are initialized and interact. Here, we simply describe the use of a distributed set of solver servers in the context of PySP.

Both the `runef` and `runph` commands are implemented such that all requests for the solution of (mixed-integer) linear problems are mediated by a *solver manager*. The default solver manager in both commands is a serial solver manager, which executes all solves locally. Alternatively, a user can invoke a remote solver manager by specifying the command-line option `--solver-manager=pyro`. The Pyro solver manager identifies available remote solver daemons, serializes the relevant Pyomo model instance for communication, and initiates a solve request with the daemon. After the daemon has solved the instance, the solution is returned to the Pyro solver manager, which then transfers the solution to the invoking command.

We will refer to a single master computer and multiple slave computers in this discussion, but the master computing processes can be on a processor that also runs a slave process. The following commands need to be executed to perform distributed optimization:

1. On the master: `coopr_ns`
2. On the master: `dispatch_srvr`
3. On each slave: `pyro_mip_server`
4. On the master: `runph ... --solver-manager=pyro ...`

Note that the command argument `solver-manger` has a dash in the middle, while the commands `coopr_ns`, `dispatch_srvr` and `pyro_mip_server` have underscores. The first three commands launch processes that have no internal mechanism for termination; i.e., they will be terminated only if they crash or if they are killed by an external process. It is common to launch these processes with output redirection, such as `coopr_ns >& cooprns.log`. The `runph` command is a executed with the usual arguments plus the specification that sub-problem solves should be directed to the Pyro solver manager.

Python's `pickle` module provides facilities for object serialization that can be applied to very complex objects, including Pyomo model instances. The accessibility of remote solvers within PySP's PH implementation immediately confers the benefit of trivial parallelization of scenario sub-problem solves. In the case of commercial solvers, all available licenses can be leveraged. In the open-source case, cluster solutions can be deployed in a straightforward manner.

Parallelism in PySP most strongly benefits stochastic mixed-integer program solves, in which the difficulty of scenario sub-problems masks the overhead associated with object serialization and client-server communication. At the same time, parallel efficiency necessarily falls as the number of scenarios increases, due to high variability in mixed-integer solve times and the presence of barrier synchronization points in PH (after Step 4 in the pseudocode introduced in Section 9.5). However, high-throughput computing is a more important driver for stochastic programming applications than parallel efficiency.

When solving the extensive form, remote solves serve a different purpose: to facilitate access to a central computing resources, with associated solver licenses. For example, our primary workstation for developing PySP is a 16-core workstation with a large amount of RAM (96GB). All commercial solver licenses are localized to this workstation, which in turn exposes parallelism enabled by a multi-threaded solver.

9.8 Discussion

The `runef` command relies on significant functionality from the Coopr Python optimization library in which PySP is embedded. This includes a wide range of solver interfaces (both commercial and open-source), problem writers, solution readers, and distributed solvers. Although this functionality is also leveraged by the `pyomo` command, PySP illustrates that extension packages can generalize the simple optimization process that is supported within Pyomo.

A variety of advanced Python capabilities are leveraged to support the WW extension and user-defined extensions. The file contents for WW extension configuration files are directly executed via the Python `execfile` command. While powerful and simplistic, this approach to initialization is potentially dangerous, as any attribute of the WW extension object is subject to manipulation.

Similarly, the `SingletonPlugin` class used to define user-defined extensions is an component of the PyUtilib Component Architecture (PCA) [32]. The PCA is used throughout Coopr to manage the definition of extension points that support customization. The PCA is also supports the setup of software factories that dynamically create Python interfaces for optimizers and external utilities that are integrated into Coopr.

Support for Progressive Hedging will continue to evolve in the form of additions to `wwphextensions.py` and the development of additional plug-ins. At present, there is no "out-of-the-box" solution for PH that is viable for instances that are not small. The core implementation of PH is stable and reasonably efficient so the research focus is on ways to accelerate convergence. The command `phsolverserver` is being developed to replace `pyro_mip_server`. It distributes more of the data and work to the slaves.

9.A Examples

9.A.1 Farmer Model

`ReferenceModel.py`: The deterministic reference Pyomo model for Birge and Louveaux's farmer problem.

```
# Farmer: rent out version has a singleton root node var
# note: this will minimize
#
# Imports
#

from coopr.pyomo import *

#
# Model
#

model = AbstractModel()

#
# Parameters
#

model.CROPS = Set()

model.TOTAL_ACREAGE = Param(within=PositiveReals)
```

```
model.PriceQuota = Param(model.CROPS, within=PositiveReals)

model.SubQuotaSellingPrice = Param(model.CROPS, \
    within=PositiveReals)

def super_quota_selling_price_validate (model, value, i):
   return model.SubQuotaSellingPrice[i] >= \
       model.SuperQuotaSellingPrice[i]

model.SuperQuotaSellingPrice = Param(model.CROPS, \
    validate=super_quota_selling_price_validate)

model.CattleFeedRequirement = Param(model.CROPS, \
    within=NonNegativeReals)

model.PurchasePrice = Param(model.CROPS, \
    within=PositiveReals)

model.PlantingCostPerAcre = Param(model.CROPS, \
    within=PositiveReals)

model.Yield = Param(model.CROPS, within=NonNegativeReals)

#
# Variables
#

model.DevotedAcreage = Var(model.CROPS, bounds=(0.0, \
    model.TOTAL_ACREAGE))

model.QuantitySubQuotaSold = Var(model.CROPS, bounds=(0.0, \
    None))
model.QuantitySuperQuotaSold = Var(model.CROPS, \
    bounds=(0.0, None))

model.QuantityPurchased = Var(model.CROPS, bounds=(0.0, \
    None))

model.FirstStageCost = Var()
model.SecondStageCost = Var()

#
# Constraints
#

def ConstrainTotalAcreage_rule(model):
   return summation(model.DevotedAcreage) <= \
       model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = Constraint()

def EnforceCattleFeedRequirement_rule(model, i):
   return model.CattleFeedRequirement[i] <= (model.Yield[i]\
```

```
         * model.DevotedAcreage[i]) + \
        model.QuantityPurchased[i] - \
        model.QuantitySubQuotaSold[i] - \
        model.QuantitySuperQuotaSold[i]

model.EnforceCattleFeedRequirement = Constraint(model.CROPS)

def LimitAmountSold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + \
        model.QuantitySuperQuotaSold[i] - (model.Yield[i] * \
        model.DevotedAcreage[i]) <= 0.0

model.LimitAmountSold = Constraint(model.CROPS)

def EnforceQuotas_rule(model, i):
    return (0.0, model.QuantitySubQuotaSold[i], \
        model.PriceQuota[i])

model.EnforceQuotas = Constraint(model.CROPS)

#
# Stage-specific cost computations
#

def ComputeFirstStageCost_rule(model):
    return model.FirstStageCost - \
        summation(model.PlantingCostPerAcre, \
        model.DevotedAcreage) == 0.0

model.ComputeFirstStageCost = Constraint()

def ComputeSecondStageCost_rule(model):
    expr = summation(model.PurchasePrice, \
        model.QuantityPurchased)
    expr -= summation(model.SubQuotaSellingPrice, \
        model.QuantitySubQuotaSold)
    expr -= summation(model.SuperQuotaSellingPrice, \
        model.QuantitySuperQuotaSold)
    return (model.SecondStageCost - expr) == 0.0

model.ComputeSecondStageCost = Constraint()

#
# Objective
#

def Total_Cost_Objective_rule(model):
    return model.FirstStageCost + model.SecondStageCost

model.Total_Cost_Objective = Objective(sense=minimize)
```

9.A.2 Farmer Data File

`ReferenceModel.dat`: The Pyomo reference model data for Birge and Louveaux's farmer problem.

```
set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota :=
         WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice :=
         WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice :=
         WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement :=
         WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no need, as cattle don't eat them)
param PurchasePrice :=
         WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre :=
         WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 3.0 CORN 3.6 SUGAR_BEETS 24 ;
```

9.A.3 Scenario Tree Model

`ScenarioStructure.py`: The PySP model for specifying the structure of the scenario tree.

```
from coopr.pyomo import *

model = AbstractModel()

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

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

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

model.StageVariables = Set(model.Stages)
model.StageCostVariable = Param(model.Stages)

model.ScenarioBasedData = Param(within=Boolean, default=True)
```

9.A.4 Scenario Data Command File

`ScenarioStructure.dat`: The data command file that defines the scenario tree for the farmer problem.

```
# IMPORTANT - THE STAGES ARE ASSUMED TO BE IN TIME-ORDER.

set Stages := FirstStage SecondStage ;

set Nodes := RootNode
          BelowAverageNode
          AverageNode
          AboveAverageNode ;

param NodeStage := RootNode FirstStage
               BelowAverageNode SecondStage
               AverageNode SecondStage
               AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                    AverageNode
                    AboveAverageNode ;

param ConditionalProbability := RootNode 1.0
                        BelowAverageNode 0.33333333
                        AverageNode 0.33333334
                        AboveAverageNode 0.33333333 ;

set Scenarios := BelowAverageScenario
             AverageScenario
             AboveAverageScenario ;

param ScenarioLeafNode :=
               BelowAverageScenario BelowAverageNode
               AverageScenario AverageNode
               AboveAverageScenario AboveAverageNode ;

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

param StageCostVariable := FirstStage FirstStageCost
                    SecondStage SecondStageCost ;
```

Chapter 10
Scripting and Algorithm Development

Abstract This chapter illustrates the use of Python scripts to interrogate Pyomo models and customize the optimization process. We present a variety of simple scripts for common analysis activities. We also discuss several advanced scripting topics using examples such as hybrid optimization and Benders decomposition. Together, these scripting examples illustrate how Pyomo users can go beyond the simple use of the `pyomo` command to formulate and analyze optimization models.

10.1 Introduction

In previous chapters, we have described how a generic optimization process can be executed with Pyomo to construct a model instance from an abstract model and a data file, and to subsequently solve the instance using a solver. The `pyomo` command can apply this generic optimization process to a specific model, 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 indices. 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 [28].

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, for example, objective functions or constraints in a fully general way. This capability is a fundamental tool for general algorithm

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_10,

development and extension [43]. 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.

The use of Python provides tremendous flexibility for both the Pyomo modeling environment and the Coopr framework. With some AMLs, a new scripting language is defined that is unique to the AML, and the developers of the package produce a parser for the new language. This separates the user from the underlying code of the framework itself. With Pyomo, Python is used for both the overall framework and the modeling environment. This provides the user with complete control over the entire model construction and solution workflow.

In this chapter, we introduce a variety of case studies that highlight the relative ease of both scripting and algorithm development in Coopr. Section 10.2 introduces the basics and shows how a user-defined script can be created to drive the solution process instead of the `pyomo` command. Section 10.3 includes a set of examples that reflect common scripting tasks. Section 10.4 discusses the implementation of a hybrid MIP-NLP optimization algorithm. Finally, Section 10.5 describes a Benders decomposition algorithm for a simple production planning problem. These last two sections illustrate the use of Pyomo and Coopr to solve optimization models that require some degree of algorithmic customization.

10.2 Scripting Basics

10.2.1 A Canonical Optimization Script

Section 2.5.2 introduced a simple Python script to perform optimization of a Pyomo model instance, which we describe further in this section. Suppose that the model in Example 8.A.4 is stored in the file `DiseaseEstimation.py` with associated Pyomo data file `DiseaseEstimation.dat`. The script in Example 10.A.1 can then be used to import this model, display the created model, create a solver interface, perform optimization, and display the results. Note that this script does not rely on the `coopr.pyomo` package directly. Pyomo is only used to create an optimization model. Subsequent optimization and analysis of this model is handled in a generic manner with other Coopr packages.

The script in Example 10.A.1 calls IPOPT, which is an open source solver for nonlinear programs. Other solvers could have been invoked if the `SolverFactory` function is passed a different solver name. Section C.2 describes the different solver interfaces that are supported by Coopr. Some solver interfaces can be used to execute more than one type of solver. For example, Acro's `coliny` command can execute a wide variety of optimizers [1]. The following command creates an `opt` object that interfaces with the `coliny` command and executes the pattern search

optimizer, `sco:ps`:

```
opt = SolverFactory('coliny', solver='sco:ps')
print opt.options
```

Example 10.A.1 can be generalized to allow the specification of multiple data sources that are used to create the model instance. Example 10.A.2 constructs a `ModelData` object, which is configured to process both the `DiseasePop.dat` and `DiseaseEstimation.dat` files. After the `ModelData` object reads these files, it is used to create the model instance. The `ModelData` object reads in data sequentially, and data values that are read in override the values that were specified in previous data sources. In this example, the `DiseasePop.dat` file specifies a different value for the `P_POP` parameter, which overrides the value provided in `DiseaseEstimation.dat`.

More advanced interaction with the solver is often needed in applications to process solver options, manage errors, and concisely summarize solver results. For example, the script in Example 10.A.1 does not output solver information to the console. Example 10.A.3 extends the script in Example 10.A.1 to include solver options, and to directly manage the execution with a `SolverManager` object. This requires the use of the `SolverManagerFactory` to create a serial `SolverManager` object; this is the default solver manager used by `coopr.opt`.

The previous examples make use of `opt.solve` or `solvermanager.solve`. However, because of the flexibility of Python, we can work with the Coopr/Pyomo framework at a deeper level. Example 10.A.4 essentially mimics the function calls used by the `pyomo` command. The `pyomo` command executes functions in the `pyomo.scripting.util` module, and these functions can be directly executed within a user script. This can be useful since most of the higher level command-line options associated with the `pyomo` driver are available.

Note that the functions in `scripting.util` assume that the `Options` object is populated with default attributes. Therefore, we create default values for some of the options manually in the script. The complete listing for this example is shown below. Comparing Examples 10.A.1 and 10.A.4, we see the two extremes; one which includes the minimum number of commands necessary to perform the solve, and one which provides a richer interaction with the Coopr framework, complete with options processing.

10.3 Common Scripting Tasks

10.3.1 Printing and Comparing Variable Values

A key feature of scripting is the ability to manage multiple optimization models that are independently optimized. This capability is essential to support the more sophisticated scripts that are described later in this chapter. Example 10.A.5 is a script that constructs two model instances by importing modules `concrete1.py`

and `concrete2.py` (shown in Examples 2.A.1 and 2.A.2 respectively). Python's import command executes the commands in these modules, so the entire process of model construction is reduced to a single line in this script. Also, note that Python's import syntax is flexible enough to allow the user to import objects from a module with a different name, which avoids naming conflicts in this script.

After performing optimization, this script loads the optimization results back into the model instance:

```
instance1 = model1.create()
results1 = opt.solve(instance1)
instance1.load(results1)
```

This provides a convenient way to analyze the optimization results. Although the `results1` object can be directly queried for the results data, this requires learning the interface for a new Python object; the interface for variable and constraint objects will be more familiar to beginning Python modelers.

This script shows different ways of printing and comparing variable data. The first set of print commands illustrates how the values of variable data can be accessed:

```
from coopr.pyomo import value

print "x_2 value:", instance1.x_2.value
print "x_2 value:", value(instance1.x_2)
print "x[2] value:", instance2.x[2].value
print "x[2] value:", value(instance2.x[2])
```

The first two print statements display the value of variable `x_2` in `instance1`. The first print statement displays the `value` attribute of the `x_2` variable, and the second print statement applies the `value()` function to retrieve the value of this variable. Similarly, the next two print statements display the value of variable `x[2]` in `instance2`.

These print statements reflect some fundamental features of Pyomo's Python-based implementation. First, the variable objects have attributes like `value` that can be directly accessed. Second, the `x_2` and `x[2]` objects are both component value objects. Although the `x` variable object in `instance2` does not have a value, `x[2]` is a component value object which does have a value. Since `x_2` is a singleton, it is an object that combines the properties of both component and component value objects.

The second set of print statements are included to illustrate how *not* to access variable values:

```
print "x_2 object:", instance1.x_2
print "x[2] object:", instance2.x[2]
```

These statements print the objects `x_2` and `x[2]`. However, this does not result in the variable values being accessed. Instead, the name of the component value objects is printed: "x_2" and "x[2]". This reflects a more fundamental element of Pyomo's design: the component values must be explicitly accessed. This design is motivated by constraints on how component values are set.

Example 10.A.5 ends with two comparisons of the variable values in `instance1` and `instance2`. Since these are mathematically equivalent, we expect them to have identical solution values. The point of these comparisons is to illustrate how they are evaluated differently within Python. In the first comparison, the values of `x_2` and `x[2]` are accessed and then compared:

```
if value(instance1.x_2) == value(instance2.x[2]):
```

The comparison operator is operating on two floating point values, so it is executed without further computational steps. In the second comparison, the `x_2` and `x[2]` component value objects are passed to the comparison operator:

```
if instance1.x_2 == instance2.x[2]:
```

Pyomo overloads the operation of comparison operators for component value objects, and these overloaded operators construct Pyomo expression objects. This design supports the construction of constraint expressions, but in this context the result is that the comparison operator creates an expression that is immediately evaluated by Python. Although this has the same result, the second comparison is clearly more expensive. This observation clearly has implications for the construction of constraint and objective expressions, which often require conditional logic that could involve the silent creation of Pyomo expression objects.

10.3.2 Looping Over Variables

The previous example illustrates how to access variables by name. Example 10.A.6 illustrates several ways of iterating over variables in a generic manner, which is useful in many scripting contexts.

The following example loops over all indices of a particular variable name:

```
for index in instance.x:
   print instance.x[index], instance.x[index].value
```

In this example, Python is treating the variable `x` as an iterator over the set of indices of `x`. The print statement displays the name of each component value object along with its value.

The next example illustrates how to loop over all active variables in a model, and then loop over their indices:

```
from coopr.pyomo import Var
for var in instance.active_components(Var):
   print "Variable",var
   obj = getattr(instance, var)
   for index in obj:
      print obj[index], obj[index].value
```

The outer loop iterates through the names of all active `Var` components. By default, Pyomo components are active, but they can be deactivated with the `deactivate()` method. The `active_components()` method returns an iterator for the active

components for the specified Pyomo component. Python's `getattr()` function is used to get the `Var` object that corresponds to the variable name, and the inner loop iterates over the indices of that variable to print the component value object name and value.

10.3.3 Initializing and Fixing Variables

As discussed in Section 8.3.2, effective initialization of variables is often critical when optimizing nonlinear models. In Section 8.4.1 we showed that different solutions could be obtained with different initial starting points for the variables. Example 10.A.7 shows how both initial points are tested in a single script.

Sometimes, it can be helpful to perform the minimization at several different starting points and explore the solution space more fully. For example, we may want to solve the problem for a large number of random initial points, or define a grid and solve the problem at each of these initial points. Example 10.A.8 explores a grid of starting points and keeps a list of all of the unique solutions that are found.

This script illustrates some of the powerful extensibility that is possible using the object-oriented features of Python. The script declares a new class with `class Solution:` that is intended to store a candidate solution and the initial starting point that was used. We have added two methods to the class, one to check if a candidate solution is different using a numerical tolerance, and one to print out the candidate solution and the corresponding initial values. The ability to define new classes on-the-fly inside of Python can be very powerful. The script then solves all the problems with different initial values, and keeps a list of all the unique solutions that are found. These unique solutions are printed to the screen.

NOTE: While the script in Example 10.A.8 is useful for small problems, note that the number of points in the grid grows exponentially with the number of variables in the problem. This is not a good strategy for large problems, and approaches like this can not replace effective initialization tailored to the problem at hand.

It is often useful to fix variables at known values to assess how other variables change. Consider Example 8.A.2, which defines a simple multimodal function of two variables. Example 10.A.9 illustrates how variable fixing is used to analyze this model by iteratively fixing and unfixing variables.

In the initial optimization of this model, the initial values for the variables are defined with a simple syntax:

```
instance.y = 3.5
instance.x = 3.5
```

Note that the use of the `value` attribute is not necessary Pyomo overloads Python's logic for setting attributes in the `instance` object. Thus, Pyomo can determine that the components `x` and `y` are already attributes of `instance`, in which case the default behavior is to set the value of the existing component.

Fixing one more more variables requires two steps:

```
instance.y.fixed = True
instance.preprocess()
```

First, the `fixed` attribute must be set to `True` for the variable, which indicates that this is a constant value in the model. Second, the `preprocess()` method needs to be executed for the model instance to reprocess the objective and constraint expressions. This eliminates the fixed variables from the expression representations by treating them as constants.

10.3.4 Adding and Dropping Constraints

There are often times when you may want to solve a sequence of models with slightly different constraints. Pyomo offers a simple mechanism to enable and disable various model components. In this example, we extend the disease estimation problem discussed in Section 8.4.3. The disease model contains a trilinear term $\beta I_{i-1} S_{i-1}/N$ that may induce multiple local minima to the problem. To obtain a good initialization, we would like to replace I_i with the measured value. That is, we would like to replace,

$$I_i = \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N} \;\; \forall \;\; i \in SI \setminus 1$$

with,

$$I_i = \frac{\beta C_{i-1}^{\alpha} S_{i-1}}{N} \;\; \forall \;\; i \in SI \setminus 1,$$

where C_i is the measured number of reported cases in time interval i. Since the C_i values are known inputs, this substitution reduces the nonlinearity in the problem (the trilinear term becomes a bilinear term). Our strategy will be to solve the problem first with the bilinear expression and use this solution as an initialization for the actual problem with the trilinear expression. Example 10.A.10 illustrates the use of the `activate` and `deactivate` methods to execute this process.

10.3.5 Sharing Solution Results Across Multiple Models

The previous section illustrated how a model can be easily altered. In that example, our goal was to solve a simpler problem to provide initialization for the more difficult problem, and we showed how different constraints could be added or dropped to allow for this. In this section, we illustrate another approach, where we define

two completely different models, yet load the results from solving one instance into another instance to provide initialization.

Results are loaded based on variable name. Therefore, any variables with the same name will have new values loaded from the results object. In this example, we will use the same disease model we described Section 10.3.4. Instead of adding or dropping the constraints, we will create two completely different models (albeit they will only be different in one constraint). Example 10.A.11 shows the model with the easy constraint. The more difficult model is identical except that the `_EasyInfDynamics` constraint is replaced with,

```
def _InfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.I[i-1]**model.alpha)/model.P_POP
   return Constraint.Skip
model.InfDynamics = Constraint(model.S_SI, \
    rule=_InfDynamics)
```

Example 10.A.11 also includes a driver script that shares results between the two models. Note that we make use of a special form of the Python `import` command to import the easy model and the hard model under different names. We use our script to solve the easy problem and then import those results into the more difficult model. We solve the more difficult model with this new initialization.

10.3.6 Plotting Data with Matplotlib

One of the primary benefits of using Python for the Pyomo framework is the availability of many Python packages that can be used within Pyomo. For example, Python's plotting packages are an effective way to summarize and display results. Example 10.A.12 extends the script used to analyze the disease estimation problem described in Section 8.4.3. Here, once the estimation is complete, we make use of the package `matplotlib` to display a plot of the estimated data against the measured values.

The script in Example 10.A.12 is derived from the Pyomo model in Example 8.A.4. To solve the problem and generate the plot, the following scripting lines are added below the model definition:

```
instance = model.create('DiseaseEstimation.dat');

options = Options()
options.solver = 'ipopt'
options.quiet = True

# solve the problem
results, opt = scripting.util.apply_optimizer(options,
                                              instance)

# load the results to plot the solution
instance.load(results)
est_incidence = []
act_incidence = []
for i in instance.S_SI:
   est_incidence.append(value(instance.I[i]))
   act_incidence.append(value(instance.P_REP_CASES[i]))

import matplotlib.pyplot as plt

plt.plot(est_incidence)
plt.plot(act_incidence, 'o')
plt.show()
```

The plot shown in Figure 10.1 should appear after executing the file with Python.

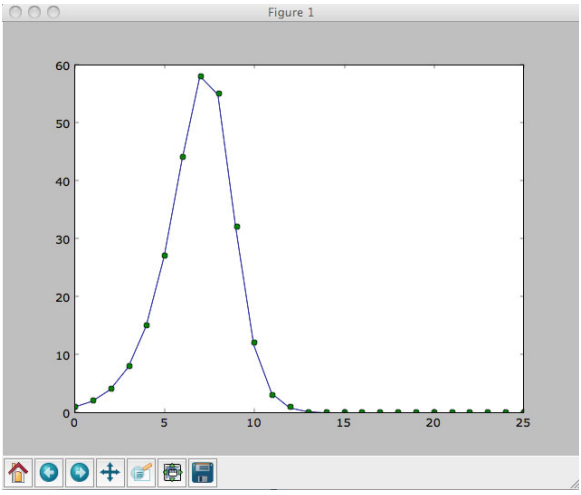

Fig. 10.1 A `matplotlib` example showing the estimated data and the measured data for the disease example.

10.3.7 Generating Different Models with a Function

In this example, we further show the power of Python within the Pyomo framework. Here, we define a Python function to create our model. The function takes several inputs that serve to define the specific form of the model.

Specifically, we will solve a feasibility problem and seek to find all the feasible solutions. We will solve the problem once, identify a feasible solution, then add an integer cut to remove this solution from the list of possible solutions and solve the problem again.

In this problem we will seek solutions to the popular Sudoku puzzle. A typical Sudoku puzzle is shown in Figure 10.2.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Fig. 10.2 An example of a Sudoku puzzle prior to solving

In this puzzle, one must fill in the missing cells with the numbers 1 through 9. Each row must have only one occurrence of each number. Likewise, each column must only have one occurence of each number. Finally, each of the nine subsquares must also only have one occurrence of each number. We will define the sets *ROWS*, *COLS*, and *VALUES* (all of which contain the integers 1 through 9. We will then define a binary variable $y[r,c,v]$ to indicate which number is in each of the cells. If $y[r,c,v] = 1$, then this implies that the value v has been selected for the cell identified by row r and column c.

Using this notation, it is relatively straightforward to define the constraints that restrict the allowable numbers in each row and column as,

$$\sum_{c \in COLS} y[r,c,v] = 1 \quad \forall\, r \in ROWS,\ v \in VALUES$$
$$\sum_{r \in ROWS} y[r,c,v] = 1 \quad \forall\, c \in COLS,\ v \in VALUES$$

The Pyomo code for these constraints is:

```
# exactly one number in each row
def _RowCon(model, i, v):
   return sum(model.y[i,c,v] for c in xrange(1,10)) == 1
model.RowCon = Constraint(model.ROWS, model.VALUES, \
    rule=_RowCon)

# exactly one nubmer in each column
def _ColCon(model, j, v):
   return sum(model.y[r,j,v] for r in xrange(1,10)) == 1
model.ColCon = Constraint(model.COLS, model.VALUES, \
    rule=_ColCon)
```

Defining the constraint that restricts the number for the subsquares is a little more difficult. To make the definition easier, we define a set with an index for each of the subsquares. Then, we define a list of tuples that describes the map from each of the subsquares to the list of corresponding indices. This list, along with the corresponding subsquares constraint, is defined in the complete code listing for this example at the end of this section. The desired constraint for the subsquares is given by,

$$\sum_{(r,c) \in ss_{map}[i]} y[r,c,v] = 1 \quad \forall\, i \in SUBSQUARES$$

The Pyomo code for this constraint is:

```
# exactly one number in each subsquare
def _SubSqCon(model, b, v):
   return sum(model.y[t[0],t[1],v] for t in \
       subsq_to_row_col[b]) == 1
model.SubSqCon = Constraint(model.SUBSQUARES, \
    model.VALUES, rule=_SubSqCon)
```

The last key constraint for the sudoku problem is to make sure that there is only one value allowed per cell. The constraint is given by,

$$\sum_{v \in VALUES} y[r,c,v] = 1 \quad \forall\, r \in ROWS,\ c \in COLS$$

The Pyomo code for this constraint is:

```
# exactly one number in each cell
def _ValueCon(model, i, j):
    return sum(model.y[i, j, v] for v in xrange(1,10)) ==\
        1
model.ValueCon = Constraint(model.ROWS, model.COLS, \
    rule=_ValueCon)
```

When designing Sudoku puzzles, two features may change frequently: the initial board layout and the number of integer cuts to remove previously seen solutions. One way to handle this variety of potential inputs is to define a function to create the model. This function will take, as inputs the desired integer cuts and the board layout.

We define the integer cuts using two lists. The list `cut_on` will have one entry for each cut we would like to add. This entry will itself be the list of tuples of the (r,c,v) values where the binary variable was 1 for the previous solution. The list `cut_off` will also have one entry for each cut. This entry will be the list of types of (r,c,v) values where the binary variable was 0 in the solution we want to cut. We will a single integer cut for every list of tuples n `cut_on`. Note: `cut_on` must have the same number of entries as `cut_off`. For example, given the following two solutions,

$$y[1,1,1] = 0,\ y[1,1,2] = 1,\ \cdots,\quad y[1,2,1] = 0, y[1,2,2] = 0,\quad y[1,2,3] = 1,\ \cdots$$
$$y[1,1,1] = 1,\ y[1,1,2] = 0,\ \cdots,\quad y[1,2,1] = 0, y[1,2,2] = 1,\quad y[1,2,3] = 0,\ \cdots$$

then,

```
cut_on[1]=[ (1,1,2), (1,2,3), ···
cut_off[1]=[ (1,1,1), (1,2,1), (1,2,2), ···
cut_on[2]=[ (1,1,1), (1,2,2), ···
cut_off[2]=[ (1,1,2), (1,2,1), (1,2,3), ···
```

Example 10.A.13 includes a Pyomo script that defines a function that generates a Pyomo model for sudoku. Because of the function definition and the structure of this script, it is not possible to run this using the `pyomo` command. Instead, Example 10.A.13 includes another script that drives the optimization process. This script defines the candidate board, and iteratively solves the sudoku problems by adding integer cuts until the problem is no longer feasible. Infeasibility is assumed when the solver status is no longer returned as `optimal`. Running this script provides all possible solutions as the output. In this example, there is only one solution to the candidate sudoku puzzle, as shown in Figure 10.3.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fig. 10.3 Solved sudoku puzzle.

10.4 Hybrid Optimization

Hybrid methods may be required to solve particularly difficult real-world optimization problems. Implementation of hybrid methods typically requires non-trivial scripting because hybrids generally require a custom optimization workflow process. To illustrate this point, we describe a hybrid optimization method to solve a parameter estimation problem arising in the context of a model for childhood infectious disease transmission.

This parameter estimation model is a difficult nonconvex, nonlinear program for which no efficient solution algorithm exists to find global optima. To facilitate 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.

Example 10.A.14 provides a Pyomo script that implements the hybrid search. Note that some initialization code is omitted from this example for clarity. In this example, user-defined Python functions are used to organize and modularize the optimization workflow. For example, the optimization model in this application is constructed via a function, `disease_mdl`. Example 10.A.14 includes a sketch of this function.

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. Also note the use of the auxiliary function, `linearize_exp`, which is used to construct specific classes of constraints in a modular manner.

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, 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 is relatively compact. Including the code for the model definition, there are a total of approximately 650 lines of Python code, including white-space.

10.5 Benders Decomposition

To further illustrate advanced scripting in Pyomo, we next consider a Pyomo example that performs Benders decomposition to solve a production planning problem formulated as a stochastic linear program. This example illustrates many of the powerful capabilities of Pyomo and Coopr, including dynamic construction of model variables and constraints, as well as parallelization of the solution of subproblems. This example derived from an AMPL example for this problem, so we can also make comparisons between AMPL and Pyomo. 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.

Given a number of product types and associated production rates, production limits, and inventory holding costs, the objective for production planning 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 first week. Although this particular problem could be solved using PySP as described in Chapter 9, we develop a Benders decomposition algorithm to illustrate Pyomo's scripting capabilities. A general discussion of how two-stage stochastic linear programs can be solved via Benders decomposition is provided by Bertsimas and Tsitsiklis [10].

The full set of files for this example are available in the Coopr distribution, in the Coopr installation directory examples/pyomo/benders. Pyomo models for the mas-

ter and sub-problems 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 definitions.

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

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 lines of the `runbenders` script involve constructing 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 = [] # a Python list
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"))
```

This code loads the abstract models defined in `master.py` and `subproblem.py` files, using the PyUtilib `import_file` function. The `runbenders` script accesses the corresponding `model` objects in the respectively Python modules. Given an abstract model, a model instance can be created by invoking its `create` method supplied with an argument specifying a data file. For reasons that will become apparent, the second stage sub-problems are gathered into a Python list.

Next, `runbenders` creates the necessary solver and solver manager plug-ins:

```
# initialize the solver / 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
```

We use CPLEX to solve model instances because it provides the variable and constraint suffixes needed for the Bender's procedure (e.g., reduced-costs). In Coopr,

the solver manager is responsible for coordinating the execution of any solver plugins. The code fragment above illustrates the use of two alternative solver managers, with parallel execution disabled (it is commented-out).

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 using the following lines of code:

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

model.Cut_Defn = Constraint(model.CUTS, noruleinit=True)
```

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 when defining Benders cuts [10]):

```
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 when the algorithm begins, 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.

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, \
    suffixes=['rc','dual'])
```

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 xsequence(len(sub_insts)):
  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].getattrvalue("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. This code makes use of standard variable "suffix" information that is available from solvers that provide it (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 previous examples. 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 rest 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. Addi-

tionally, this script supports parallelization in a generic and straightforward manner.

10.6 Discussion

The scripting examples in this chapter leverage the design of Pyomo and Coopr in a deeper manner than most of the earlier examples in this book. Thus, we can distinguish the modeling capabilities in Pyomo from the functionality in Coopr and Pyomo that supports scripting. Pyomo's modeling capabilities are relatively mature. Although there have been significant enhancements to Pyomo in recent releases, with few exceptions the API of the modeling components has remained stable.

By contrast, the design of Coopr and Pyomo scripting functionality continues to evolve. The writing of this book has highlighted many design issues, some of which have been resolved in recent Coopr releases. However, we expect that the scripting functionality in Coopr and Pyomo will evolve significantly in future Coopr releases. Scripting is increasingly becoming the main way that users interact with Pyomo models. Consequently, scripting use-cases will likely drive many improvements in the design of Pyomo and Coopr.

10.A Examples

10.A.1 A Simple Optimization Script

A Python script that creates an instance of the Pyomo model in Example 8.A.4 with a data file, and performs optimization using the IPOPT solver.

```
from coopr.opt import SolverFactory
from DiseaseEstimation import model

model.pprint()

instance = model.create('DiseaseEstimation.dat')
instance.pprint()

opt = SolverFactory("ipopt")
results = opt.solve(instance)

results.write()
```

10.A.2 A Simple Optimization Script with Multiple Data Files

A Python script that creates an instance of the Pyomo model in Example 8.A.4 using two different data files, and performs optimization using the IPOPT solver.

```
from coopr.pyomo import ModelData
from coopr.opt import SolverFactory
from DiseaseEstimation import model

model.pprint()

modeldata = ModelData()
modeldata.add('DiseaseEstimation.dat')
modeldata.add('DiseasePop.dat')
modeldata.read(model)

instance = model.create(modeldata)
instance.pprint()

opt = SolverFactory("ipopt")
results = opt.solve(instance)

results.write()
```

10.A.3 A More Comprehensive Optimization Script

A Python script that creates an instance of the Pyomo model in Example 8.A.4 with a data file, and performs optimization using the IPOPT solver. This script provides more detailed control of the optimization process than is provided in Example 10.A.1.

```
from coopr.pyomo import *
from coopr.opt import SolverFactory, SolverManagerFactory

from DiseaseEstimation import model

# create the instance
instance = model.create('DiseaseEstimation.dat')

# define the solver and its options
solver = 'ipopt'
opt = SolverFactory( solver )
if opt is None:
   raise ValueError, "Problem constructing solver \
       `"+str(solver)
opt.set_options('max_iter=2')

# create the solver manager
solver_manager = SolverManagerFactory( 'serial' )

# solve
results = solver_manager.solve(instance, opt=opt, tee=True,\
     timelimit=None)
instance.load(results)

# display results
display(instance)
```

10.A.4 A Comprehensive Optimization Script that Mimics the `pyomo` *Command*

A Python script that creates an instance of the Pyomo model in Example 8.A.4 with a data file, and performs optimization using the IPOPT solver. This script leverages Pyomo scripting functionality to provide a high-level of control over the optimization process.

```
# Mimic the pyomo script
from coopr.pyomo import *
from pyutilib.misc import Options

# set high level options that mimic pyomo comand line
options = Options()
options.model_file = 'DiseaseEstimation.py'
options.data_files = ['DiseaseEstimation.dat']
options.solver = 'ipopt'
options.solver_io = 'nl'
#options.keepfiles = True
#options.tee = True

# mimic the set of function calls done by pyomo command line
scripting.util.setup_environment(options)

# the following imports the model found in \
    options.model_file,
# sets this to options.usermodel, and executes preprocessors
scripting.util.apply_preprocessing(options, parser=None)

# create the wrapper for the model, the data, the instance,\
     and the options
model_data = scripting.util.create_model(options)
instance = model_data.instance

# solve
results, opt = scripting.util.apply_optimizer(options, \
    instance)

# the following simply outputs the final time elapsed
scripting.util.finalize(options)

# load results into instance and print
instance.load(results)
display(instance)
```

10.A.5 An Optimization Script with Multiple Models

A Python script that creates the concrete Pyomo models in Example 2.A.1 and 2.A.2, performs optimization using the GLPK linear programming solver, and then compares the value of model variables.

```
from coopr.opt import SolverFactory
from concrete1 import model as model1
from concrete2 import model as model2

opt = SolverFactory("glpk")

instance1 = model1.create()
results1 = opt.solve(instance1)
instance1.load(results1)

instance2 = model2.create()
results2 = opt.solve(instance2)
instance2.load(results2)

from coopr.pyomo import value

print "x_2 value:", instance1.x_2.value
print "x_2 value:", value(instance1.x_2)
print "x[2] value:", instance2.x[2].value
print "x[2] value:", value(instance2.x[2])

print "x_2 object:", instance1.x_2
print "x[2] object:", instance2.x[2]

if value(instance1.x_2) == value(instance2.x[2]):
   print "x_2 == x[2]"
else:
   print "x_2 != x[2]"

if instance1.x_2 == instance2.x[2]:
   print "x_2 == x[2]"
else:
   print "x_2 != x[2]"
```

10.A.6 An Optimization Script that Prints the Value of Variables

A Python script that creates the concrete Pyomo model in Example 2.A.2, performs optimization using the GLPK linear programming solver, and then outputs the values of all variables.

```
from coopr.opt import SolverFactory
from concrete2 import model

instance = model.create()

opt = SolverFactory("glpk")
results = opt.solve(instance)
instance.load(results)

for index in instance.x:
   print instance.x[index], instance.x[index].value

from coopr.pyomo import Var
for var in instance.active_components(Var):
   print "Variable",var
   obj = getattr(instance, var)
   for index in obj:
      print obj[index], obj[index].value
```

10.A.7 A Script that Performs Optimization from Two Starting Points

A Python script that finds and prints the solution for two starting different starting points.

```
from coopr.pyomo import *
from pyutilib.misc import Options
from math import pi

model = ConcreteModel()

model.x = Var(bounds=(0,4))
model.y = Var(bounds=(0,4))

def multimodal(m):
   return (2-cos(pi*m.x)-cos(pi*m.y)) * (m.x**2) * (m.y**2)
model.obj = Objective(rule=multimodal, sense=minimize)

instance = model.create();

options = Options()
options.solver = 'ipopt'
options.quiet = True

instance.x = 0.25
instance.y = 0.25
print "x0=", instance.x.value, " y0=", instance.y.value,
results, opt = scripting.util.apply_optimizer(options, \
    instance)
instance.load(results)
print "x*=", instance.x.value, " y*=", instance.y.value

instance.x = 2.5
instance.y = 2.5
print "x0=", instance.x.value, " y0=", instance.y.value,
results, opt = scripting.util.apply_optimizer(options, \
    instance)
instance.load(results)
print "x*=", instance.x.value, " y*=", instance.y.value
```

10.A.8 A Script that Performs Optimization from a Grid of Starting Points

A Python script that explores a grid of starting points and keeps a list of all of the unique solutions that are found.

```
from coopr.pyomo import *
from pyutilib.misc import Options
from math import pi

model = ConcreteModel()

model.x = Var(bounds=(2,4))
model.y = Var(bounds=(2,4))

def multimodal(m):
   return (2-cos(pi*m.x)-cos(pi*m.y)) * (m.x**2) * (m.y**2)
model.obj = Objective(rule=multimodal, sense=minimize)

instance = model.create();

options = Options()
options.solver = 'ipopt'
options.quiet = True

# define a new class to store, compare, and print potential\
    solutions
class Solution(object):
   def __init__(self, xinit, yinit, xsol, ysol, objsol):
      self.xinit = xinit
      self.yinit = yinit
      self.xsol = xsol
      self.ysol = ysol
      self.objsol = objsol

   def is_different(self, soln_obj, tol):
      if (abs(self.xsol - soln_obj.xsol) > tol or \
          abs(self.ysol-soln_obj.ysol) > tol):
         return True
      return False

   def pprint(self):
      print 'x0=', self.xinit, 'y0=', self.yinit,
      print 'x*=', self.xsol, 'y*=', self.ysol,
      print 'obj*=', self.objsol

# setup the grid of starting points
N=8
step = 0.25
xinit = [2 + i*step for i in range(0,N+1)]
yinit = [2 + i*step for i in range(0,N+1)]

# loop through all the starting points and add
```

```
# unique solns to the list
unique_solns = list()
for i in range(0, N):
   for j in range(0,N):
      # initialize at the current grid point and solve the \
          problem
      instance.x = xinit[i]
      instance.y = yinit[j]
      results, opt = \
          scripting.util.apply_optimizer(options, instance)
      instance.load(results)

      # create a Solution object for the candidate solution
      candidate_soln = Solution(xinit[i], yinit[j], \
          instance.x.value, instance.y.value, 0.0)

      soln_is_unique = True
      # loop through the current list of unique solutions
      # and see if the candidate solution is new
      for soln in unique_solns:
         if (soln.is_different(candidate_soln, 1e-3) == \
             False):
            # soln is already in the list
            soln_is_unique = False
            break

      if soln_is_unique:
         unique_solns.append(candidate_soln)

   # print progress
   print 'Percent Complete:', float(i+1)/N*100, '%'

# print out the unique solutions
print
print "*** Unique Solutions Found ***"
for soln in unique_solns:
   soln.pprint()
```

10.A.9 A Script that Reoptimizes a Model after Fixing Variables

A Pyomo script that creates the nonlinear multimodal problem in Example 8.A.2, performs optimization with IPOPT, and reoptimizes the problem with a fixed variable.

```
from coopr.opt import SolverFactory
from multimodal import model

opt = SolverFactory("ipopt")
instance = model.create()

instance.y = 3.5
instance.x = 3.5
instance.y.fixed = True
instance.preprocess()

results = opt.solve(instance)
instance.load(results)

xval = instance.x.value
print "First x was", instance.x.value, \
      "and y was", instance.y.value

instance.x.fixed = True
instance.y.fixed = False
instance.preprocess()

results = opt.solve(instance)
instance.load(results)

print "Next x was", instance.x.value, \
      "and y was", instance.y.value

instance.x.fixed = False
instance.y.fixed = True
instance.preprocess()

results = opt.solve(instance)
instance.load(results)

print "Finally x was", instance.x.value, \
      "and y was", instance.y.value
```

10.A.10 An Iterative Optimization Process that Explicitly Activates Select Constraints

A Pyomo script defining a model with both *hard* and *easy* constraints. A model with the easy constraints is solved first, and that solution is used to initialize the optimizer when solving with the hard constraints.

```
from coopr.pyomo import *
from pyutilib.misc import Options

model = AbstractModel()

model.S_SI = Set(ordered=True)

model.P_REP_CASES = Param(model.S_SI)
model.P_POP = Param()

model.I = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=1)
model.S = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=300)
model.beta = Var(bounds=(0.05, 70))
model.alpha = Var(bounds=(0.5, 1.5))
model.eps_I = Var(model.S_SI, initialize=0.0)

def _objective(model):
   return sum((model.eps_I[i])**2 for i in model.S_SI)
model.objective = Objective(rule=_objective, sense=minimize)

def _InfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.I[i-1]**model.alpha)/model.P_POP
   return Constraint.Skip
model.InfDynamics = Constraint(model.S_SI, \
    rule=_InfDynamics)

def _EasierInfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.P_REP_CASES[i-1])/model.P_POP
   return Constraint.Skip
model.EasierInfDynamics = Constraint(model.S_SI, \
    rule=_EasierInfDynamics)

def _SusDynamics(model, i):
   if i != 1:
      return model.S[i] == model.S[i-1] - model.I[i]
   return Constraint.Skip
model.SusDynamics = Constraint(model.S_SI, \
    rule=_SusDynamics)

def _Data(model, i):
```

```
   return model.P_REP_CASES[i] == model.I[i]+model.eps_I[i]
model.Data = Constraint(model.S_SI, rule=_Data)

instance = model.create('DiseaseEstimation.dat');

options = Options()
options.solver = 'ipopt'
options.quiet = True

# disable the hard constraints
instance.InfDynamics.deactivate()

# preprocess the instance to remove the disabled constraints
instance.preprocess()

# solve the problem with the easy constraints
print "*** Solving the \"easy\" problem"
results, opt = scripting.util.apply_optimizer(options, \
    instance)

# load the results so that they become the initial \
    conditions
# for the more difficult solve
instance.load(results)
print "beta from easy problem: " + str(instance.beta.value)
print "alpha from easy problem: " + \
    str(instance.alpha.value)
print

# enable the hard constraints and
# disable the easy constraints
instance.InfDynamics.activate()
instance.EasierInfDynamics.deactivate()

# preprocess the instance to remove the constraints
instance.preprocess()

# solve the problem with the hard constraints
print "*** Solving the \"hard\" problem"
results, opt = scripting.util.apply_optimizer(options, \
    instance)
instance.load(results)
print "beta from hard problem: " + str(instance.beta.value)
print "alpha from hard problem: " + \
    str(instance.alpha.value)
```

10.A.11 Sharing Results Between Models

A Pyomo disease model adapted from Example 10.A.11 that contains only the easy constraints.

```
from coopr.pyomo import *

model = AbstractModel()

model.S_SI = Set(ordered=True)

model.P_REP_CASES = Param(model.S_SI)
model.P_POP = Param()

model.I = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=1)
model.S = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=300)
model.beta = Var(bounds=(0.05, 70))
model.alpha = Var(bounds=(0.5, 1.5))
model.eps_I = Var(model.S_SI, initialize=0.0)

def _objective(model):
   return sum((model.eps_I[i])**2 for i in model.S_SI)
model.objective = Objective(rule=_objective, sense=minimize)

def _EasierInfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.P_REP_CASES[i-1])/model.P_POP
   return Constraint.Skip
model.EasierInfDynamics = Constraint(model.S_SI, \
    rule=_EasierInfDynamics)

def _SusDynamics(model, i):
   if i != 1:
      return model.S[i] == model.S[i-1] - model.I[i]
   return Constraint.Skip
model.SusDynamics = Constraint(model.S_SI, \
    rule=_SusDynamics)

def _Data(model, i):
   return model.P_REP_CASES[i] == model.I[i]+model.eps_I[i]
model.Data = Constraint(model.S_SI, rule=_Data)
```

A Pyomo script that shares results between two models: `DiseaseEasy` and `DiseaseHard`.

```
from coopr.pyomo import *
from pyutilib.misc import Options
from DiseaseEasy import model as easy_model
from DiseaseHard import model as hard_model

easy_instance = easy_model.create('DiseaseEstimation.dat')
hard_instance = hard_model.create('DiseaseEstimation.dat')

options = Options()
options.solver = 'ipopt'
options.quiet = True

# solve the easier problem
results, opt = scripting.util.apply_optimizer(options, \
    easy_instance)

# load the results into the hard instance to provide
# a good initial point
hard_instance.load(results)

# solve the hard problem with the new initialization
results, opt = scripting.util.apply_optimizer(options, \
    hard_instance)

# print the final results
print results
```

10.A.12 Plotting Solver Results with Matplotlib

A Pyomo script that optimizes the disease model described in Example 8.A.4 and plots the estimated data versus the measured data.

```
from coopr.pyomo import *
from pyutilib.misc import Options

model = AbstractModel()

model.S_SI = Set(ordered=True)

model.P_REP_CASES = Param(model.S_SI)
model.P_POP = Param()

model.I = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=1)
model.S = Var(model.S_SI, bounds=(0,model.P_POP), \
    initialize=300)
model.beta = Var(bounds=(0.05, 70))
model.alpha = Var(bounds=(0.5, 1.5))
model.eps_I = Var(model.S_SI, initialize=0.0)

def _objective(model):
   return sum((model.eps_I[i])**2 for i in model.S_SI)
model.objective = Objective(rule=_objective, sense=minimize)

def _InfDynamics(model, i):
   if i != 1:
      return model.I[i] == (model.beta * model.S[i-1] * \
          model.I[i-1]**model.alpha)/model.P_POP
   return Constraint.Skip
model.InfDynamics = Constraint(model.S_SI, \
    rule=_InfDynamics)

def _SusDynamics(model, i):
   if i != 1:
      return model.S[i] == model.S[i-1] - model.I[i]
   return Constraint.Skip
model.SusDynamics = Constraint(model.S_SI, \
    rule=_SusDynamics)

def _Data(model, i):
   return model.P_REP_CASES[i] == model.I[i]+model.eps_I[i]
model.Data = Constraint(model.S_SI, rule=_Data)

# @script:
instance = model.create('DiseaseEstimation.dat');

options = Options()
options.solver = 'ipopt'
options.quiet = True
```

```
# solve the problem
results, opt = scripting.util.apply_optimizer(options, \
    instance)

# load the results to plot the solution
instance.load(results)
est_incidence = []
act_incidence = []
for i in instance.S_SI:
   est_incidence.append(value(instance.I[i]))
   act_incidence.append(value(instance.P_REP_CASES[i]))

import matplotlib.pyplot as plt

plt.plot(est_incidence)
plt.plot(act_incidence, 'o')
plt.show()
# @:script
```

10.A.13 A Sudoku Problem Solved by Iteratively Adding Cuts

A Pyomo script defining a function returning a Pyomo model for a Sudoku problem.

```
from coopr.pyomo import *

# This python file defines a function to create a
# model for the sudoku problem

# create a standard python dict for the map from
# subsq to (row,col) numbers
subsq_to_row_col = dict()

subsq_to_row_col[1] = [(i,j) for i in xrange(1,4) for j in \
    xrange(1,4)]
subsq_to_row_col[2] = [(i,j) for i in xrange(1,4) for j in \
    xrange(4,7)]
subsq_to_row_col[3] = [(i,j) for i in xrange(1,4) for j in \
    xrange(7,10)]

subsq_to_row_col[4] = [(i,j) for i in xrange(4,7) for j in \
    xrange(1,4)]
subsq_to_row_col[5] = [(i,j) for i in xrange(4,7) for j in \
    xrange(4,7)]
subsq_to_row_col[6] = [(i,j) for i in xrange(4,7) for j in \
    xrange(7,10)]

subsq_to_row_col[7] = [(i,j) for i in xrange(7,10) for j in\
     xrange(1,4)]
subsq_to_row_col[8] = [(i,j) for i in xrange(7,10) for j in\
     xrange(4,7)]
subsq_to_row_col[9] = [(i,j) for i in xrange(7,10) for j in\
     xrange(7,10)]

# use this function to create the sudoku model defining a \
    list
# of integer cuts. Entry i in cut_on lists all the (r,c,v) \
    tuples
# where y[r,c,v] was 1. Entry i in cut_off lists the ones \
    that were 0
# The input board is a list of the fixed numbers in the \
    board in
# (r,c,v) tuples
def create_sudoku_model( cut_on, cut_off, board ):

   model = ConcreteModel()

   # create sets for rows columns and squares
   model.ROWS = RangeSet(1,9)
   model.COLS = RangeSet(1,9)
   model.SUBSQUARES = RangeSet(1,9)
   model.VALUES = RangeSet(1,9)
   model.CUTS = RangeSet(1,len(cut_on))
```

```
    # create the binary variables to define the values
    def _y_rule(model, r, c, v):
        if (r,c,v) in board:
            model.y[r,c,v].fixed = True
            return 1
        return 0
    model.y = Var(model.ROWS, model.COLS, model.VALUES, \
        initialize=_y_rule, within=Binary)

    # create the objective - this is a feasibility problem
    # so we just make it a constant
    def _Obj(model):
        return 1
    model.obj = Objective(rule=_Obj)

    # exactly one number in each row
    def _RowCon(model, i, v):
        return sum(model.y[i,c,v] for c in xrange(1,10)) == 1
    model.RowCon = Constraint(model.ROWS, model.VALUES, \
        rule=_RowCon)

    # exactly one nubmer in each column
    def _ColCon(model, j, v):
        return sum(model.y[r,j,v] for r in xrange(1,10)) == 1
    model.ColCon = Constraint(model.COLS, model.VALUES, \
        rule=_ColCon)

    # exactly one number in each subsquare
    def _SubSqCon(model, b, v):
        return sum(model.y[t[0],t[1],v] for t in \
            subsq_to_row_col[b]) == 1
    model.SubSqCon = Constraint(model.SUBSQUARES, \
        model.VALUES, rule=_SubSqCon)

    # exactly one number in each cell
    def _ValueCon(model, i, j):
        return sum(model.y[i, j, v] for v in xrange(1,10)) ==\
            1
    model.ValueCon = Constraint(model.ROWS, model.COLS, \
        rule=_ValueCon)

    # integer cuts to prune previous solutions
    def _IntCuts(model, i):
        return sum( (1.0-model.y[r,c,v]) for (r,c,v) in \
            cut_on[i-1] ) + \
            sum ( model.y[r,c,v] for (r,c,v) in cut_off[i-1] \
                ) \
            >= 1
# if (len(cut_on) > 0):
# model.IntCuts = Constraint(model.CUTS, rule=_IntCuts)
    model.IntCuts = Constraint(model.CUTS, rule=_IntCuts)

    return model
```

A Pyomo script that iteratively adds cuts for solutions that have been found.

```
from coopr.pyomo import *
from pyutilib.misc import Options
from sudoku import create_sudoku_model

# define the board
board = [(1,1,5),(1,2,3),(1,5,7), \
       (2,1,6),(2,4,1),(2,5,9),(2,6,5), \
       (3,2,9),(3,3,8),(3,8,6), \
       (4,1,8),(4,5,6),(4,9,3), \
       (5,1,4),(5,4,8),(5,6,3),(5,9,1), \
       (6,1,7),(6,5,2),(6,9,6), \
       (7,2,6),(7,7,2),(7,8,8), \
       (8,4,4),(8,5,1),(8,6,9),(8,9,5), \
       (9,5,8),(9,8,7),(9,9,9)]

# create the empty list of cuts to start
cut_on = []
cut_off = []

done = False
while (not done):
   model = create_sudoku_model(cut_on, cut_off, board)
   instance = model.create()

   options = Options()
   options.solver = 'glpk'
   options.quiet = True
   #options.tee = True

   results, opt = scripting.util.apply_optimizer(options, \
       instance)
   instance.load(results)

   if str(results.Solution.Status) != 'optimal':
      break

   # add cuts
   new_cut_on = []
   new_cut_off = []
   for r in instance.ROWS:
      for c in instance.COLS:
         for v in instance.VALUES:
            # check if the binary variable is on or off
            # note, it may not be exactly 1
            if value(instance.y[r,c,v]) >= 0.5:
               new_cut_on.append((r,c,v))
            else:
               new_cut_off.append((r,c,v))

   cut_on.append(new_cut_on)
   cut_off.append(new_cut_off)
```

```
print "Solution #" + str(len(cut_on))
for i in xrange(1,10):
   for j in xrange(1,10):
      for v in xrange(1,10):
         if value(instance.y[i,j,v]) >= 0.5:
            print v, " ",
   print
```

10.A.14 Hybrid Optimization for Parameter Estimation

A Pyomo script that iterates between two different optimizers to find global optima.

```
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
```

The disease_mdl function, which manages the construction of the Pyomo model used in this example.

```
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
```

Appendix A
Installing Coopr

Abstract Pyomo is a package within the Coopr software framework. Coopr is a collection of Python packages that support optimization. Consequently, Pyomo users need to install a variety of Coopr packages. This chapter describes various ways that Coopr can be installed.

A.1 Installation Overview

Coopr is available as open-source software under the BSD license. It requires Python version 2.6 or 2.7. Coopr requires the installation of several freely available Python libraries, and it can optionally employ other libraries, including Python extension libraries for commercial optimization software.

Coopr is executed using Python, which is freely available in source or binary form from the Python website (see http://www.python.org/download). The standard Python distribution is built with the C language, and thus this is sometimes called CPython. CPython distributions are available for all major platforms, and this is the most common version of Python that is used in practice. Most Coopr packages are designed to work with versions 2.6 and 2.7 of CPython. The Python language underwent some rather large changes to version 3.0. These changes are not backward-compatible with Python 2.x, and consequently Coopr does not currently work with Python 3.x.

There are several different ways that Coopr can be installed:

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5,

Installer	Coopr's installer executables can be used on Windows platforms to install Coopr and the Python packages that Coopr depends on.
easy_install	Coopr releases can be directly installed using the Python `easy_install` command.
coopr_install	Use the `coopr_install` command to install a Coopr release in a virtual Python installation.
coopr_install + svn	Use the `coopr_install` command to install a trunk or stable branch of Coopr in a virtual Python installation.
Source	Coopr can be installed from source.

The first two options are simpler, but they require administrative privileges. The third and fourth options can be used by any user to install Coopr in a user-defined directory, but the fourth option requires the installation of Subversion. The final option is complex for a multi-package software framework like Coopr, and we do not describe this option in detail here.

The remaining sections in this chapter describe these installation options. The Coopr wiki provides additional details concerning the prerequisite software packages that are used to install and use Coopr. Note that the Coopr installation does not include the solver packages that Coopr uses to perform optimization. The Coopr wiki describes these solvers and provides links for downloading and installing these packages.

A.2 Using an Installer

On Windows platforms, Coopr installer executables provide the simplest installation process. The Coopr installer includes the Coopr software, as well as related Python packages that Coopr depends on. This installer does not automate the installation of solver packages that are used to optimize Coopr models. The user will need to install these packages separately, and update the system PATH environment if necessary to ensure that their associated executables can be found by Coopr.

The Windows installer can be downloaded from the Coopr wiki. A series of windows walk users through various steps in the installation process. Previous versions of Coopr will be removed to ensure that a robust installation is performed. Also, note that the installation of some packages will require an internet connection to automate the download of the most recent version of the software. If this is not possible, then these packages will need to be downloaded and installed separately.

The Coopr installer places the Coopr software in the site-packages directory for your Python installation. Consequently, these installers require administrative privileges to install the Coopr software. Once installation is complete, you can use Coopr within Python without specifying any additional system configuration. For example, you do not need to setup a `PYTHONPATH` environment. Additionally, `pyomo` and the other commands provided by Coopr are installed in the Python `Scripts` directory.

A.3 Installing Coopr as a Site Package

A standard way to install Python packages like Coopr is to install them as a Python site package. This generally requires administrative privileges, since Python is installed in system directories that are not accessible to general users. Additionally, this requires internet access to download Coopr and the packages it depends on.

Coopr can be installed with the following steps:

1. Download the `wget` command if you are running on MS Windows (see http://users.ugent.be/ bpuyge/wget)
2. Download and install the `setuptools` package:

```
wget http://peak.telecommunity.com/dist/ez_setup.py
python ez_setup.py
```

3. Run `easy_install` to install Coopr:

```
easy_install Coopr
```

The `easy_install` command downloads and installs Coopr from PyPI (http://pypi.python.org/pypi), which is the standard web-based service for hosting Python packages. This command handles package dependencies automatically, so there are no additional steps required; Coopr packages can be imported with your Python installation, and Coopr scripts are installed with other Python commands.

Notes:

- The `pip` package is now generally recommended over `setuptools`. If `pip` is installed, then you can install Coopr with

```
pip install Coopr
```

- If your internet connect is managed with a proxy server, then you need to specify the name of the server in the `HTTP_PROXY` environment.

A.4 Installing a Coopr Release Using `coopr_install`

Another simple way way to install Coopr is to use the `coopr_install` script to create a virtual Python environment. This environment is an isolated Python installation that depends on the system Python installation, but which mimics a complete Python installation. A virtual Python environment can be created by a user in their directory without requiring administrative privileges. However, running `coopr_install` requires internet access to download Coopr and the packages it depends on.

Coopr can be installed with the following two steps:

1. Download the `coopr_install` package from the Coopr wiki.
2. Run `coopr_install` to install Coopr:

```
python coopr_install coopr
```

The `coopr_install` command downloads and installs Coopr from PyPI, which is the standard web-based service for hosting Python packages. This command handles package dependencies automatically, so there are no additional steps required. This script creates the virtual Python environment in the `coopr` directory, and the Coopr scripts are installed in the `coopr/bin` directory.

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

```
admin       Administrative data for maintaining this
            distribution
bin         Scripts and executables
dist        Python packages that are not intended for
            development
doc         Coopr documentation and tutorials
examples    Coopr examples
include     Python header files
lib         Python libraries and installed packages
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 the user's `PATH` environment, then the `bin/python` command can be used to execute Coopr packages packages without further configuration. Further, Coopr's Python scripts are installed in the `bin` directory such that they reference this virtual Python installation directly. Note that the `bin` directory needs to be added *before* the system Python path to ensure that the virtual Python installation is used.

Notes:

- If your internet connect is managed with a proxy server, then you need to specify the name of the server in the `HTTP_PROXY` environment.
- The `--help` option can be used to print documentation for the command-line options of `coopr_install`. See the Coopr wiki for additional details for this command.

A.5 Installing a Development Branch Using `coopr_install`

The `coopr_install` script also includes options that allow Coopr packages to be directly checked out from its Subversion repository. These options are useful for both *power users*, who interact closely with the core developers, and to obtain updates for Coopr packages that are under active development and extension. Running

`coopr_install` requires internet access to download Coopr and the packages it depends on, and these options also require the installation of the Subversion software.

Coopr can be installed with the following two steps:

1. Download the `coopr_install` package from the Coopr wiki.
2. Run `coopr_install` to install Coopr. For example

```
python coopr_install --trunk coopr
```

   installs the trunk branches of Coopr and PyUtilib packages.

Although these options perform Subversion checkouts, read-only access to the Subversion repository does not require a password. This command handles package dependencies automatically, so no additional steps are required. This script creates the virtual Python environment in the `coopr` directory, and the Coopr scripts are installed in the `coopr/bin` directory.

Notes:

- If your internet connect is managed with a proxy server, then you need to specify the name of the server in the `HTTP_PROXY` environment.
- The Subversion software may need to be configured to work with a proxy server on your network.
- The Subversion software is commonly available on Linux and MacOS, but it is not preinstalled with MS Windows.

A.6 Discussion

Other Package Dependencies

Coopr installs with a variety of third-party libraries required for executing Coopr packages. However, there are a variety of other Python packages with which Coopr has a weak dependency; if these packages are installed, then Coopr provides additional capabilities. For example, Coopr is weakly dependent on database interfaces like `pyodbc`. These packages have complicated installation procedures, or require the compilation of Python extensions (with a C or C++ compiler). Thus, Coopr's installation process cannot ensure that these can be installed.

These packages are often easily installed using the `easy_install` utility. For example, it is convenient to install `PyYAML` using `easy_install`:

```
easy_install pyyaml
```

Additionally, a wide variety of useful packages can be simply installed by installing the `coopr.extras` package:

```
easy_install coopr.extras
```

If you install Coopr with the `coopr_install` command, you can use the `--with-extras` option to automatically install all packages in `coopr.extras`.

However, this requires an internet connection to perform this installation. See the Coopr wiki for further details concerning the installation of these packages.

Other Python Implementations

Other noteworthy implementations of Python are Jython, written in Java, and IronPython, written for the Common Language Runtime. A major difference between CPython, Jython, and IronPython is that they support extensions and integrations with C, Java and the .NET application framework, respectively. For example, Jython programs can employ Java's Swing GUI toolkit, and Jython programs can be distributed in Java JAR files. Coopr is developed and tested with CPython, and Coopr 3.1 is known to not work properly on some other Python implementations.

Installing Solvers

Coopr can employ both open source and commercial solvers to perform optimization. Unfortunately, it can be difficult to install these solvers. This is especially true for open source solvers that are not commonly distributed with binary executables that are easily installed on many platforms. One possible solution is to use the `PyCoinInstall` project, which provides a script that manages the creation of a variety of optimization-related packages. PyCoinInstall is focused on the installation of Python projects related to COIN-OR, but it also supports the installation of other software like scipy, glpk, and Acro that can be leveraged by the COIN software. PyCoinInstall is a component of the COIN-OR CoinBazaar project (see https://projects.coin-or.org/CoinBazaar). However, it is not under active development.

Appendix B
A Brief Python Tutorial

Abstract This chapter provides a short tutorial of the Python programming language. This chapter briefly covers basic concepts of Python, including variables, expressions, control flow, functions, and classes. The goal is to provide a reference for that Python constructs that are used in the book. A full introduction to Python is provided by resources such as those listed at the end of the chapter.

B.1 Overview

Python is a powerful programming language that is easy to learn. Python is an interpreted language, so developing and testing Python software does not require the compilation and linking that is required by traditional software languages like FORTRAN and C. Furthermore, Python includes a command-line interpreter that can be used interactively. This allows the user to work directly with Python data structures, which is invaluable for learning about data structure capabilities and for diagnosing software failures.

Python has an elegant syntax that enables programs to be written in a compact, readable style. Programs written in Python are typically much shorter than equivalent software developed with languages like C, C++, or Java because:

- Python supports many high-level data types that simplify complex operations
- Python uses indentation to group statements, which enforces a clean coding style
- Python uses dynamically typed data, so variable and argument declarations are not necessary

Python is a highly structured programming language that provides support for large software applications. Consequently, Python is a much more powerful language than many scripting tools (e.g., shell languages and Windows batch files). Python also includes modern programming language features like object-oriented programming, as well as rich set of built-in standard libraries that can be used to quickly build sophisticated software applications.

The goal in this Appendix is to provide a reference for Python constructs used in the rest of the book. A full introduction to Python is provided by resources such as those listed at the end of the chapter.

B.2 Installing and Running Python

Python codes are executed using an interpreter. When this interpreter starts, a command prompt is printed and the interpreter waits for the user to enter Python commands. For example, a standard way to get started with Python is to execute the interpreter from a shell environment and then print "Hello World":

```
% python
Python 2.6 (r26:66714, Nov 3 2009, 17:33:38)
[GCC 4.4.1 20090725 (Red Hat 4.4.1-2)] on linux2
>>> print "Hello World"
Hello World
>>>
```

On Windows the `python` command can be launched from the DOS shell, and on Linux the `python` command can be launch from a bash or csh shell. The Python interactive shell is similar to these shell environments; when a user enters a valid Python statement, it is immediately evaluated and its corresponding output is immediately printed.

The interactive shell is useful for interrogating the state of complex data types. In most cases, this will involve single-line statements, like the "print" statement shown above. Multi-line statements can also be entered into the interactive shell. Python uses the "..." prompt to indicate that a continuation line is needed to define a valid multi-line statement. For example, a conditional statement requires a block of statements that are defined on continuation lines:

```
>>> x = True
>>> if x:
... print "x is True"
... else:
... print "x is False"
x is True
```

(Note: `True` is a predefined Python literal so `x = True` assigns this value to `x` in the same way that the predefined literal `6` would be assigned by `x = 6`.)

The Python interpreter can also be used to execute Python statements in a file, which allows the automated execution of complex Python programs. Python source files are text files, and the convention is to name source files with the `.py` suffix. For example, consider the `example.py` file:

```
# This is a comment line, which is ignored by Python

print "Hello World"
```

The code can be executed in several ways. Perhaps the most common is to execute the Python interpreter within a shell environment:

```
% python example.py
Hello World
%
```

On Windows, Python programs can be executed by double clicking on a `.py` file; this launches a console window in which the Python interpreter is executed. The console window terminates immediately after the interpreter executes, but this example can be easily adapted to wait for user input before terminating:

```
# A modified example.py program
print "Hello World"

import sys
sys.stdin.readline()
```

B.3 Python Line Format

Python does not make use of begin-end symbols for blocks of code. Instead, a colon is used to indicate the end of a statement that defines the start of a block and then indentation is used to demarcate the block. For example, consider a file containing the following Python commands:

```
# This comment is the first line of LineExample.py
# all characters on a line after the #-character are
# ignored by python

print "Hello World, have you lost weight?"

weight = 400

if weight > 300:
   print "Oh, sorry, I guess not."
   print "My mistake."
else:
   print "Keep up the good work!"
```

When passed to Python, this program will cause some text to be output.

Because indentation has meaning, Python requires consistency. The following program will generate an error message because the indentation within the if-block is inconsistent:

```
# This comment is the first line of BadIndent.py
# it will cause python to give an error message
# concerning indentation

print "Hello World, have you lost weight?"

weight = 200

if weight > 300:
   print "Oh, sorry, I guess not."
      print "My mistake."
else:
   print "Keep up the good work!"
```

Generally, each line of a Python script or program contains a single statement. Long statements with long variable names can result in very long lines in a Python script. Although this is syntactically correct, it is sometimes convenient to split a statement across two or more lines. The backslash (\) tells Python that text that is logically part of the current line will be continued on the next line. In a few situations, a backslash is not needed for line continuation. For Pyomo users, the most important case where the backslash is not needed is in the argument list of a function. Arguments to a function are separated by commas, and after a comma the arguments can be continued on the next line without using a backslash.

Conversely, it is sometimes possible to combine multiple Python statements on one line. However, we recommend against it as a matter of style and to enhance maintainability of code.

B.4 Variables and Data Types

Python variables do not need to be explicitly declared. A statement that assigns a value to an undefined symbol implicitly declares the variable. Additionally, a variable's type is determined by the data that it contains. The statement

```
weight=200
```

creates a variable called `weight`, and it has the integer type because `200` is an integer. Python is case sensitive, so the statement

```
Weight=``Less than yesterday.''
```

creates a variable that is not the same as `weight`. The assignment

```
weight = Weight
```

would cause the variable `weight` to have the same value as `Weight` and therefore the same type.

Python programmers need to be especially careful about types in code that does arithmetic. For example, division involving only integers results in integer division, which is often not intended. Consider the following program:

```
weight = 200
Weight = 400

print "Division of", weight,"by", Weight, \
      "gives",weight / Weight
print "Casting the operands to float gives",\
      float(weight) / float(Weight)
print "Casting the result to float gives",\
      float(weight / Weight)
```

Pyomo language users need to be careful about this issue if they write scripts that do arithmetic. Generally they cannot be sure what types the variables will have because the type depends on the data contained by the variable. A simple approach is to cast variables explicitly to `float` when floating point arithmetic is desired, as this example illustrates.

Note that this does not apply to objective and constraint expressions that are part of a Pyomo model that are evaluated by solvers. Our concern here is with expressions that are part of a Python script. The distinction may be subtle because script expressions may make use of Pyomo components and may be part of the code used to construct Pyomo components. Consider the following model:

```
model = AbstractModel()

model.pParm = Param(within=Integers, default = 2)
model.wParm = Param(within=PositiveIntegers, default = 4)
model.aVar = Var(within=NonNegativeReals)

def MyConstraintRule(model):
   if float(model.pParm) / float(model.wParm) > 0.6:
      return model.aVar / model.wParm <= 0.9
   else:
      return model.aVar / model.wParm <= 0.8
model.MyConstraint = Constraint(rule=MyConstraintRule)

def MyObjectiveRule(model):
   return model.wParm * model.aVar
model.MyObjective = Objective(rule=MyObjectiveRule,
                              sense=maximize)
```

The line that begins with `if` is interpreted by Python to control which expression will be used to construct a constraint that will be passed to a solver. Thus, it is important to use the `float()` function because a float result is desired in the conditional.

B.5 Data Structures

This section summarizes Python data structures that can be helpful in scripting Pyomo applications. Many Python and Pyomo data structures can be accessed by indexing their elements. Pyomo typically starts indices and ranges with a one, while Python is zero based.

B.5.1 Strings

String literals can be enclosed in either single or double quotes, which enables the other to be easily included in a string. Python supports a wide range of string functions and operations. For example, the addition operator (+) concatenates strings. To cast another type to string, use the `string` function. The Python line:

```
NameAge = ``SPAM was introduced in '' + string(1937)
```

assigns a string to the Python variable called `NameAge`.

B.5.2 Lists

Python lists correspond roughly to arrays in many other programming languages. Lists can be accessed element by element, as an entire list, or as a partial list. The slicing character is a colon (:) and negative indices indicate indexing from the end. The following Python session illustrates these operations:

```
>>> a = [3.14, 2.72, 100, 1234]
>>> a
[3.14, 2.72, 100, 1234]
>>> a[0]
3.14
>>> a[-2]
100
>>> a[1:-1]
[2.72, 100]
>>> a[:2] + ['bacon', 2*2]
[3.14, 2.72, 'bacon', 4]
```

The addition operator concatenates lists, and multiplication by an integer replicates lists. Note that in Python, lists can have mixed types such as the mixture of floats and integers just given.

There are many list functions. Perhaps the most common is the `append` function, which adds elements to the end of a list:

```
>>> a = []
>>> a.append(16)
>>> a.append(22.4)
>>> a
[16, 22.4]
```

B.5.3 Tuples

Tuples are similar to lists, but are intended to describe multi-dimensional objects. For example, it would be reasonable to have a list of tuples. Tuples differ from lists in that they use parentheses rather than square brackets for initialization. Additionally, the members of a list can be changed by assignment while tuples cannot be changed (i.e., tuples are immutable while lists are mutable). Although parentheses are used for initialization, square brackets are used to reference individual elements in a tuple (which is the same as for lists; this allows members of a list of tuples to be accessed with code that looks like access to an array).

Suppose we have a tuple intended to represent the location of a point in a three-dimensional space. The origin would be given by the tuple `(0,0,0)`. Consider the following Python session:

```
>>> orig = (0,0,0)
>>> pt = (-1.1, 9, 6)
>>> pt[1]
9
>>> pt = orig
>>> pt
(0, 0, 0)
```

For example, the statement

```
pt[1] = 4
```

would generate an error because tuple elements cannot be overwritten. Of course the entire tuple can be overwritten, since that assignment only impacts the variable containing the tuple.

B.5.4 Sets

Python sets are extremely similar to Pyomo Set components. Pyomo Set components implement the mathematical notion of a set, so they cannot have duplicate members and are, by default, unordered. Python sets are declared using the `set` function, which takes a list (perhaps an empty list) as an argument. Once a set has been created, it has member functions for operations such as `add` (one new mem-

ber), `update` (with multiple new members), and `discard` (existing members). The following Python session illustrates the functionality of `set` objects:

```
>>> A = set([1, 3])
>>> B = set([2, 4, 6])
>>> A.add(7)
>>> C = A | B
>>> print C
set([1, 2, 3, 4, 6, 7])
```

B.5.5 Dictionaries

Python dictionaries are somewhat similar to lists; however, they are unordered and they can be indexed by any immutable type (e.g., strings, numbers, tuples composed of strings and/or numbers, and more complex objects). The indices are called keys, and within any particular dictionary the keys must be unique. Dictionaries are created using brackets, and they can be initialized with a list of key-value pairs separated by commas. Dictionary members can be added by assignment of a value to the dictionary key. The values in the dictionary can be any object (even other dictionaries), but we will restrict our attention to simpler dictionaries. Here is an example:

```
>>> D = {}
>>> D['Bob'] = '123-1134'
>>> D['Alice'] = '331-9987'
>>> print D.keys()
['Bob', 'Alice']
>>> print D['Bob']
123-1134
```

B.6 Conditionals

Python supports conditional code execution using structures like:

```
if CONDITIONAL1:
   statements
elif CONDITIONAL2:
   statements
else:
   statements
```

Any number of `elif` statements (including none) and the `else` statement is optional can be used. Any number of statements can be in each conditional code block. The conditionals can be replaced by a logical expression, a call to a boolean function, or a boolean variable (and it could even be called `CONDITIONAL1`). The

boolean literals `True` and `False` are sometimes used in these expressions. The following program illustrates some of these ideas:

```
x = 6
y = False

if x == 5:
   print "x happens to be 5"
   print "for what that is worth"
elif y:
   print "x is not 5, but at least y is True"
else:
   print "This program cannot tell us much."
```

B.7 Iterations and Looping

As is typical for modern programming languages, Python offers `for` and `while` looping as modified by `continue` and `break` statements. When an `else` statement is given for a `for` or `while` loop, the code block controlled by the `else` statement is executed when the loop terminates. The `continue` statement causes the current block of code to terminate and transfers control to the loop statement. The `break` command causes an exit from the entire looping construct.

The following example illustrates these constructs:

```
D = {'Mary':231}
D['Bob'] = 123
D['Alice'] = 331
D['Ted'] = 987

for i in D.keys():
   if i == 'Alice':
      continue
   if i == 'John':
      print "Loop ends. Cleese alert!"
      break;
   print i, D[i]
else:
   print "Cleese is not in the list."
```

In this example, the for-loop iterates over all keys in the dictionary. The `in` keyword is particularly useful in Python to facilitate looping over iterable types such as lists and dictionaries. Note that the order of keys is arbitrary; the `sorted()` function can be used to sort them.

This program will print the list of keys and dictionary entries, except for the key "Alice," and then it prints "No chance that Cleese is in the list." If the name "John" was one of the keys, the loop would terminate whenever it was encountered and in that case, the `else` clause would skipped because `break` causes control to exit the entire looping structure, including its `else`.

B.8 Generators and List Comprehensions

Generators and list comprehensions are closely related. List comprehensions are commonly used in Pyomo models because they create a list "on-the-fly" using a concise syntax. Generators allow iteration over a list without creating it.

Before discussing list comprehensions and generators, it is helpful to review the Python `range` function. It takes up to three integer arguments: `start`, `beyond`, and `step`. The `range` function returns a list that starts with `start`, adds `step` to it for each element, and stops without creating `beyond`. The default value for `start` is zero and the default value for `step` is one. If only one argument is given, it is `beyond`. If two arguments are given, then they are `start` and `beyond`.

A list comprehension is an expression within square brackets that specifies the creation of a list. The following Python session illustrates the use of a list comprehension that generates the squares of the first six natural numbers:

```
>>> a = [i*i for i in range(1,6)]
>>> a
[1, 4, 9, 16, 25]
```

A common use for list comprehensions in Pyomo models is to create a list to pass to the `sum` function. Suppose an abstract model has defined a Pyomo `Var` called `model.N` and Pyomo `Param` objects called `model.P` and `model.B`, then the following rule might be use to create a constraint expression:

```
def constraint_rule(i, model):
  return sum([model.N[j,i] for j in model.P if j > i]) >= 0
```

Generators are used in iteration expressions (such as `for` loops) in the same way that list comprehensions are, but they do not actually create a list. In some situations, the memory and time savings that result from using a generator versus a list comprehension can be important.

An interesting generator is the Python `xrange` function. It takes the same arguments as the `range` function but does not generate a list. For example, the statements

```
a = [i*i for i in range(1,6)]
```

and

```
a = [i*i for i in xrange(1,6)]
```

are both list comprehensions that result in exactly the same list being assigned to the variable `a`. The only difference is that the function `range` returns a list that it creates, while `xrange` is a generator so it returns one value at a time as it is iterated.

Similarly, the `constraint_rule` function can be rewritten to use a generator rather than a list expression:

```
def constraint_rule(i, model):
  return sum(model.N[j,i] for j in model.P if j > i) >= 0
```

This syntax is simpler and more efficient.

B.9 Functions

Python functions can take objects as arguments and return objects. Because Python offers built-in types like tuples, lists, and dictionaries, it is easy for a function to return multiple values in an orderly way. Writers of a function can provide default values for unspecified arguments, so it is common to have Python functions that can be called with a variable number of arguments. In Python, a function is also an object; consequently, functions can be passed as arguments to other functions.

Function arguments are passed by reference, but many types in Python are immutable so it can be a little confusing for new programmers to determine which types of arguments can be changed by a function. It is somewhat uncommon for Python developers to write functions that make changes to the values of any of their arguments. However, if a function is a member of a class, it is very common for the function to change data within the object that called it.

User-defined functions are declared with a `def` statement. The `return` statement causes the function to end and the specified values to be returned. There is no requirement that a function return anything; the end of the function's indent block can also signal the end of a function.

Some of these concepts are illustrated by the

```
def Apply(f, a):
    r = []
    for i in range(len(a)):
        r.append(f(a[i]))
    return r

def SqifOdd(x):
    # if x is odd, 2*(x/2) is not x
    # due to integer divide of x/2
    if 2*(x/2) == x:
        return x
    else:
        return x*x

ShortList = range(4)
B = Apply(SqifOdd, ShortList)
print B
```

This program prints `[0, 1, 2, 9]`. The `Apply` function assumes that it has been passed a function and a list; it builds up a new list by applying the function to the list and then returns the new list. The `SqifOdd` function returns its argument (`x`) unless `2*(x/2)` is not `x`. If `x` is an odd integer, then `x/2` will be an integer divide (because 2 is also an integer) so the result will be truncated (e.g., the integer division of three by two results in one) and the truncated result times two will not be equal to `x`.

B.10 Objects and Classes

Classes define objects. Put another way: objects instantiate classes. Objects can have members that are data or functions. In this context, functions are often called methods. As an aside, we note that in Python both data and functions are technically objects, so it would correct to simply say that objects can have member objects.

User-defined classes are declared using the `class` command and everything in the indent block of a class command is part of the class definition. An overly simple example of a class is a storage container that prints its value:

```
from types import *

class IntLocker:
   sint = None
   def __init__(self, i):
      self.set_value(i)
   def set_value(self, i):
      if type(i) is not IntType:
         print "Error: ", i, " is not integer."
      else:
         self.sint = i
   def pprint(self):
      print "The Int Locker has ",self.sint

a = IntLocker(3)
a.pprint()
a.set_value(5)
a.pprint()
```

The class `IntLocker` has a member data element called `sint` and two member functions. When a member function is called, Python automatically supplies the object as the first argument. Thus, it makes sense to list the first argument of a member function as `self`, because this is the way that a class can refer to itself. The `__init__` method is a special member function that is automatically called when an object is created; this function is not required.

B.11 Modules

A *module* is a file that contains Python statements. For example, any file containing a Python "program" that defines classes or functions is a module. Definitions from one module can be made available in another module (or program file) via the `import` command, which can specify which names to import or specify the import of all names by using an asterisk.

Python is typically installed with many standard modules present, such as `types`. The command `from types import *` causes the import of all names from the `types` module.

Multiple module files in a directory can be organized into a *package*, and packages can contain modules and subpackages. Imports from a package can use a statement that gives the package name (i.e., directory name) followed by a dot followed by a the module name. For example, the command `from coopr.pyomo import *` imports the names from the `coopr` package module called `pyomo`.

B.12 Python Resources

- *Python Home Page*, http://www.python.org.
- *Python Essential Reference*, David M. Beazley, Addison-Wesley, 2009.

Appendix C
Pyomo and Coopr: The Bigger Picture

Abstract Pyomo is distributed as a package within the Coopr software library. This allows Pyomo to leverage the optimization capabilities in Coopr, including optimization solvers. Similarly, Coopr includes packages that extend Pyomo's modeling functionality; the PySP stochastic programming package is supported in this manner. This chapter provides an overview of Coopr. Additional detail is provided for key aspects of Coopr that are useful for analyzing Pyomo models.

C.1 Coopr Overview

Coopr is a collection of Python software packages that supports a diverse set of optimization capabilities for formulating and analyzing optimization models. Pyomo is a key driver for Coopr development, and Pyomo's support for abstract modeling has motivated a variety of Coopr extensions, including modeling extensions for stochastic programming and generalized disjunctive programming, as well as the Coopr Advanced Graphical Environment.

Coopr has also proven an effective framework for developing high-level optimization tools. For example, the PySP package provides generic solvers for stochastic programming. PySP leverages the fact that Pyomo's modeling objects are embedded within a full-featured, high-level programming language, which allows for transparent parallelization of sub-problems using Python parallel communication libraries.

Table C.1 describes the packages that are included in the Coopr 3.1 release. Coopr is decomposed into independent Python packages to leverage the fact that many packages are loosely coupled. Thus, independent Coopr packages can be developed and released separately. The `coopr` package represents the integration of Coopr packages; this package defines which Coopr packages are included in the overall Coopr releases.

Core Coopr Packages	
coopr	The Coopr package that defined the overall Coopr releases
coopr.opt	Core infrastructure for optimization solvers and results
coopr.pyomo	A Python algebraic modeling tool
coopr.sucasa	A tool exposing algebraic modeling structure to integer programming solvers
Coopr Extension Packages	
coopr.age	A graphical interface for Coopr and Pyomo
coopr.gdp	Pyomo modeling extensions for generalized disjunctive programming
coopr.openopt	Extensions that allow Pyomo models to be optimized by OpenOpt solvers
coopr.os	Extensions that allow Pyomo models to be optimized by OS solvers
coopr.pysp	Pyomo modeling extensions and solvers for stochastic programming
coopr.plugins	Various plugins (solvers, converters, etc)
Coopr Data Packages	
coopr.data.cute	CUTE test problems formulated as Pyomo models
coopr.data.pyomo	Pyomo models used to test solvers
coopr.data.pysp	Large stochastic programming test problems
coopr.data.samples	Various example scripts and test problems

Table C.1 The Python packages that are included in the Coopr 3.1 release.

C.2 Optimization Solvers

Pyomo models can be analyzed with a wide array of optimization solvers. The `coopr.opt` package manages interfaces to optimization solvers, which are typically defined outside of Coopr. Table C.2 summarizes the solver interfaces that are supported in Coopr 3.1. These interfaces allow a user to specify the type of I/O that will be used to perform optimization. The following I/O types are used by Coopr solver interfaces:

- **lp**: The solver optimizes an LP file that is generated by Pyomo, and the logfile and other output files generated by the solver and parsed to create the solver results object.
- **nl**: The solver optimizes an NL file that is generated by Pyomo, and the SOL file generated by the solver is process to create the solver results object.
- **os**: The solver optimizes an OSiL file that is generated by Pyomo, and the OSrL file generated by the solver is processed to create the solver results object.
- **python**: Coopr directly interacts with the solver through a Python interface. No explicit file I/O is required.

The `python` I/O type is generally more efficient because it avoids file I/O, and a lot of detail about the optimization process can be collected from the solver. However, this requires the installation of a Python interface for the solver. The `lp` I/O type can also provide a lot of detail about the optimization process, though the logfile parsing can be fragile. The `nl` I/O type allows Coopr to leverage any solver that has been built with the AMPL Solver Library. Thus, Pyomo models can be optimized with any optimizer that can be used with the AMPL modeling software! However, this I/O type provides limited detail about the optimization process. The `os` I/O

solver	I/O type				Description
	lp	nl	os	py	
cbc	x	x	x		The CBC LP/MIP solver
cplex	x	x	x	x	The CPLEX LP/MIP solver
glpk	x		x	x	The GLPK LP/MIP solver
gurobi	x	x	x	x	The GUROBI LP/MIP solver
pico	x	x	x		The PICO LP/MIP solver

Table C.2 Predefined solver interfaces supported by Coopr, with a summary of the I/O types that they support.

type also provides a lot detail about the optimization process, and it employ a robust XML specification. However, this capability remains experimental in Coopr 3.1.

Aside from predefined solver interfaces, Coopr can dynamically create solver interfaces for any executable that is specified on the user's path. If an unknown solver interface is specified and an executable is found matching that name, then Coopr creates a solver interface with the `nl` I/O type. Thus, AMPL-enabled optimizers like `ipopt` can be specified without requiring a custom interface in Pyomo.

The `--solver-io` option for the `pyomo` command can also be used to specify the `os` I/O type for a dynamically created solver interface. Note that Coopr does not attempt to verify that an appropriate I/O type is specified; the selected I/O simply defines how Coopr attempts to execute the solver using the specified executable.

References

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

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

[3] Antonio Alonso-Ayuso, Laureano F. Escudero, and M. Teresa Ortuno. BFC, a branch-and-fix coordination algorithmic framework for solving some types of stochastic pure and mixed 0-1 programs. *European Journal of Operational Research*, 151(3):503–519, 2003.

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

[5] Prasanth Anbalagan and Mladen Vouk. On reliability analysis of open source software - FEDORA. In *19th International Symposium on Software Reliability Engineering*, 2008.

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

[7] S. Bailey, D. Ho, D. Hobson, and SN Busenberg. Population dynamics of deer. *Mathematical Modelling*, 6(6):487–497, 1985.

[8] B.W. Bequette. *Process control: modeling, design, and simulation*. Prentice Hall, 2003.

[9] Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996.

[10] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific / Dynamic Ideas, 1997.

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

[12] J.R. Birge, M.A. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King, and S.W. Wallace. A standard input format for multiperiod stochastic linear program. *COAL (Math. Prog. Soc., Comm. on Algorithms) Newsletter*, 17:1–19, 1987.

[13] BSD. Open Source Initiative (OSI) - the BSD license. http://www.opensource.org/licenses/bsd-license.php, 2009.

[14] C.C. Caroe and R. Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24(1–2):37–45, 1999.

[15] COINOR. COIN-OR home page. http://www.coin-or.org, 2009.

[16] Forrester Consulting. Open source software's expanding role in the enterprise. http://www.unisys.com/eprise/main/admin/corporate/doc/Forrester_research-open_source_buying_behaviors.pdf, 2007.

[17] COOPR. Coopr: A common optimization python repository. http://software.sandia.gov/coopr, 2009.

[18] CPLEX. http://www.cplex.com, July 2010.

[19] T.G. Cranic, X. Fu, M. Gendreau, W. Rei, and S.W. Wallace. Progressive hedging-based meta-heuristics for stochastic network design. Technical Report CIRRELT-2009-03, University of Montreal CIRRELT, January 2009.

[20] G.B. Dantzig. Linear programming under uncertainty. *Management Science*, 1:197–206, 1955.

[21] Y. Fan and C. Liu. Solving stochastic transportation network protection problems using the progressive hedging-based method. *Networks and Spatial Economics*, 10(2):193–208, 2010.
[22] FLOPC++. FLOPC++ home page. https://projects.coin-or.org/FlopC++, 2008.
[23] 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.
[24] GAMS. GAMS home page. http://www.gams.com, 2008.
[25] H.I. Gassmann and E. Schweitzer. A comprehensive input format for stochastic linear programs. *Annals of Operations Research*, 104:89–125, 2001.
[26] D.M. Gay. Hooking your solver to ampl. *Numerical Analysis Manuscript*, pages 93–10, 1993.
[27] D.M. Gay. Writing. nl files, 2005.
[28] Arthur M. Geoffrion. An introduction to structured modeling. *Management Science*, 33(5):547–588, 1987.
[29] GLPK. GLPK: GNU linear programming toolkit. http://www.gnu.org/software/glpk/, 2009.
[30] Harvey J. Greenberg. A bibliography for the development of an intelligent mathematical programming system. *ITORMS*, 1(1), 1996.
[31] GUROBI. Gurobi optimization. http://www.gurobi.com, July 2010.
[32] William E. Hart and John D. Siirola. The PyUtilib component architecture. Technical report, Sandia National Laboratories, 2010.
[33] William E. Hart, Jean-Paul Watson, and David L. Woodruff. Pyomo: Modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, to appear, 2011.
[34] T. Helgason and S.W. Wallace. Approximate scenario solutions in the progressive hedging algorithm: A numerical study. *Annals of Operations Research*, 31(1–4):425–444, 1991.
[35] A. Holder, editor. *Mathematical Programming Glossary*. INFORMS Computing Society, http://glossary.computing.society.informs.org, 2006–11. Originally authored by Harvey J. Greenberg, 1999-2006.
[36] L.M. Hvattum and A. Løkketangen. Using scenario trees and progressive hedging for stochastic inventory routing problems. *Journal of Heuristics*, 15 (6):527–557, 2009.
[37] Ipopt. Ipopt home page. https://projects.coin-or.org/ipopt, 2008.
[38] S. Jorjani, C.H. Scott, and D.L. Woodruff. Selection of an optimal subset of sizes. *International Journal of Production Research*, 37(16):3697–3710, 1999.
[39] Peter Kall and Janos Mayer. *Stochastic Linear Programming: Models, Theory, and Computation*. Springer, 2005.
[40] Josef Kallrath. *Modeling Languages in Mathematical Optimization*. Kluwer Academic Publishers, 2004.
[41] O. Listes and R. Dekker. A scenario aggregation based approach for determining a robust airline fleet composition. *Transportation Science*, 39:367–382, 2005.

[42] A. Løkketangen and D. L. Woodruff. Progressive hedging and tabu search applied to mixed integer (0,1) multistage stochastic programming. *Journal of Heuristics*, 2:111–128, 1996.
[43] Roy E. Marsten. The design of the XMP linear programming library. *ACM Transactions on Mathematical Software*, 7(4):481–497, 1981.
[44] J. Nocedal and SJ Wright. Numerical optimization, series in operations research and financial engineering, 2006.
[45] OpenOpt. OpenOpt home page. http://scipy.org/scipy/scikits/wiki/OpenOpt, 2008.
[46] OptimJ. Ateji home page. http://www.ateji.com, 2008.
[47] PuLP. PuLP: A python linear programming modeler. http://130.216.209.237/engsci392/pulp/FrontPage, 2008.
[48] PyGlpk. PyGlpk: A python module which encapsulates GLPK. http://www.tfinley.net/software/pyglpk/, 2011.
[49] Pyipopt. Pyipopt home page. http://code.google.com/p/pyipopt/, 2008.
[50] PYRO. PYRO: Python remote objects. http://pyro.sourceforge.net, 2009.
[51] R.T. Rockafellar and R.J.-B. Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of Operations Research*, 16(1):119–147, 1991.
[52] Hermann Schichl. Models and the history of modeling. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, Dordrecht, Netherlands, 2004. Kluwer Academic Publishers.
[53] R. Schultz and S. Tiedemann. Conditional value-at-risk in stochastic programs with mixed-integer recourse. *Mathematical Programming*, 105(2–3):365–386, February 2005.
[54] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczynski. *Lectures on Stochastic Programming: Modeling and Theory*. Society for Industrial and Applied Mathematics, 2009.
[55] R.M. Van Slyke and R.J. Wets. L-shaped linear programs with applications to optimal control and stochastic programming. *SIAM Journal on Applied Mathematics*, 17:638–663, 1969.
[56] J. Thénié, Ch. van Delft, and J.-Ph. Vial. Automatic formulation of stochastic programs via an algebraic modeling language. *Computational Management Science*, 4(1):17–40, January 2007.
[57] TOMLAB. TOMLAB optimization environment. http://www.tomopt.com/tomlab, 2008.
[58] Stein W. Wallace and William T. Ziemba, editors. *Applications of Stochastic Programming*. Society for Industrial and Applied Mathematics, 2005.
[59] J.-P. Watson, D.L. Woodruff, and W.E. Hart. Pysp: Modeling and solving stochastic programs in python. *Mathematical Programming Computation*, to appear, 2011.
[60] J.P. Watson and D.L. Woodruff. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science (to appear)*, 2010.

[61] D.L. Woodruff and E. Zemel. Hashing vectors for tabu search. *Annals of Operations Research*, 41(2):123–137, 1993.
[62] YAML. The official YAML web site. http://yaml.org/, 2009.
[63] Ying Zhou and Joseph Davis. Open source software reliability model: An empirical approach. *ACM SIGSOFT Software Engineering Notes*, 30:1–6, 2005.

Index