William E. Hart • Carl D. Laird
Jean-Paul Watson • David L. Woodruff
Gabriel A. Hackebeil • Bethany L. Nicholson
John D. Siirola

Pyomo - Optimization Modeling in Python

Second Edition

Springer

William E. Hart
Sandia National Laboratories
Albuquerque, New Mexico, USA

Jean-Paul Watson
Sandia National Laboratories
Albuquerque, New Mexico, USA

Gabriel A. Hackebeil
Department of Industrial
and Operations Engineering
University of Michigan
Ann Arbor, Michigan, USA

John D. Siirola
Sandia National Laboratories
Albuquerque, New Mexico, USA

Carl D. Laird
Sandia National Laboratories
Albuquerque, New Mexico, USA

David L. Woodruff
Graduate School of Management
University of California, Davis
Davis, California, USA

Bethany L. Nicholson
Sandia National Laboratories
Albuquerque, New Mexico, USA

ISSN 1931-6828 ISSN 1931-6836 (electronic)
Springer Optimization and its Applications
ISBN 978-3-319-58819-3 ISBN 978-3-319-58821-6 (eBook)
DOI 10.1007/978-3-319-58821-6

Library of Congress Control Number: 2017940404

Preface

This book describes a 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 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 [53]. AMLs streamline the process of formulating 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 the custom analysis of complex problems.

The core of Pyomo is an object-oriented capability for representing optimization models. Pyomo also contains packages that define modeling extensions and model reformulations. For example, the `pyomo.pysp` package defines modeling extensions for stochastic programs as well as solvers that can analyze these problems. Pyomo also includes packages that define interfaces to solvers like CPLEX and Gurobi, as well as solver services like NEOS.

Goals of the Book

This second edition provides an updated description of Pyomo's modeling capabilities. A key goal of this book is to provide a broad description of Pyomo that will enable the user to develop and optimize models with Pyomo. The book uses many examples to illustrate different techniques that can be used to formulate models.

Another goal of this book is to illustrate the breadth of Pyomo's capabilities. Pyomo supports the formulation and analysis of common optimization models, including linear programs, mixed-integer linear programs, nonlinear programs, mixed-integer nonlinear programs, mathematical programs with equilibrium constraints, generalized disjunctive programs, bilevel programs, and stochastic programs. Additionally, Pyomo includes solver interfaces for a variety of widely used optimization software packages, including CBC, CPLEX, GLPK, and Gurobi. Additionally, Pyomo models can be optimized with optimizers like IPOPT that employ the AMPL Solver Library interface.

Finally, a goal of this book is to help users get started with Pyomo even if they have little knowledge of Python. Appendix A provides a quick introduction to Python, but we have been impressed with how well 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.

However, 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 discussion of modeling components distinguishes between class definitions and class instances. 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.

Who Should Read This Book

This book provides 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 [45] as a reference for the reader.

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. Consequently, issues like run-time performance and robust solver interfaces are a priority.

Additionally, we believe that researchers will find that Pyomo provides an effective framework for developing high-level optimization and analysis tools. For example, Pyomo 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.

Revisions for the Second Edition

We have made several major changes while preparing the second edition of this book. The book was divided into two parts: (1) chapters that provide an introduction to optimization and Pyomo, and (2) chapters that describe advanced features and extensions. The introductory chapters were revised to provide a more tutorial description. In particular, reference material was removed from the first edition, which will be provided online at the Pyomo website. The chapters describing advanced features were extended to include new functionality added to Pyomo since the first edition, including generalized disjunctive programming, mathematical programming with equilibrium constraints, and bilevel programming.

Comments and Questions

This book documents the capabilities of the Pyomo 5.1 release. Most examples in the book work with Pyomo 5.0, but some errors in Pyomo DAT file processing were resolved in the Pyomo 5.1 release. Further information is available on the Pyomo website:

```
http://www.pyomo.org
```

Pyomo's open source software is hosted at GitHub:

```
https://github.com/Pyomo/pyomo
```

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

```
pyomo-forum@googlegroups.com
```

We hope this will include feedback on the presentation of this material, including typos and errors in our examples. Note that all of the examples used in this book are included with Pyomo in the `pyomo/examples/doc/pyomobook` directory!

Good Luck!

Albuquerque, New Mexico, USA — *William Hart*
Albuquerque, New Mexico, USA — *Carl Laird*
Albuquerque, New Mexico, USA — *Jean-Paul Watson*
Davis, California, USA — *David Woodruff*
Ann Arbor, Michigan, USA — *Gabe Hackebeil*
Albuquerque, New Mexico, USA — *Bethany Nicholson*
Albuquerque, New Mexico, USA — *John Siirola*
7 April, 2017

Contents

Chapter 1
Introduction

Abstract This chapter introduces and motivates Pyomo, a Python-based tool for modeling and solving optimization problems. Modeling is a fundamental process in many aspects of scientific research, engineering, and business. Algebraic modeling languages like Pyomo are high-level languages for specifying and solving mathematical optimization problems. Pyomo is a flexible, extensible modeling framework that captures and extends central ideas found in modern algebraic modeling languages, all within the context of a widely used programming language.

1.1 Modeling Languages for Optimization

This book describes a tool for mathematical modeling: the Python Optimization Modeling Objects (Pyomo) software package. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with commercial algebraic modeling languages (AMLs) such as AIMMS [1], AMPL [2], and GAMS [31]. Pyomo implements a rich set of modeling and analysis capabilities, and it provides access to these capabilities within Python, a full-featured, high-level programming language with a large set of supporting libraries.

Optimization models define the goals or objectives for a system under consideration. Optimization models can be used to explore trade-offs between goals and objectives, identify extreme states and worst-case scenarios, and identify key factors that influence phenomena in a system. Consequently, optimization models are used to analyze a wide range of scientific, business, and engineering applications.

The widespread availability of computing resources has made the numerical analysis of optimization models commonplace. The computational analysis of an optimization model requires the specification of a model that is communicated to a solver software package. Without a language to specify optimization models, the process of writing input files, executing a solver, and extracting results from a solver is tedious and error-prone. This difficulty is compounded in complex, large-scale,

real-world applications that are difficult to debug when errors occur. Additionally, solvers use many different input formats, but few of them are considered to be standards. Thus, the application of multiple solvers to analyze a single optimization model introduces additional complexities. Furthermore, model verification (i.e., ensuring that the model communicated to the solver accurately reflects the model the developer intended to express) is extremely difficult without high-level languages for expressing models.

AMLs are high-level languages for describing and solving optimization problems [36, 53]. AMLs minimize the difficulties associated with analyzing optimization models by enabling high-level specification of optimization problems. Furthermore, AML software provides rigorous interfaces to external solver packages that are used to analyze problems, and it allows the user to interact with solver results in the context of their high-level model specification.

Custom AMLs like AIMMS [1], AMPL [2, 29], and GAMS [31] implement optimization model specification languages with an intuitive and concise syntax for defining variables, constraints, and objectives. Further, these AMLs support specification of abstract concepts such as sparse sets, indices, and algebraic expressions, which are essential when specifying 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 enables expression of high-level analysis algorithms concurrently with optimization model specifications. Similarly, standard programming languages like Java and C++ have been extended to include AML constructs. For example, modeling libraries like FlopC++ [27] and OptimJ [67] support the specification of optimization models using an object-oriented design in C++ and Java, respectively. Although these modeling libraries sacrifice some of the intuitive mathematical syntax of a custom AML, they allow the user to leverage the flexibility of modern high-level programming languages. A further advantage of these AML libraries is that they can link directly to high-performance optimization libraries and solvers, which can be an important consideration in some applications.

A complementary strategy is to use an AML that extends a *standard* high-level programming language (as opposed to being based a proprietary language) to formulate optimization models that are analyzed with solvers 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 is an increasingly common approach for scientific computing software. The Matlab TOMLAB Optimization Environment [83] is among the most mature optimization software package using this approach; Pyomo strongly leverages this approach as well.

1.2 Modeling with Pyomo

The goal of Pyomo is to provide a platform for specifying optimization models that embodies central ideas found in modern AMLs, within a framework that promotes flexibility, extensibility, portability, openness, and maintainability. Pyomo is an AML that extends Python to include objects for optimization modeling [42]. These objects can be used to specify optimization models and translate them into various formats that can be processed by external solvers.

We now provide some motivating examples to illustrate the use of Pyomo in specifying optimization models.

1.2.1 Simple Examples

Consider the following linear program (LP):

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

This LP can be easily expressed in Pyomo as follows:

```
from pyomo.environ 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)
```

The first line is a standard Python import statement that initializes the Pyomo environment and loads Pyomo's core modeling component library. The next lines construct a model object and define model attributes. This example describes a *concrete* model. Model components are objects that are attributes of a model object, and the `ConcreteModel` object initializes each model component as they are added. The model decision variables, constraints, and objective are defined using Pyomo *model components*.

Users rarely have a single instance of a particular optimization problem to solve. Rather, they commonly have a general optimization model and then create a particular instance of that model using specific data. For example, the following equations represent an LP with scalar parameters n and m, vector parameters b and c, and matrix parameter a:

$$\begin{aligned} \min\ & \textstyle\sum_{i=1}^{n} c_i x_i \\ \text{s.t.}\ & \textstyle\sum_{i=1}^{n} a_{ji} x_i \geq b_j \quad \forall j = 1 \ldots m \\ & x_i \geq 0 \qquad\qquad\quad \forall i = 1 \ldots n \end{aligned}$$

This LP can be expressed with a concrete model in Pyomo as follows:

```
from pyomo.environ import *
import mydata

model = ConcreteModel()

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

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

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

This script requires that the data used to construct the model is available while each modeling component is constructed. In this example, the necessary data exists in `mydata.py`:

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

This LP can also be viewed as an *abstract* mathematical model, where unspecified, symbolic parameter values are later defined when the model is initialized. For example, this LP can be expressed as an abstract model in Pyomo as follows:

```
from pyomo.environ import *

model = AbstractModel()

model.N = Set()
model.M = Set()
model.c = Param(model.N)
model.a = Param(model.M, model.N)
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[m,i]*model.x[i] for i in model.N) \
               >= model.b[m]
model.con = Constraint(model.M, rule=con_rule)
```

This example includes model components that provide abstract or symbolic definitions of set and parameter values. The `AbstractModel` object defers initial-

ization of model components until a *model instance* is created, using user-supplied set and parameter data. Both concrete and abstract models can be initialized with data from a variety of different data sources, including data command files that are adapted from AMPL's data commands. For example:

```
param : N : c :=
1 1
2 2 ;

param : M : b :=
1 1
2 2 ;

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

1.2.2 Graph Coloring Example

We further illustrate Pyomo's modeling capabilities with a simple, well-known optimization problem: minimum graph coloring (also known as vertex coloring). The graph coloring problem concerns the assignment of colors to vertices of a graph such that no two adjacent vertices share the same color. Graph coloring has many practical applications, including register allocation in compilers, resource scheduling, and pattern matching, and it appears as a kernel in recreational puzzles like Sudoku.

Let $G = (V,E)$ denote a graph with vertex set V and edge set $E \subseteq V \times V$. Given G, the objective in the minimum graph coloring problem is to find a valid coloring that uses the minimum number of distinct colors. For simplicity, we assume that the edges in E are ordered such that if $(v_1, v_2) \in E$ then $v_1 < v_2$. Let k denote the maximum number of colors, and define the set of possible colors $C = \{1, \ldots, k\}$.

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

$$\begin{array}{lll} \min & y & \\ \text{s.t.} & \sum_{c \in C} x_{v,c} = 1 & \forall v \in V \\ & x_{v_1,c} + x_{v_2,c} \leq 1 & \forall (v_1, v_2) \in E \\ & y \geq c \cdot x_{v,c} & \forall v \in V, c \in C \\ & x_{v,c} \in \{0,1\} & \forall v \in V, c \in C \end{array} \tag{1.1}$$

In this formulation, the variable $x_{v,c}$ equals one if vertex v is colored with color c and zero otherwise; y denotes the number of colors that are used. The first constraint requires that each vertex is colored with exactly one color. The second constraint requires that vertices that are connected by an edge must have different colors. The third constraint defines a lower bound on y that guarantees that y will be no less than

the number of colors used in a solution. The fourth and final constraint enforces the binary constraint on $x_{v,c}$.

Figure 1.1 shows a Pyomo specification of the above graph coloring formulation, using a concrete model; the example is adapted from Gross and Yellen [37]. This specification consists of Python commands that define a `ConcreteModel` object, and then define various attributes of this object, including variables, constraints, and the optimization objective. Lines 10–24 define the model data. Line 28 is a standard Python import statement that adds all of the symbols (e.g., classes and functions) defined in `pyomo.environ` to the current Python namespace. Line 31 specifies creation of the `model` object, which is an instance of the `ConcreteModel` class. Lines 34 and 35 define the model decision variables. Note that y is a scalar variable, while x is a two-dimensional array of variables. The remaining lines in the example define the model constraints and objective. The `Objective` class defines a single optimization objective using the `expr` keyword option. The `ConstraintList` class defines a list of constraints, which are individually added.

When compared to custom AMLs, Pyomo models are clearly more verbose (e.g., see Hart et al. [42]). However, this example illustrates how Python's clean syntax still allows Pyomo to express mathematical concepts intuitively and concisely. Aside from the use of Pyomo classes, this example employs standard Python syntax and methods. For example, line 41 uses 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.

1.2.3 Key Pyomo Features

Python

Python's clean syntax enables Pyomo to express mathematical concepts in an intuitive and concise manner. Furthermore, Python's expressive programming environment can be used to formulate complex models and to define high-level solvers that customize the execution of high-performance optimization libraries. Python provides extensive scripting capabilities, allowing users to analyze Pyomo models and solutions, leveraging Python's rich set of third-party libraries (e.g., numpy, scipy, and matplotlib). Finally, the embedding of Pyomo in Python allows users to learn core syntax through Python's rich documentation.

Customizable Capability

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 model components, model transformations,

```
#
# Graph coloring example adapted from
#
#  Jonathan L. Gross and Jay Yellen,
#  "Graph Theory and Its Applications, 2nd Edition",
#  Chapman & Hall/CRC, Boca Raon, FL, 2006.
#

# Define data for the graph of interest.
vertices = set(['Ar', 'Bo', 'Br', 'Ch', 'Co', 'Ec',
                'FG', 'Gu', 'Pa', 'Pe', 'Su', 'Ur', 'Ve'])

edges = set([('FG','Su'), ('FG','Br'), ('Su','Gu'),
             ('Su','Br'), ('Gu','Ve'), ('Gu','Br'),
             ('Ve','Co'), ('Ve','Br'), ('Co','Ec'),
             ('Co','Pe'), ('Co','Br'), ('Ec','Pe'),
             ('Pe','Ch'), ('Pe','Bo'), ('Pe','Br'),
             ('Ch','Ar'), ('Ch','Bo'), ('Ar','Ur'),
             ('Ar','Br'), ('Ar','Pa'), ('Ar','Bo'),
             ('Ur','Br'), ('Bo','Pa'), ('Bo','Br'),
             ('Pa','Br')])

ncolors = 4
colors = range(1, ncolors+1)


# Python import statement
from pyomo.environ import *

# Create a Pyomo model object
model = ConcreteModel()

# Define model variables
model.x = Var(vertices, colors, within=Binary)
model.y = Var()

# Each node is colored with one color
model.node_coloring = ConstraintList()
for v in vertices:
    model.node_coloring.add(
                sum(model.x[v,c] for c in colors) == 1)

# Nodes that share an edge cannot be colored the same
model.edge_coloring = ConstraintList()
for v,w in edges:
    for c in colors:
        model.edge_coloring.add(
                model.x[v,c] + model.x[w,c] <= 1)

# Provide a lower bound on the minimum number of colors
# that are needed
model.min_coloring = ConstraintList()
for v in vertices:
    for c in colors:
        model.min_coloring.add(
                model.y >= c * model.x[v,c])

# Minimize the number of colors that are needed
model.obj = Objective(expr=model.y)
```

Fig. 1.1: A concrete Pyomo model for the minimum graph coloring problem.

solvers, and solver managers. A plug-in framework manages the registration of these capabilities. Thus, users can customize Pyomo in a modular manner without the risk of destabilizing core functionality.

Command-Line Tools and Scripting

Pyomo models can be analyzed both using command-line tools and via Python scripts. The `pyomo` command line utility provides a generic interface to most Pyomo modeling capabilities. An exception is the stochastic programming capabilities available in `pyomo.pysp`. The `pyomo` command supports a generic optimization process. This process can easily be replicated in a Python script and further customized for a user's specific needs.

Concrete and Abstract Model Definitions

The examples in Section 1.2.1 illustrate Pyomo's support for both concrete and abstract model definitions. The difference between these modeling approaches relates to when modeling components are initialized: concrete models immediately initialize components, and abstract models delay the initialization of components until a later model initialization action. Consequently, these modeling approaches are equivalent, and the choice of approach depends on the context in which Pyomo is used and user preference. Both types of models can be easily initialized with Pyomo's `DataPortal` class, which can load data from a wide range of data sources (e.g., csv, json, yaml, excel, and databases).

Object-Oriented Design

Pyomo employs an object-oriented library design. Models are Python objects, and model components are attributes of these models. This design allows Pyomo to automatically manage the naming of modeling components, and it naturally segregates modeling components within different model objects. Pyomo models can be further structured with blocks, which support a hierarchical nesting of model components. Many of Pyomo's advanced modeling features leverage this structured modeling capability.

Expressive Modeling Capability

Pyomo's modeling components can be used to express a wide range of optimization problems, including but not limited to:

- linear programs,
- quadratic programs,

- nonlinear programs,
- mixed-integer linear programs,
- mixed-integer quadratic programs,
- generalized disjunctive programs,
- mixed-integer stochastic programs,
- dynamic problems with differential algebraic equations,
- mathematical programs with equilibrium constraints, and
- bilevel programs.

Solver Integration

Pyomo supports both tightly and loosely coupled solver interfaces. Tightly coupled modeling tools directly access optimization solver libraries (e.g., via static or dynamic linking), and loosely coupled modeling tools apply external optimization executables (e.g., through the use of system calls). Many optimization solvers read problems from well-known data formats (e.g., the AMPL `nl` format [34]); these solvers are loosely coupled with Pyomo. Solvers with Python interfaces (e.g., Gurobi and CPLEX) can be tightly coupled, which avoids writing external files.

Open Source

Pyomo is managed as an open source project to facilitate transparency in software design and implementation. Pyomo is licensed under the BSD license [12], which has few restrictions on government or commercial use. Pyomo is managed at GitHub [73], and through the COIN-OR project [14]. Developer and user mailing lists are managed on Google Groups. There is growing evidence that the reliability of open source software is similar to closed source software [3, 89], and Pyomo is carefully managed to ensure the robustness and reliability for users.

1.3 Getting Started

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

- Python 2.6, 2.7, 3.4, and 3.5. Pyomo currently relies on CPython; there is only experimental support for Jython and PyPy for a subset of Pyomo's capability.
- Pyomo 5.1, which is used throughout this book.
- The `PyYAML` package, which is used to generate human readable display of solver results in the YAML format. Note that this package is not strictly necessary when using Pyomo; the default output format for solver results is JSON.
- The `GLPK` [35] solver, which 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 and widely available.

- The `IPOPT` [48] solver, which is used to generate output for nonlinear model examples. Other nonlinear optimizers can be easily used for these examples if they are compiled with the AMPL Solver Library [33].
- The `CPLEX` [16] solver, which is used to generate output for stochastic programming examples. This commercial solver provides capabilities needed for these examples that are not commonly available in open source optimization solvers (e.g., optimization of quadratic integer programs).
- The `matplotlib` Python package, which is used to generate plots.

Installation instructions for Pyomo are provided at the Pyomo website: `www.pyomo.org`. Appendix A provides a brief tutorial of the Python scripting language; various on-line sources provide more comprehensive tutorials and documentation.

NOTE: Most examples in the book work with Pyomo 5.0, but some errors in Pyomo DAT file processing were resolved in the Pyomo 5.1 release.

1.4 Book Summary

The remainder of this book is divided into two parts. The first part provides an introduction to Pyomo. Chapter 2 provides a primer on optimization and mathematical modeling, including brief illustrations of how Pyomo can be used to specify and solve algebraic optimization models. Chapter 3 illustrates Pyomo's modeling capabilities with simple concrete and abstract models, and Chapter 4 describes Pyomo's core modeling components. Chapter 5 describes Pyomo's command-line interface. Finally, Chapter 6 describes the syntax of Pyomo data command files.

The second part of this book documents advanced features and extensions. Chapter 7 describes the nonlinear programming capabilities of Pyomo, and Chapter 8 describes how hierarchical models can be expressed in Pyomo. The next chapters describe modeling extensions: generalized disjunctive programming (Chapter 9), stochastic programming (Chapter 10), dynamic models expressed with differential and algebraic equations (Chapter 11), programs with equilibrium constraints (Chapter 12), and bilevel programming (Chapter 13). Finally, Chapter 14 illustrates the use of Python scripts to customize the optimization process.

NOTE: This book does not provide a *complete* reference for Pyomo. Instead, our goal is to discuss all of Pyomo's stable core functionality that is available in the Pyomo 5.1 release.

1.5 Discussion

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 [42]. A variety of open source software packages provide optimization modeling capabilities in Python, such as PuLP [69], APLEpy [4], and OpenOpt [66]. Additionally, there are many Python-based solver interface packages, including open source packages such as PyGlpk [70] and pyipopt [71], in addition to Python interfaces for the commercial solvers CPLEX [16] and Gurobi [39].

Several features distinguish Pyomo from these other Python-based optimization modeling tools. First, Pyomo 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 significant flexibility in 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, stochastic programs, bilevel programs, generalized disjunctive programs, problems constrained by differential equations, and mathematical programs with equilibrium conditions.

Part I
An Introduction to Pyomo

Chapter 2
Mathematical Modeling and Optimization

Abstract This chapter provides a primer on optimization and mathematical modeling. It does not provide a complete description of these topics. Instead, this chapter provides enough background information to support reading the rest of the book. For more discussion of optimization modeling techniques see, for example, Williams [86]. Implementations of simple examples of models are shown to provide the reader with a quick start to using Pyomo.

2.1 Mathematical Modeling

2.1.1 Overview

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. Schichl [78] notes that models are used 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.

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

Mathematics has always played a fundamental role in representing and formulating our knowledge. Mathematical modeling has become increasingly formal as new

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., which decisions to take, or the characteristic of a system outcome).
- **Parameters**: These are symbolic representations for real-world data, which might vary for different problem instances or scenarios.
- **Relations**: These are *equations*, *inequalities*, or other mathematical relationships that define how different parts of a model are related 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.

2.1.2 A Modeling Example

A *Model*, in the sense that we will use the word, represents items by abstracting away some features. Everyone is familiar with physical models, such as model railroads or model cars. Our interest is in mathematical models that use symbols to represent aspects of a system or real-world object.

For example, a person might want to determine the best number of scoops of ice cream to buy. We could use the symbol x to represent the number of scoops. We might use c to represent the cost per scoop. So then we could model the total cost as c times x, which we usually write as cx.

We might need a more sophisticated model of total cost if there are volume discounts or surcharges for buying fractional scoops. Also, this model is probably not valid for negative values of x. It is seldom possible to sell back ice cream for the same price paid for it.

It is more complicated to provide a mathematical model of the happiness associated with scoops of ice cream on an ice cream cone. One approach is to use a scaled measure of happiness. We will do that using the basic unit of the happiness associated with one scoop of ice cream, which we call h. A simple model, then, would be to say that the total happiness from x scoops of ice cream is h times x, which we write as hx. For some people, that might be a pretty good approximation for values of x between one-half and three, but there is almost no one who is 100 times as happy to have 100 scoops of ice cream on their ice cream cone as they are to have one scoop. For some people, the model of happiness for values of x between zero and ten might be something like

$$h \cdot \left(x - (x/5)^2\right).$$

Note that this model becomes negative when there are more than 25 scoops on the cone, which might not be a good model for everyone.

It is common to want to model more than one thing at a time. For example, you might be able to have scoops of ice cream and peanuts. Since there are multiple things that can be purchased, we can represent the quantity purchased using a vector x (i.e. the symbol x now represents a list). We refer to *elements* of the list using the notation x_i where the symbol i indexes the vector. For example, if we agree that the first element is the number of scoops of ice cream, then this number could be referenced using x_1. For higher dimensions we use a *tuple*, such as i, j or (i, j) as the index.

Let's change c to be a vector of costs with the same indices as x (i.e., c_1 is the cost per scoop of ice cream and c_2 is the cost per cup of peanuts). So now, we write the total cost of ice cream and peanuts as

$$c_1x_1 + c_2x_2 = \sum_{i=1}^{2} c_ix_i.$$

Once again, this cost model is probably not valid for all possible values of all elements of x, but it might be good enough for some purposes.

Often, it is useful to refer to indices as being members of a set. For the example just given, we could use the set $\{1,2\}$ to write the total cost as

$$\sum_{i\in\{1,2\}} c_ix_i.$$

but it would be more common to use a more abstract expression like

$$\sum_{i\in\mathscr{A}} c_ix_i$$

where the set $\mathscr{A}$ is understood to be the index set for c and x (and for our example the set $\mathscr{A}$ would be $\{1,2\}$.)

In addition to summing over an index set, we might want to have conditions that hold for all members of an index set. This is done simply by using a comma. For example, if we want to require that none of the values of x can be negative, we would write

$$x_i \geq 0,\ i \in \mathscr{A}$$

and we read this line out loud as "x subscript i is greater than or equal to zero for all i in A."

There is no law of mathematics or even mathematical modeling that requires the use of single letter symbols such as x and c or i. It would be perfectly okay for the set $\mathscr{A}$ to be composed of a picture of an ice cream cone and a picture of a cup of peanuts, but that is hard to work with in some settings. The set could also be $\{Scoops, Cups\}$, but that is not commonly done in books because it takes up too much space and causes lines to overflow. Also, x could be replaced by something like $Quantity$. Long names are, importantly, supported by modeling languages such as Pyomo, and it is generally a good idea to use meaningful names when writing Pyomo models. Spaces or dashes embedded in names often cause troubles and confusion, so underscores

are often used in long names instead.

2.2 Optimization

The symbol x is often used as a *variable* in optimization modeling. It is sometimes called a *decision variable* because we build optimization models to help make decisions. This can sometimes cause a little confusion for people who are familiar with modeling as practiced by statisticians. They often use the symbol x to refer to data. Thus statisticians give values of x to the computer to have it compute statistics, while optimization modelers give other data to the computer and ask the computer to compute good values of x. Of course, symbols other than x can be used; however, in text books and introductions x is often chosen.

Values such as cost (we used the symbol c) are referred to as *data* or *parameters*. An optimization model can be described with undefined parameter values, but a specific instance that is optimized must have specific data values, which we sometimes call *instance data*.

A model must have an objective to perform optimization, which is expressed as an *objective function*. Optimal values of the decision variables result in *the* best possible value of the objective function. It is important to note that we did not say "the optimal values" because it is often the case that more than one set of variable values result in the best possible value of the objective function. It is common to write this function in a very abstract way, such as $f(x)$. Whether the best is the smallest or the largest possible value is determined by the *sense* of the optimization: *minimize* or *maximize*.

For example, suppose that x is not a vector, but rather a *scalar* that denotes the number of scoops of ice cream to buy. If we use the model of happiness given before, then

$$f(x) \equiv h \cdot \left(x - (x/5)^2\right),$$

where h is given as data. (It turns out not to matter what value of h is given for the purpose of finding the x that maximizes happiness in this particular example.) The optimization problem, as we have modeled it, is given as

$$\max\ h \cdot \left(x - (x/5)^2\right),$$

but very careful authors would write

$$\max_x\ h \cdot \left(x - (x/5)^2\right)$$

to make it clear that x is the decision variable. In this case, there is only one best value of x, which can be found using numerical optimization. The best value of x turns out to be fractional, which means that it is not an integer number of scoops. This model might not be considered useful for a typical ice cream shop, where the number of scoops must be a non-negative integer. To specify this requirement, we

add a *constraint* to the optimization model:

$$\begin{aligned} \max_x \; & h \cdot \left(x - (x/5)^2\right) \\ \text{s.t.} \; & \\ & x \in \text{non-negative integers} \end{aligned}$$

where "s.t." is an abbreviation for either "subject to" or "such that." Suppose that the model is not being used in an ice cream shop, but rather at home, where the ice cream is being served by the model user's parent. If the parent is willing to make partial scoops but not willing to go above two scoops, then the constraint

$$x \in \text{non-negative integers}$$

would be replaced with

$$0 \leq x \leq 2.$$

This is not a perfect model because really, not all fractional values of x would be reasonable.

To illustrate the model aspects discussed so far, let us return to multiple products described by an index set $\mathscr{A}$, so x is a vector. Let us make use of the following model of happiness for a product index i:

$$h_i \cdot \left(x_i - (x_i/d_i)^2\right),$$

where h and d are both data vectors with the same index set as x. Further, let c be a vector of costs and u be a vector of the most of any product that can be purchased. Let us assume that all products can be purchased in fractional quantities for the moment. Finally, suppose there is a total budget given by b. The optimization problem would be written as:

$$\begin{aligned} \max_x \; & \sum_{i \in \mathscr{A}} h_i \cdot \left(x_i - (x_i/d_i)^2\right) \quad (H) \\ \text{s.t.} \; & \sum_{i \in \mathscr{A}} c_i x_i \leq b \\ & 0 \leq x_i \leq u_i, \;\; i \in \mathscr{A} \end{aligned}$$

Some modelers would express the last constraint separately:

$$\begin{aligned} \max_x \; & \sum_{i \in \mathscr{A}} h_i \cdot \left(x_i - (x_i/d_i)^2\right) \quad (H) \\ \text{s.t.} \; & \sum_{i \in \mathscr{A}} c_i x_i \leq b \\ & x_i \leq u_i, i \in \mathscr{A} \\ & x_i \geq 0, i \in \mathscr{A} \end{aligned}$$

It is common to put a short, abbreviated name of the model in parentheses on the same line as the objective. The name (P) is very common, but we used (H) as a mnemonic for "happiness." The name (H) allows us to refer to this model later in the chapter, where we show how to implement it in Pyomo and solve it.

2.3 Linear and Nonlinear Optimization Models

2.3.1 Definition

An expression in an optimization model is said to be linear if it is composed only of sums of decision variables and/or decision variables multiplied by data. Thus, a linear expression is an expression that is a non-constant, linear function of the decision variables. Assume that x is a variable vector, c is a vector of data and that both are indexed by $\mathscr{A}$. Further assume that 2 and 3 are members of $\mathscr{A}$. The following are linear expressions:

$$\begin{array}{l}\sum_{i\in\mathscr{A}} c_i x_i \\ \sum_{i\in\mathscr{A}} x_i \\ x_2 \\ c_3 x_2 + c_2 x_3 \\ c_3 x_2 + c_2 x_3 + 4\end{array}$$

On the other hand, the following expressions are not linear: x_i^2, $x_2 x_3$ and cosine(x_2).

Linear expressions often result in problems that can be solved with much less computational effort than similar models with nonlinear expressions. Consequently, many modelers make an effort to use linear expressions as much as possible, and some modelers strive to use only linear expressions. Additionally, many models develop linear approximations to nonlinear models in hopes of finding "good enough" solutions to the original nonlinear model. In the context of nonlinear models, those with nonlinear objective functions are typically easier to optimize than models with nonlinear constraint expressions.

2.3.2 A Linear Approximation

We consider a simple way to linearize (H) to illustrate a linear optimization model. We describe a version of (H) that does not have a squared objective, but that is somewhat similar to (H).

For the moment, suppose that x is not an indexed list. We call x a *scalar* value. The nonlinearities lie in the expression

$$h \cdot \left(x - (x/5)^2\right),$$

and in (H) we generalize this to be of the form:

$$h \cdot \left(x - (x/d)^2\right).$$

To simplify matters, let us require x to be non-negative and less than or equal to u. We can form a linear expression that has the function value correctly computed at the endpoints, namely zero and u. The linear function will connect these points with a line. This expression is zero when x is zero, and it is $h \cdot \left(u - (u/d)^2\right)$ when x=u. The slope of the line between these two points is

$$\frac{h \cdot \left(u - (u/d)^2\right) - 0}{u - 0} = h \cdot \left(1 - u/d^2\right).$$

So a simple approximation for $h \cdot \left(x - (x/d)^2\right)$ on the interval $[0, u]$ is

$$h \cdot \left(1 - u/d^2\right) x.$$

As can be seen in Figure 2.1, the linear approximation is quite good for the ice cream example when $h = 1$, $d = 5$ and $u = 5$. In contrast, this simple method would work very poorly if $u = 25$ for the same value of d.

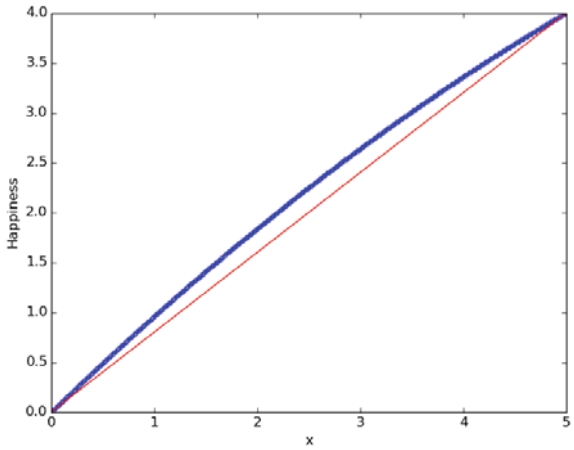


Fig. 2.1: Plotting the quadratic happiness function (the thicker line) and the linear approximation (the thin line). Both lines are drawn for $h = 1$, $d = 5$, and $u = 5$.

For illustrative purposes, to construct an expedient linear approximation to (H) we replace the objective function with

$$\max_x \sum_{i \in \mathscr{A}} h_i \cdot \left(1 - u/d_i^2\right) x_i \tag{2.1}$$

We say that this expression is linear because the decision variables are only multiplied by data, and summed. It is true that the parameter d is squared, but this is not a decision variable. The numerical value of the entire expression

$$h_i \cdot \left(1 - u_i / d_i^2\right)$$

is computed by Pyomo before the problem instance is passed to a solver, and the task of the solver is to find values that are optimal for the decision variables.

2.4 Modeling with Pyomo

We now consider different strategies for formulating and optimizing algebraic optimization models using Pyomo. Although a detailed explanation of Pyomo models is deferred to Chapter 3, the following examples illustrate the use of Pyomo for model (H).

2.4.1 An Abstract Formulation

An *abstract* mathematical formulation relies on unspecified parameter values. Since model (H) is an abstract model, a natural way of expressing this model in Pyomo is with Pyomo's `AbstractModel` class. A Pyomo `AbstractModel` defers initialization of model components until a *model instance* is created using set and parameter data. Thus, this modeling approach closely reflects the character of an abstract model.

Consider the following abstract Pyomo model for model (H):

```
# AbstractH.py - Implement model (H)
from pyomo.environ import *

model = AbstractModel(name="(H)")

model.A = Set()

model.h = Param(model.A)
model.d = Param(model.A)
model.c = Param(model.A)
model.b = Param()
model.u = Param(model.A)

def xbounds_rule(model, i):
   return (0, model.u[i])
model.x = Var(model.A, bounds=xbounds_rule)

def obj_rule(model):
   return sum(model.h[i] * \
```

```
            (model.x[i] - (model.x[i]/model.d[i])**2) \
            for i in model.A)
model.z = Objective(rule=obj_rule, sense=maximize)

def budget_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.A) <= \
       model.b
model.budgetconstr = Constraint(rule=budget_rule)
```

Given particular data for the parameters in this model, then one might be interested in finding an optimal assignment of values to `model.x`. There are numerous ways to provide data to Pyomo for an abstract model. Here is data (saved in `AbstractH.dat`) that defines a suitable happiness objective for one of the authors of this book:

```
# Pyomo data file for AbstractH.py
set A := I_C_Scoops Peanuts ;
param h := I_C_Scoops 1 Peanuts 0.1 ;
param d :=
  I_C_Scoops 5
  Peanuts 27 ;
param c := I_C_Scoops 3.14 Peanuts 0.2718 ;
param b := 12 ;
param u := I_C_Scoops 100 Peanuts 40.6 ;
```

This is a Pyomo data file, which includes `set` and `param` commands that closely resemble AMPL data commands.

NOTE: The backslash character at the end of a line tells Python that the line continues; we use it to help make the lines fit on a book page. In this particular case it is not strictly required because the line is breaking inside a parenthetical grouping.

2.4.2 A Concrete Formulation

A *concrete* Pyomo model initializes components as they are constructed. This allows modelers to easily make use of native Python data structures when defining a model instance. There are many ways to implement our model as a concrete model, and we consider one that uses Python lists and dictionaries.

```
# ConcreteH.py - Implement a particular instance of (H)
from pyomo.environ import *

model = ConcreteModel(name = "(H)")

A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}

def x_bounds(m, i):
    return (0,u[i])
model.x = Var(A, bounds=x_bounds)

def obj_rule(model):
    return sum(h[i] * (model.x[i] - (model.x[i]/d[i])**2)
               for i in A)
model.z = Objective(rule=obj_rule, sense=maximize)

model.budgetconstr = \
    Constraint(expr = sum(c[i]*model.x[i] for i in A) <= b)
```

Note that in the `budget_constr`, we define the constraint directly with the `expr` keyword argument, however, a construction rule could also be used. Also note that the lines,

```
A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}
```

could have been placed in a separate Python file and loaded with an `import` command. For example, if the data was in `mydata.py`, we could have placed the following line near the top of the file

```
from mydata import *
```

More details concerning `AbstractModel` and `ConcreteModel` are given in the next two chapters.

2.4.3 Linear Version

If we want to modify the abstract model given on page 22 to use expression (2.1), we would change the objective function expression rule as follows:

```
# AbstractHLinear.py - A simple linear version of (H)
from pyomo.environ import *

model = AbstractModel(name="Simple Linear (H)")

model.A = Set()

model.h = Param(model.A)
model.d = Param(model.A)
model.c = Param(model.A)
model.b = Param()
model.u = Param(model.A)

def xbounds_rule(model, i):
    return (0, model.u[i])
model.x = Var(model.A, bounds=xbounds_rule)

def obj_rule(model):
    return sum(model.h[i] * \
               (1 - model.u[i]/model.d[i]**2) * model.x[i] \
               for i in model.A)
model.z = Objective(rule=obj_rule, sense=maximize)

def budget_rule(model):
    return summation(model.c, model.x) <= model.b
model.budgetconstr = Constraint(rule=budget_rule)
```

Similarly, the modified concrete Pyomo model uses the same expression, as shown here:

```
# ConcreteHLinear.py - Linear (H)
from pyomo.environ import *

model = ConcreteModel(name="Linear (H)")

A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}

def x_bounds(m, i):
    return (0,u[i])
model.x = Var(A, bounds=x_bounds)

def obj_rule(model):
    return sum(h[i]*(1 - u[i]/d[i]**2) * model.x[i] \
               for i in A)
model.z = Objective(rule=obj_rule, sense=maximize)

model.budgetconstr = \
    Constraint(expr = sum(c[i]*model.x[i] for i in A) <= b)
```

2.5 Solving the Pyomo Model

Pyomo provides automated methods to (1) combine the model and data, (2) send the resulting *model instance* to a solver, and (3) recover the results for display and further use. Pyomo does not, itself, solve optimization problem instances. They are always passed to a solver of some sort.

2.5.1 Solvers

Pyomo can be installed without any solvers. For example, Pyomo can simply write out problem instances into files that are suitable as direct input to a solver. This use of Pyomo might be necessary if the solver is run separately on a different computer. Typically, however, a solver should be installed and accessible to Pyomo, and most of the examples in this book make this assumption.

Recall that the objective in (H) is not a linear function of the variable, x, and that the budget constraint is linear. Although many solvers can solve an instance with a quadratic objective and linear constraints, some solvers cannot. If the only solver on your computer is limited to linear problems, then you would need to approximate (H) with a linear model.

2.5.2 The `pyomo` Command

The `pyomo` command provides a command-line interface for solving Pyomo models. To use the `pyomo` command, the user must be in a terminal window, but **not** in a Python interpreter. To verify that pyomo is properly installed, run the command

```
pyomo --version
```

The version number of Pyomo should be displayed without error messages. The command

```
pyomo --help
```

displays some high level help on the terminal. Note that there are two dashes in `--version` and `--help`.

We assume that you have the solver `glpk` properly installed, which can be verified by running the command

```
glpsol --help
```

in your terminal. If the model file `ConcreteHLinear.py` is in the current directory, then the following command will run glpk to find a solution to the problem:

```
pyomo solve --solver=glpk ConcreteHLinear.py
```

If you have some other solver that you want to use, substitute its name for `glpk`. Note that there are two dashes in `--solver`. The `solve` subcommand displays information about the time required for each step in the optimization process as well as a little bit of information about the final solution. Information about the values of the decisions variables is put in the file `results.json` (or in `results.yml` if the Python `pyyaml` package is installed).

A data file is specified to use the `solve` subcommand with an abstract model:

```
pyomo solve --solver=glpk AbstractHLinear.py AbstractH.dat
```

Many other solvers can be used with Pyomo. For example, if the Gurobi solver is installed, then the following command can be used to solve the original, quadratic implementation of (H):

```
pyomo solve --solver=gurobi AbstractH.py AbstractH.dat
```

Also note that the solution can be printed to the screen with the `--summary` option

```
pyomo solve --solver=glpk --summary ConcreteHLinear.py
```

2.5.3 Python Scripts

An alternative to using the `pyomo` command is to add additional Python commands in the model file to explicitly optimize the model. Such a Python script is then executed using Python from the command line or within a development environment like Spyder. A script defining a concrete model can easily be solved by adding the following lines to the bottom:

```
opt = SolverFactory('glpk')

results = opt.solve(instance) # solves and updates instance

instance.display()
```

If the resulting file is called `ConcHLinScript.py`, then it can be run from the terminal with the command line:

```
python ConcHLinScript.py
```

Similarly, the following lines can be used to optimize an abstract model:

```
opt = SolverFactory('glpk')

instance = model.create_instance("AbstractH.dat")
results = opt.solve(instance) # solves and updates instance

instance.display()
```

For abstract models that rely on external data, the call to the `create_instance` method must specify the data source, as in

```
instance = model.create_instance("AbstractH.dat")
```

Chapter 3
Pyomo Overview

Abstract This chapter provides an overview of the modeling strategies and capabilities of Pyomo. We provide a brief overview of the core modeling components supported by Pyomo. We then discuss the differences between concrete and abstract models, and give some guidance on when to select one approach over another. We provide some examples that illustrate the use of the `pyomo` command and general scripting capabilities. Finally, we close the chapter with a discussion of some of the modeling capabilities within Pyomo (e.g., discrete variables and nonlinear models).

3.1 Introduction

Pyomo supports an object-oriented design for the definition of optimization models. A Pyomo *model* object contains a collection of modeling *components* that define the optimization problem. The Pyomo package includes modeling components that are necessary to formulate an optimization problem: variables, objectives, and constraints, as well as other modeling components that are commonly supported by modern AMLs, including index sets and parameters. These modeling components are defined in Pyomo through the following Python classes:

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

In this chapter, we give an overview of these components and how to define and solve Pyomo models. The basic steps of a simple modeling process are as follows:

1. create an instance of a model using Pyomo modeling components
2. pass this instance to a solver to find a solution
3. report and analyze results from the solver

Pyomo provides a command line utility (the `pyomo` command) that executes this process. Additionally, Pyomo supports general scripting with Python where a user can flexibly control the solution process and develop a custom workflow, such as solving sequences of problems with modifications, or more complex meta-algorithms.

In this chapter, we use an example problem to illustrate the process of formulating a real-world model, including the use of modeling components, indexed components, and construction rules. Additionally, we describe Pyomo's support for two general model strategies: abstract and concrete models. We use the `pyomo` command to analyze these models, and discuss the use of scripting for more advanced workflows.

3.2 The Warehouse Location Problem

We use the warehouse location problem throughout this chapter, which considers the optimal locations to build warehouses to meet delivery demands. Let N be a set of candidate warehouse locations, and let M be a set of customer locations. For each warehouse n, the cost of delivering product to customer m is given by $d_{n,m}$. We wish to determine the optimal warehouse locations that will minimize the total cost of product delivery. The binary variables y_n are used to define whether or not a warehouse should be built, where y_n is 1 if warehouse n is selected and 0 otherwise. The variable $x_{n,m}$ indicates the fraction of demand for customer m that is served by warehouse n.

The variables x and y are determined by the optimizer, and all other quantities are known inputs or parameters in the problem. This problem is a particular description of the *p-median* problem, and it has the interesting property that the x variables will converge to $\{0,1\}$ even though they are not specified as binary variables.

The complete problem formulation is:

$$\min_{x,y} \sum_{n\in N}\sum_{m\in M} d_{n,m}x_{n,m} \tag{WL.1}$$

$$\text{s.t.} \sum_{n\in N} x_{n,m} = 1, \ \forall\, m \in M \tag{WL.2}$$

$$x_{n,m} \le y_n, \ \forall\, n \in N,\ m \in M \tag{WL.3}$$

$$\sum_{n\in N} y_n \le P \tag{WL.4}$$

$$0 \le x \le 1 \tag{WL.5}$$

$$y \in \{0,1\} \tag{WL.6}$$

Here, the objective (equation WL.1) is to minimize the total cost associated with delivering products to all the customers. Equation WL.2 ensures that each customer's

demand is fully met, and equation WL.3 ensures that a warehouse can deliver product to customers only if that warehouse is selected to be built. Finally, with equation WL.4 the number of warehouses that can be built is limited to P.

For our example, we will assume that $P = 2$, with the following data for warehouse and customer locations,

Customer locations	$M = \{$'NYC', 'LA', 'Chicago', 'Houston'$\}$
Candidate warehouse locations	$N = \{$'Harlingen', 'Memphis', 'Ashland'$\}$

with the costs $d_{n,m}$ as given in the following table:

	NYC	LA	Chicago	Houston
Harlingen	1956	1606	1410	330
Memphis	1096	1792	531	567
Ashland	485	2322	324	1236

3.3 Pyomo Models

Pyomo supports an object-oriented design where modeling components are added to a Pyomo model to define the optimization problem. In this section, we give an overview of the common modeling components, discuss abstract and concrete models, and provide complete Pyomo examples of the warehouse location problem.

3.3.1 Components for Variables, Objectives and Constraints

Optimization problems require, at least, one variable and an objective function. Most problems also include constraints. The Pyomo classes for implementing these modeling components are `Var`, `Objective`, and `Constraint`. The following example shows how these components could be defined:

```
model.x = Var()
model.y = Var(bounds=(-2,4))
model.z = Var(initialize=1.0, within=NonNegativeReals)

model.obj = Objective(expr=model.x**2 + model.y + model.z)

model.eq_con = Constraint(expr=model.x + model.y + model.z \
    == 1)
model.ineq_con = Constraint(expr=model.x + model.y <= 0)
```

In this example, we have created three optimization variables (x, y, and z), a single objective, and two constraints. For each optimization variable, we create an instance of the `Var` class and add that instance as an attribute to the model object. The code `model.x=Var()` creates an instance of the Pyomo class `Var` and makes it accessible by `model.x`. Furthermore, the `model` object identifies when a component is

being added and performs special processing that includes, for example, setting the name of the instance of `Var` to "x", and setting a reference to the owner model.

This example declares x as a continuous variable, but keyword arguments can be used to define properties of the variable. For example, `bounds` is used to set lower and upper bounds, `initialize` is used to set initial values, and `within` is used to set the domain. In this example, `model.y` has a lower bound of -2 and an upper bound of 4, and `model.z` has a lower bound of 0, and no upper bound (since the keyword argument `within` is set to non-negative reals).

NOTE: The use of keyword arguments is common in the constructors for Pyomo components to specify component properties. See Chapter 4 for more details about supported keyword arguments for Pyomo components.

This example also defines an objective function using the `Objective` component. Here, the `expr` keyword is used to define the expression for the objective function. By default, optimization objectives are minimized, but the `sense` keyword can be set to `maximize` for maximization problems. This example also declares an equality constraint and inequality constraint using `Constraint` components. The `expr` keyword is used again to define the mathematical expressions for the constraints, including the logical operator separating the left hand side expression and the right hand side expression. Constraints can include a logical operator that indicates equal to (==), less than or equal to (<=), or greater than or equal to (>=). See Chapter 4 for a detailed description of these components and the available keyword arguments.

NOTE: In the previous example, the objective and constraints were defined with the `expr` keyword. While this is convenient for illustrating the examples with few lines of code, these components are often defined using *construction rules*, which are discussed in more detail in Sections 3.3.3 and 4.2.1.

3.3.2 Indexed Components

In the previous example, each of the modeling components were *scalar*. That is, each of the optimization variables x, y, and z were single values only (not vectors or arrays). The constraints were also scalar (each declaration created only a single mathematical constraint). When modeling large, complex applications, it is common to have vectors of variables and constraints whose dimension and indexing is determined according to model data. This is handled within Pyomo through *indexed* components .

To illustrate the concept of an indexed component, consider the warehouse location problem (WL). We could formulate this problem using only scalar components,

and, for example, create separate x variables for each pair of warehouses and customers,

```
model.x_Harlingen_NYC = Var(bounds=(0,1))
model.x_Harlingen_LA = Var(bounds=(0,1))
model.x_Harlingen_Chicago = Var(bounds=(0,1))
model.x_Harlingen_Houston = Var(bounds=(0,1))
model.x_Memphis_NYC = Var(bounds=(0,1))
model.x_Memphis_LA = Var(bounds=(0,1))
#...
```

We could manually expand the constraint described in WL.4 as,

```
model.maxY = Constraint(expr=model.y_Harlingen + \
    model.y_Memphis + model.y_Ashland <= P)
```

and write all the constraints in equation (WL.2) explicitly as,

```
model.one_warehouse_for_NYC = \
    Constraint(expr=model.x_Harlingen_NYC + \
    model.x_Memphis_NYC + model.x_Ashland_NYC == 1)

model.one_warehouse_for_LA = \
    Constraint(expr=model.x_Harlingen_LA + \
    model.x_Memphis_LA + model.x_Ashland_LA == 1)
#...
```

Using indexed components, we can provide Pyomo with the list of valid indices for the x and y variables when we declare them. Using the following Python data:

```
N = ['Harlingen', 'Memphis', 'Ashland']
M = ['NYC', 'LA', 'Chicago', 'Houston']
```

we can declare variables as follows:

```
model.x = Var(N, M, bounds=(0,1))
model.y = Var(N, within=Binary)
```

We refer to `N` and `M` as index sets for the indexed variables `model.x` and `model.y`. Specifically, the variable y is indexed over N, and the variable x is a two-dimensional array that is indexed over both N and M. With this declaration, an element of x can be accessed by `model.x[i,j]` where `i` and `j` are elements of the sets N and M, respectively.

NOTE: Pyomo modeling components can include any number of index sets as unnamed arguments in their declaration. These index sets specify the valid indices for individual elements of the component. (Note that they must be passed before any other named keyword arguments.)

Given these declarations, constraint (WL.4) can be defined as

```
model.num_warehouses = Constraint(expr=sum(model.y[n] for \
    n in N) <= P)
```

This declaration uses Python's iteration syntax to sum over a set of indexed variables. The *list comprehension* syntax enables a concise specification of the summation, where the syntax specifies that the terms `model.y[n]` are generated by iterating over the set `N`. As these terms are generated, the function `sum` adds them together to form the overall expression. Similarly, the objective can be defined as

```
model.obj = Objective(expr=sum(d[n,m]*model.x[n,m] for n \
    in N for m in M))
```

where the terms `d[n,m]*model.x[n,m]` are generated by iterating over both N and M.

3.3.3 Construction Rules

Nearly all of Pyomo's modeling components can be indexed, and the construction of many indexed constraints is performed with *construction rules*. Consider constraint (WL.2):

$$\sum_{n \in N} x_{n,m} = 1, \ \forall\, m \in M$$

This mathematical notation indicates that there is a single constraint for each m in the set M. The `Constraint` component can be declared as an indexed constraint over the elements in this set. However, we need a mechanism to provide Pyomo with the explicit expressions for each element in M. Pyomo allows model components to be initialized with user-defined functions, which we call *rules*.

The following example illustrates the use of a construction rule to define constraint (WL.2):

```
def one_per_cust_rule(model, m):
   return sum(model.x[n,m] for n in N) == 1
model.one_per_cust = Constraint(M, rule=one_per_cust_rule)
```

The last line in this example declares the constraint by creating a `Constraint` component that is indexed over the set M. The `rule` keyword argument indicates that the function `one_per_cust_rule` is used to construct each constraint.

The first argument in the function `one_per_cust_rule` is the model instance being constructed. It is followed by the particular values for the indices of the constraint being constructed. When Pyomo constructs the `Constraint` object, the construction rule is called for each of the values of the specified index sets.

NOTE: Pyomo expects a construction rule to return an expression for every index value. If no constraint is needed for a particular combination of indices, then the value `Constraint.Skip` can be returned instead.

Construction rules can be used for most modeling components, using the `rule`

keyword argument, even if the component is not indexed. Although the function arguments for component rules are the same for all component types, the following table illustrates that the expected type of the return value is different:

Component	Construction Rule Return Types
`Set`	A Python set or list object
`Param`	An integer or float value
`Objective`	An expression
`Constraint`	A constraint expression.

3.3.4 Abstract and Concrete Models

Pyomo supports two strategies for model declaration: concrete models, which immediately construct model components, and abstract models, which defer component construction. Abstract models reflect the structure of many mathematical optimization formulations. For example, the formulation of the warehouse location problem (WL) on page 30 is written in a general manner that describes a class of optimization problems. However, we cannot *solve* this problem yet since the actual data for the problem (N, M, d, and P) has not been specified. A solver must be given a specific instance of a problem (with the data specified).

In Pyomo, an abstract model is declared first, and component construction is delayed until the data is loaded and Pyomo creates the model instance. This modeling approach is illustrated in the top pane of Figure 3.1. An `AbstractModel` object is created, and then the data for a particular problem is given to Pyomo, and then Pyomo performs the construction process in order to create an *instance* of the model with all the variables, constraint expressions, and objective expressions that can be sent to a solver. This requires a two-pass approach where the model is declared in the first pass, and subsequently the model is constructed using data values that are specified separately. To support delayed construction, the model must be defined using construction rules.

A *concrete* model can be used when data is available before model components are declared. Concrete models support a more programmatic style where the model instance is created immediately; model components are constructed and initialized on the first pass as Python executes the model script. This modeling approach is illustrated in the bottom pane of Figure 3.1. A `ConcreteModel` object is created, and the data needs to be present *before* each component is declared. As Python executes your model script, the particular model instance and its components are created immediately as Python encounters the component declaration. Once the execution through the Python file is completed the model is ready to be sent to the solver (i.e., a single pass). Note that, at this point, the `ConcreteModel` *is* the specific instance.

NOTE: Construction rules can be still be used with concrete models (the rules are immediately fired as they are encountered in the model's Python file).

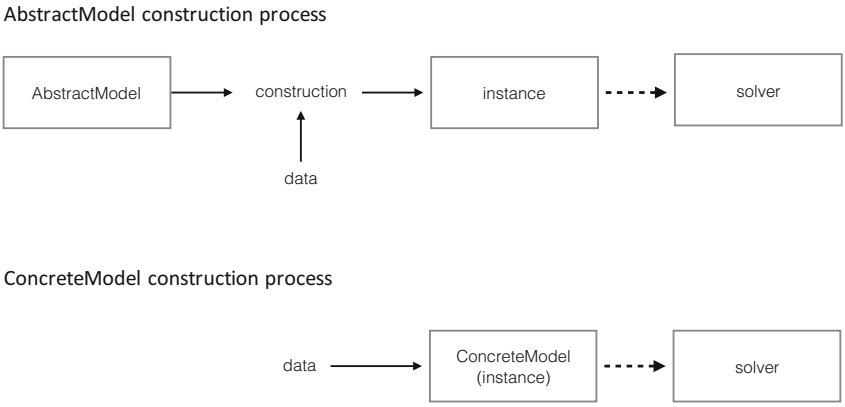


Fig. 3.1: This figure describes the construction process for both abstract and concrete models. The top pane describes the declarative style used for abstract models. The `AbstractModel` is first created. Then, given a particular realization of the data, Pyomo performs the construction process in order to create an *instance* of the model that can then be sent to the solver. The bottom pane describes the programmatic style used for concrete models. As Python executes the model script, the component objects are constructed immediately using data that is previously declared. The particular model *instance* is ready to be sent to the solver once the first pass is complete.

The choice of which model object to use (`AbstractModel` or `ConcreteModel`) is largely a matter of taste and preference. The biggest difference between the use of `AbstractModel` or `ConcreteModel` relates to the specification of data. When using an `AbstractModel`, the names and structure of the data that will be used in a model is declared prior to construction (but not the values themselves). This means that Pyomo is aware of the existence of the sets and parameters that *will* be encountered when constructing the instance. More importantly, Pyomo knows the associated names and types of these quantities and something about the relationships among them (e.g., the variable y is indexed by set N). This allows the user to specify the data using any number of Pyomo supported data formats while referring to these quantities by name. Pyomo includes many options for supplying data to an abstract model, including 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 [2].

Concrete models on the other hand, require that the data be available before individual components are declared. This programmatic approach allows for straight-

forward use of native Python data types when creating a model instance. Therefore, if you are more comfortable building models in a procedural programming environment (like Python or MATLAB), or if your application requires a more extensive workflow than that supported by the `pyomo` command, then a `ConcreteModel` is more appropriate. This is especially true if your data can be easily loaded into Python through other Python packages (e.g., pandas).

NOTE: In general, an `AbstractModel` is more straightforward for users that are unfamiliar with Python or prefer to work in a more traditional AML environment. A `ConcreteModel` often requires more Python coding on the part of the user to load the data (e.g., using an existing Python package for the raw data format) and apply it to the model, but offers transparent control over execution order as well as pre- and post-analysis.

3.3.5 A Concrete Model for the Warehouse Location Problem

The warehouse location problem can be represented as a concrete model as follows:

```
1  # wl_concrete.py: ConcreteModel version of warehouse \
       location determination problem
2  from pyomo.environ import *
3
4  model = ConcreteModel(name="(WL)")
5
6  N = ['Harlingen', 'Memphis', 'Ashland']
7  M = ['NYC', 'LA', 'Chicago', 'Houston']
8  d = {('Harlingen', 'NYC'): 1956, \
9      ('Harlingen', 'LA'): 1606, \
10     ('Harlingen', 'Chicago'): 1410, \
11     ('Harlingen', 'Houston'): 330, \
12     ('Memphis', 'NYC'): 1096, \
13     ('Memphis', 'LA'): 1792, \
14     ('Memphis', 'Chicago'): 531, \
15     ('Memphis', 'Houston'): 567, \
16     ('Ashland', 'NYC'): 485, \
17     ('Ashland', 'LA'): 2322, \
18     ('Ashland', 'Chicago'): 324, \
19     ('Ashland', 'Houston'): 1236 }
20 P = 2
21
22 model.x = Var(N, M, bounds=(0,1))
23 model.y = Var(N, within=Binary)
24
25 def obj_rule(model):
```

```
26      return sum(d[n,m]*model.x[n,m] for n in N for m in \
            M)
27  model.obj = Objective(rule=obj_rule)
28
29  def one_per_cust_rule(model, m):
30      return sum(model.x[n,m] for n in N) == 1
31  model.one_per_cust = Constraint(M, \
        rule=one_per_cust_rule)
32
33  def warehouse_active_rule(model, n, m):
34      return model.x[n,m] <= model.y[n]
35  model.warehouse_active = Constraint(N, M, \
        rule=warehouse_active_rule)
36
37  def num_warehouses_rule(model):
38      return sum(model.y[n] for n in N) <= P
39  model.num_warehouses = \
        Constraint(rule=num_warehouses_rule)
```

This script begins by importing the Pyomo environment, which defines the Python classes used to build a model. Lines 4 creates a `ConcreteModel` and provides a name.

Lines 6-20 define the data for our problem. The Python lists `N` and `M` are used to specify the valid warehouse locations and the customer locations respectively. The Python dictionary `d` defines the costs associated with serving each customer from each location, and line 20 specifies P, which is the number of warehouses we want to place. These native Python data structures are used to declare and construct the Pyomo modeling components (the `Var`, `Objective`, and `Constraint`) objects.

Lines 22 and 23 declare and construct the variables for the problem. The `model` object is a `ConcreteModel`, and once these lines are executed, the variables x and y are completely constructed with known indices. Lines 25 and 26 define the *construction rule* for the objective function, and line 27 declares the objective function (in `model.obj`). As soon as line 27 executes, the rule that is declared on lines 25 and 26 is called to construct the expression for the objective function. Similarly, in the remaining lines of the Python file, the constraint rules are declared, followed by the constraint objects themselves. Since this is a `ConcreteModel`, the constraint rules are called immediately (as soon as Python executes lines 31, 35, and 39).

This model can be solved using the `pyomo` command,

```
pyomo solve --solver=glpk wl_concrete.py
```

By default, the results are stored in a file indicated by the output from `pyomo`. The results can also be printed to the screen using the `--summary` option.

In this example, the python data for the problem (`N`, `M`, `d`, and `P`) were explicitly defined in the model file. While this is convenient to create a self-contained example, in practice is it often more convenient to load data from another source (e.g., another python file).

The earlier `ConcreteModel` examples in this chapter included the data for the model explicitly in the main Python file with the model. This is typically not viewed as good practice, since we frequently want to solve the same model (or model class) with different data. Instead, we could *import* the data into our model file or execution script. Then, we would only need to change one line in our file to run with different data. We could also load our data from an external file.

Consider, for example, Figure 3.2, which shows some example data specified in Microsoft Excel. The following script loads this data from Excel. This example makes use of the Python sys package to allow a user to specify the name of the Excel file to load and the value of the parameter P on the command line.

```
# wl_excel.py: Loading Excel data using Pandas
import pandas
import sys
from pyomo.environ import *

# read the data from excel using pandas
df = pandas.read_excel(sys.argv[1], 'Delivery Costs', \
    header=0, index_col=0)

N = list(df.index.map(str))
M = list(df.columns.map(str))
d = {(r, c):df.at[r,c] for r in N for c in M}
P = int(sys.argv[2])

# create the model (could be imported)
model = ConcreteModel(name="(WL)")

model.x = Var(N, M, bounds=(0,1))
model.y = Var(N, within=Binary)

def obj_rule(model):
   return sum(d[n,m]*model.x[n,m] for n in N for m in M)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(model, m):
   return sum(model.x[n,m] for n in N) == 1
model.one_per_cust = Constraint(M, rule=one_per_cust_rule)

def warehouse_active_rule(model, n, m):
   return model.x[n,m] <= model.y[n]
model.warehouse_active = Constraint(N, M, \
    rule=warehouse_active_rule)

def num_warehouses_rule(model):
   return sum(model.y[n] for n in N) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

# solve the model and report the results
solver = SolverFactory('glpk')
solver.solve(model)
model.pprint()
```

	A	B	C	D	E	F
1		NYC	LA	Chicago	Houston	
2	Harlingen	1956	1606	1410	330	
3	Memphis	1096	1792	531	567	
4	Ashland	485	2322	324	1236	
5						
6						

Fig. 3.2: This figure shows the data for our warehouse location problem as formatted in Microsoft Excel.

3.3.6 Modeling Components for Sets and Parameters

Abstract models require declarations for sets and parameters that are used in the model. The Pyomo `Set` and `Param` components provide this functionality and several other features, including data validation.

A Pyomo `Set` component is used to declare valid indices for any component that is indexed. For example, in the context of our warehouse location problem, we have two sets: N stores the valid warehouse locations and M stores the customer locations. We can easily declare these sets using the following code:

```
model.N = Set()
model.M = Set()
```

These `Set` objects can be used to define indexed variables or constraints:

```
model.x = Var(model.N, model.M, bounds=(0,1))
model.y = Var(model.N, within=Binary)
```

This example passes `Set` objects into the `Var` constructor, rather than the Python lists used for the concrete model in the previous section. A `Set` component can be initialized with the `initialize` keyword argument, using a Python set, list, or tuple. The `Set` component can also be initialized with external data sources.

Pyomo `Set` objects can also be indexed by other sets. Consider the following example:

```
model.PremierSundaes = Set()
model.Toppings = Set(model.PremierSundaes)
```

The set `model.Toppings` is an indexed set. If `model.PremierSundaes` is given the values {'PBC-Banana', 'Very Berry' }, then we can define toppings for each of these indices. For example, `model.Toppings['PBC-Banana']` may contain the set {'Peanut Butter', 'Chocolate Fudge', 'Banana' }, whereas `model.Toppings['Very Berry']` may contain {'Strawberries', 'Raspberries', 'Blueberries', 'Crunch-berries'}.

A Pyomo `Param` component is used to define data values for our problem. In the context of the warehouse location problem, two pieces of data need to be specified:

P and $d_{n,m}$. These parameters can be declared using the following code:

```
model.d = Param(model.N,model.M)
model.P = Param()
```

This example declares a scalar parameter P and an indexed parameter d. The parameter d is indexed by the Pyomo sets for the warehouse and customer locations that we defined earlier. As with the `Set` object, values for these parameters could be provided through the `initialize` keyword argument using a Python dictionary, by defining a construction rule, or via an external data source.

By default, parameters are immutable, meaning that once their values are set, the values cannot be changed. This default behavior allows for increased efficiency within Pyomo when handling expressions. However, you can define a parameter whose values are mutable with the `mutable=true` keyword argument. This can be useful if you have a model that you want to solve multiple times with different values of some of the parameters.

3.3.7 An Abstract Model for the Warehouse Location Problem

The warehouse location problem can be represented as an abstract model as follows:

```
1  # wl_abstract.py: AbstractModel version of warehouse \
       location determination problem
2  from pyomo.environ import *
3
4  model = AbstractModel(name="(WL)")
5  model.N = Set()
6  model.M = Set()
7  model.d = Param(model.N,model.M)
8  model.P = Param()
9  model.x = Var(model.N, model.M, bounds=(0,1))
10 model.y = Var(model.N, within=Binary)
11
12 def obj_rule(model):
13    return sum(model.d[n,m]*model.x[n,m] for n in \
          model.N for m in model.M)
14 model.obj = Objective(rule=obj_rule)
15
16 def one_per_cust_rule(model, m):
17    return sum(model.x[n,m] for n in model.N) == 1
18 model.one_per_cust = Constraint(model.M, \
       rule=one_per_cust_rule)
19
20 def warehouse_active_rule(model, n, m):
21    return model.x[n,m] <= model.y[n]
22 model.warehouse_active = Constraint(model.N, model.M, \
       rule=warehouse_active_rule)
23
```

```
24  def num_warehouses_rule(model):
25      return sum(model.y[n] for n in model.N) <= model.P
26  model.num_warehouses = \
        Constraint(rule=num_warehouses_rule)
```

Line 4 creates an `AbstractModel` object. Lines 5 through 8 declare the model sets and parameters, and no data needs to be associated with these components yet. Likewise, lines 9 and 10 declare the model variables x and y. Pyomo knows that these components will be indexed by the sets `model.N` and `model.M`, but does not yet know the specific indices.

The objective construction rule is defined on lines 12 and 13, and the objective `model.obj` is declared on line 14. Since this is an abstract model, the objective rule `obj_rule` is not yet called. At this point, Pyomo knows what rule to call to construct the objective component, but it has not called the constructor because this is an abstract model. Constraint rules and constraint components are declared in a similar manner.

This script can be executed with the `python` command, but that would not actually *do* anything. This script declares the model, but it does not define the model data or create the problem instance for the solver. The action of applying a data file to this abstract model can be scripted explicitly in Python code, or it can be done using the `pyomo` command.

Data can be expressed in several different formats. For example, the following Pyomo data file can be used:

```
# wl_data.dat: Pyomo format data file for the warehouse \
    location problem

set N := Harlingen Memphis Ashland ;
set M := NYC LA Chicago Houston;

param d :=
    Harlingen NYC 1956
    Harlingen LA  1606
    Harlingen Chicago  1410
    Harlingen Houston  330
    Memphis NYC  1096
    Memphis LA  1792
    Memphis Chicago  531
    Memphis Houston  567
    Ashland NYC  485
    Ashland LA  2322
    Ashland Chicago  324
    Ashland Houston  1236
;

param P := 2 ;
```

This problem can be solved using the following `pyomo` command:

```
pyomo solve --solver=glpk wl_abstract.py wl_data.dat
```

The `--summary` flag can be used to provide more detailed output about the solution.

When `pyomo` runs, it executes `wl_abstract.py` to create an `AbstractModel` with the name `model`. This model object contains the Pyomo modeling components that have been declared. Then `pyomo` reads the data file `wl_data.dat` and applies this data to the `Set` and `Param` components in the same order that the components were declared in the model. Next, the `pyomo` command constructs all of the remaining components in declaration order: the variables, the objective, and the constraints. After the model is constructed, `pyomo` calls the solver to find the solution.

3.4 Solving the Pyomo Model

As shown previously, the `pyomo` command can be used to solve a problem by specifying the python model file and the data file on the command line. The `pyomo` command is a pre-defined script that builds the model instance using the specified data, sends that instance to a solver to be solved, and reports the solution if successful. The Pyomo package also allows users to control this flow by writing their own script. Writing a script to drive the process is common with concrete models. We will provide an introduction to both of these approaches.

3.4.1 Using the `pyomo` Command

As noted previously, the Pyomo software distribution includes a pre-defined execution script, the `pyomo` command, that can be used to solve Pyomo models. The `pyomo solve` command automatically executes the following steps:

1. Create the model (abstract or concrete)
2. Read the data and generate a model instance (if applicable)
3. Apply a solver to the model instance
4. Load the results into the model instance
5. Display the solver results

For example, the following command solves the warehouse location problem defined in `wl_concrete.py` using the LP solver `glpk`:

```
pyomo solve --solver=glpk wl_concrete.py
```

Similarly, the following command uses `glpk` to solve the model declared in `wl_abstract.py` with data from `wl_data.dat`:

```
pyomo solve --solver=glpk wl_abstract.py wl_data.dat
```

When the pyomo command loads a user-defined Pyomo model, it looks for a ConcreteModel or AbstractModel object named model. 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).

Documentation of the command-line options for pyomo is generated by specifying the --help option. Help on a subcommand like solve can be obtained by using --help after the subcommand (e.g., pyomo solve --help. For some options, a valid solver must be specified through the --solver option before the help can be seen. Options can control how much information is printed to the screen, including --summary to print a summary of the solution after the problem is solved, and --stream-solver to print the solver output to the screen while solving. See Chapter 5 for further details about the pyomo command.

3.4.2 Scripting the Solution Process

Pyomo users can leverage Python's powerful scripting capabilities to execute custom workflows that manipulate and optimize models. Consider the following script, which imports a model from wl_concrete.py, creates a solver interface, performs optimization, and displays the results:

```
from pyomo.environ import * # import pyomo environment
from wl_concrete import model # import model

solver = SolverFactory('glpk') # create the glpk solver
solver.solve(model) # solve

model.y.pprint() # print the optimal warehouse locations
```

This example uses the Python import statement to import the model from our code in wl_concrete.py. Pyomo's implementation of pprint is used to show the results, but we can also implement a problem-specific output. For example:

```
from pyomo.environ import * # import pyomo environment
from wl_concrete import model, N, M # import model and sets

solver = SolverFactory('glpk') # create the glpk solver
solver.solve(model) # solve

# produce nicely formatted output
for wl in N:
   if value(model.y[wl]) > 0.5:
      customers = [str(cl) for cl in M if \
          value(model.x[wl, cl] > 0.5)]
      print(str(wl)+' serves customers: '+str(customers))
   else:
      print(str(wl)+": do not build")
```

which produces the output

```
Harlingen serves customers: ['LA', 'Houston']
Memphis: do not build
Ashland serves customers: ['NYC', 'Chicago']
```

NOTE: Once a Pyomo model has been constructed, the model can be printed using the `pprint` method, `model.pprint()`. This summarizes the information in the Pyomo model, including the constraint and objective expressions. This can be a very useful debugging tool when a model is not generating the expected results, since it shows the fully expanded version of the model.

Abstract models can also be used in scripts, but a concrete instance must be created from the `AbstractModel` object using the `create_instance` method. The following example takes an `AbstractModel`, constructs the instance using a data file called `wl_data.dat`, solves the instance, and prints some results:

```
instance = model.create_instance('wl_data.dat')
solver = SolverFactory('glpk')
solver.solve(instance)
instance.y.pprint()
```

NOTE: This section has only scratched the surface of what is possible with the `pyomo` command and with scripting. Both of these topics are covered in more detail in Chapters 5 and 14, respectively.

Chapter 4
Pyomo Models and Components: An Introduction

Abstract This chapter describes the core classes that are used to define optimization models in Pyomo. Most of the discussion focuses on modeling components that are used to declare parts of a model. We include a discussion of the options that can be used when declaring the components and information about key component attributes and methods.

4.1 An Object-Oriented AML

Pyomo supports an object-oriented approach for representing mathematical optimization models. A model object is created, and then modeling components are added to this object to declare different parts of the model. Pyomo includes modeling components that are commonly supported by modern AMLs: variables, constraints, objectives, index sets, and symbolic parameters. In this chapter we will describe Pyomo modeling components. In subsequent chapters, additional components are introduced that provide enhanced functionality to represent advanced optimization model features.

Users can create two types of models in Pyomo: concrete and abstract. A *concrete* model is constructed "on-the-fly" as each model component is declared. Therefore, the data associated with a concrete model must be specified before model components are declared. A user can leverage native Python data structures to define components in a concrete model. The `ConcreteModel` class is used to represent a concrete model.

In contrast, an *abstract* model supports complete declaration of a model abstractly. A specific problem instance is not constructed until all components are declared and the data is provided. The `AbstractModel` class is used to create an abstract model. Because abstract models allow components to reference data before it is defined, they often rely on Pyomo data components such as `Set` and `Param` to provide an abstract definition of the data used to construct the model (although these components can be used on concrete models as well).

The following are the core modeling components in Pyomo:

`Var`	The `Var` component is used to represent optimization variables. Pyomo supports continuous and discrete variables, and includes several pre-defined domains.
`Objective`	The `Objective` component defines the function or functions that are to be optimized by the solver. This component contains the expression used to define the objective function, and a flag to indicate the sense (maximize or minimize).
`Constraint`	Constraints are used to define additional restrictions on the optimization variables. The `Constraint` component contains expressions and the appropriate relational operator. Pyomo supports equality (==) and general inequality (<= or >=) constraints.
`Set`	The `Set` component represents a collection of data that can include numeric (e.g., integer), or symbolic (e.g., string) elements. They are most commonly used to define valid indices for other components. Several common set operations are also supported.
`Param`	The `Param` component is used to represent numerical or symbolic values for data in the optimization problem. In contrast with simple Python data types (e.g., float), `Param` objects support the ability to change values (meaning they are *mutable*), and include features like sparse representations and default values.
`Expression`	The `Expression` component can be used to create a Pyomo expression that can be reused in different parts of a Pyomo model. This is useful for representing common sub-expressions for memory efficiency. Furthermore, like mutable parameters, the underlying expression can be changed between calls to the solver.
`Suffix`	Frequently, there is a need to provide meta-data about a model or a component (e.g., dual information from a constraint). This is supported through the Pyomo `Suffix` component.

In this chapter, we will describe each of these components in more detail. A variety of other modeling components included in Pyomo, some of which are briefly discussed at the end of this chapter and covered in more detail in the remaining chapters of the book.

NOTE: Unless otherwise stated, the code snippets and examples used in this chapter refer to concrete models.

4.2 Common Component Paradigms

There are behaviors that are common across most of the Pyomo modeling components listed in the previous section. Additionally, there are some common paradigms

that are adopted across many components. In this section, we will describe these common behaviors.

4.2.1 Indexed Components

As shown in the previous chapter, Pyomo components can be declared as individual, atomic entities or as indexed collections. Indexed components will appear in several of the examples that follow in this chapter. Consider the following model:

```
model = ConcreteModel()
model.A = Set(initialize=[1,2,3])
model.B = Set(initialize=['Q', 'R'])
model.x = Var()
model.y = Var(model.A, model.B)
model.o = Objective(expr=model.x)
model.c = Constraint(expr = model.x >= 0)
def d_rule(model, a):
    return a*model.x <= 0
model.d = Constraint(model.A, rule=d_rule)
```

The component `c` specifies a single constraint in this model, and the component `d` specifies a collection of constraints indexed over the set `A`. The `Constraint` component can be used to declare both simple constraints and indexed constraints. In general, components can also be indexed by multiple index sets. For example, `model.y` is indexed over both `A` and `B`, and it can be referenced by `model.y[i,j]` where `i` is any valid element from `model.A` and `j` is any valid element from `model.B` (e.g., `model.y[2,'Q']`).

NOTE: Any unnamed arguments in a component constructor are assumed to be index sets for the component. They specify the set of valid indices for the component.

Declaration of arguments for indexed components is often more complex. For example, the `initialize` keyword argument can be used when declaring a single variable,

```
model.x = Var(initialize=3.14)
```

Specifying a value for these types of keyword arguments is straightforward when the component is not indexed. When the component is indexed, however, we may want to specify a different value for each of the indices. There are three approaches typically supported for these kinds of keyword arguments.

- When a single scalar value is passed, then that value is used for all the indices of the component.
- In many cases, you can also pass a Python dictionary (index-value pairs) where the keys of the dictionary agree with the keys of the index set for the component.

- It is also possible to pass in a function (rule) that will be called to provide the value for every index in the component.

These uses are illustrated here:

```
model.A = Set(initialize=[1,2,3])
model.x = Var(model.A, initialize=3.14)
model.y = Var(model.A, initialize={1:1.5, 2:4.5, 3:5.5})
def z_init_rule(m, i):
   return float(i) + 0.5
model.z = Var(model.A, initialize=z_init_rule)
```

4.3 Variables

Pyomo variables are created using the `Var` class, which can represent a single value or an array of values. Variables can have initial values, and the value of a variable can be retrieved and set by the user or by a solver as part of the solution process.

4.3.1 `Var` Declarations

The following code provides a simple declaration of a non-indexed `Var` object.

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

Named and un-named arguments are supported, and Table 4.1 provides a list of the common arguments that can be passed when declaring the `Var` component

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`within` or `domain`	specifies the valid domain or values for a variable	a Pyomo `Set` object, Python list, or rule function
`bounds`	provide lower and upper bounds for the variable	a 2-tuple, or a rule function
`initialize`	provides initial values for the variables	a scalar value, Python dictionary of index-value pairs, or rule function

Table 4.1: Common Declaration Arguments for `Var` Component

The domain of a variable (i.e., the set of legal values) is specified with either the `domain` or `within` keyword options to the `Var` constructor:

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

In this example, `model.y` is only allowed to take on the integer values 1, 2, or 3. The variable `model.r` can have any real value, and `model.w` is restricted to be binary (that is 0/1 or True/False). If the domain is not specified, the default is the `Reals` virtual set. Other virtual sets supported by Pyomo are defined in Table 4.2. We note that these virtual sets can also be used in other contexts (e.g., when constructing `Param` objects).

`Any`	The set of all possible values, except `None`
`AnyWithNone`	The set of all possible values
`EmptySet`	The set with no data 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]
`UnitInterval`	The same as 'PercentFraction'
`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.

The `domain` or `within` argument can also accept a function, which is used to define the domain for individual elements of an indexed variable. For example:

```
model.A = Set(initialize=[1,2,3])
def s_domain(model, i):
   return IntegerInterval(bounds=(i,i+1))
model.s = Var(model.A, domain=s_domain)
```

In this example, `s` is an indexed variable whose individual entities are defined over consecutive integer intervals.

NOTE: While Pyomo supports a general representation for restricting the domain of the variables, not all solvers support this general behavior. You may need to restrict your definitions to those supported by the selected solver.

Variable bounds can be explicitly specified with the `bounds` keyword option:

```
model.A = Set(initialize=[1,2,3])
model.a = Var(bounds=(0.0,None))

lower = {1:2.5, 2:4.5, 3:6.5}
upper = {1:3.5, 2:4.5, 3:7.5}
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 in place of the lower or upper bound to indicate that no bound should be enforced. In the code snippet above, `model.a` has a lower bound of 0, and does not have an upper bound, while `model.b` has different bounds for each of its indices. For example, `model.b[3]` has a lower bound of 6.5 and an upper bound of 7.5.

The initial value of variables can be set with the `initialize` keyword argument as in the following example:

```
model.A = Set(initialize=[1,2,3])
model.za = Var(initialize=9.5, within=NonNegativeReals)
model.zb = Var(model.A, initialize={1:1.5, 2:4.5, 3:5.5})
model.zc = Var(model.A, initialize=2.1)

print(value(model.za)) # 9.5
print(value(model.zb[3])) # 5.5
print(value(model.zc[3])) # 2.1
```

For non-indexed variables, a single scalar value is provided to the `initialize` keyword argument. If the component is indexed, a single value can still be provided, in which case all entries in an indexed variable will be initialized to the same value. As well, a dictionary can be passed in where the keys correspond to the valid indices of the variable. Additionally, this argument can be passed a rule (function) that accepts the variable indices and model and returns the value of that variable element:

```
model.A = Set(initialize=[1,2,3])
def g(model, i):
   return 3*i
model.m = Var(model.A, initialize=g)

print(value(model.m[1])) # 3
print(value(model.m[3])) # 9
```

4.3.2 Working with `Var` Objects

When generating formatted output, or scripting advanced workflows, there are several attributes and methods of `Var` that are commonly used. Consider the following declarations:

```
model.A = Set(initialize=[1,2,3])
model.za = Var(initialize=9.5, within=NonNegativeReals)
model.zb = Var(model.A, initialize={1:1.5, 2:4.5, 3:5.5})
model.zc = Var(model.A, initialize=2.1)
```

The current value of the variable can be obtained with the `value()` function, and the attributes `lb` and `ub` hold values for the lower and upper bounds on the variable, respectively. These values may be inferred from the domain of the variable.

```
print(value(model.zb[2])) # 4.5
print(model.za.lb) # 0
print(model.za.ub) # None
```

The `setlb` and `setub` methods are used to set lower and upper bounds for a each variable.

Variable values can be set using the Python assignment operator,

```
model.za = 8.5
model.zb[2] = 7.5
```

One can also call the `set_values` method to set all the variable values from a dictionary.

`Var` components can be fixed to specific values. If the `fixed` attribute is `True`, then the variable has a fixed value that will not be altered by an optimizer. The `fix` method is used to fix elements of a `Var`, and the `unfix` method is used to unfix elements of a `Var`.

```
model.zb.fix(3.0)
print(model.zb[1].fixed) # True
print(model.zb[2].fixed) # True
model.zc[2].fix(3.0)
print(model.zc[1].fixed) # False
print(model.zc[2].fixed) # True
```

4.4 Objectives

An objective is a function 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. The following sections describe the syntax for objective declaring and working with objectives.

4.4.1 `Objective` Declarations

Most solvers can be applied to optimization models with a single objective. The following code provides simple declaration of an `Objective` object:

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

Named and un-named arguments are supported, and Table 4.3 provides a list of the common arguments that can be passed when declaring the `Objective` component.

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`expr`	provides the expression that defines the objective function	any valid Pyomo expression
`rule`	provides the rule function that will be called to provide the expression that defines the objective function	a function that returns a Pyomo expression or `Objective.Skip`
`sense`	determines if the objective is to be minimized or maximized (default is to minimize)	`minimize` or `maximize`

Table 4.3: Common Declaration Arguments for the `Objective` Component

The `expr` keyword can be used to specify the actual expression for the objective. One can also use the `rule` keyword to specify a rule (Python function) that returns an expression. A rule function provides control over how the objective is formed or used to build up the expression. These are illustrated here:

```
model.x = Var([1,2], initialize=1.0)

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

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

Some solvers can perform multi-objective optimization with two or more objectives. Multiple objectives can be declared individually. As well, they can be indexed, and defined using a rule, as shown here:

```
A = ['Q', 'R', 'S']
model.x = Var(A, initialize=1.0)
def d_rule(model, i):
   return model.x[i]**2
model.d = Objective(A, rule=d_rule)
```

When the `Objective` object is declared as an indexed component, Pyomo iterates over all elements of the index set during object construction, passing each set element to the function given as the argument to the `rule` keyword. If multiple sets are specified in an `Objective` declaration, then Pyomo iterates over the cross product of all sets, providing an element for each set to the rule function.

In some contexts, it may be convenient to not define objectives for some index values. If the construction rule returns `Objective.Skip`, then the objective is ignored.

```
def e_rule(model, i):
   if i == 'R':
      return Objective.Skip
   return model.x[i]**2
model.e = Objective(A, rule=e_rule)
```

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 using `sense=maximize`

4.4.2 Working with `Objective` Objects

The objective function contains a few attributes that may be useful for scripting or debugging. The `expr` stores the expression for the objective. The `sense` attribute indicates whether the objective is to minimize or maximize. The `value` function and the `()` operator can be used to compute the value of the objective. These are illustrated in the following example:

```
A = ['Q', 'R']
model.x = Var(A, initialize={'Q':1.5, 'R':2.5})
model.o = Objective(expr=model.x['Q'] + 2*model.x['R'])
print(model.o.expr) # x[Q] + 2*x[R]
print(model.o.sense) # minimize
print(value(model.o)) # 6.5
```

4.5 Constraints

A constraint defines one or more expressions that place limits on the feasible values of variables. The declaration of constraint expressions is similar to the declaration of objective function expressions. However, constraints differ from objectives in

that the expressions include relationships (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.

4.5.1 `Constraint` Declarations

The following code provides a simple declaration of a single, non-indexed `Constraint` object:

```
model.x = Var([1,2], initialize=1.0)
model.diff = Constraint(expr=model.x[2]-model.x[1] <= 7.5)
```

Several named arguments are supported, and Table 4.4 provides a list of the common arguments that can be passed when declaring the `Constraint` component.

The expression specified by the `expr` keyword can alternatively be generated with a rule function. For example, the `diff` constraint can also be declared as follows:

```
model.x = Var([1,2], initialize=1.0)
def diff_rule(model):
    return model.x[2] - model.x[1] <= 7.5
model.diff = Constraint(rule=diff_rule)
```

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`expr`	provides the expression that defines the constraint expressions	any valid Pyomo expression with a relational operator, a 2-tuple, or a 3-tuple
`rule`	the rule function that will be called to provide the expression that defines the constraint function	a function that returns a Pyomo expression with a relational operator, a 2-tuple, a 3-tuple, or `Constraint.Skip`

Table 4.4: Common Declaration Arguments for `Constraint` Component

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, rule=CoverConstr_rule)
```

Indexed constraints are specified in the same manner as indexed objectives. Pyomo iterates over the cross product of all input sets, providing an index for each set to the rule function. The `CoverConstr` constraint in this example implements the following mathematical model:

$$a_i y_i \geq b_i \ \forall i \in \{1,2,3\} \tag{4.1}$$

However, the specific model instance that is passed to the solver will include the following explicit constraints,

$$\begin{aligned} y[1] &\geq 1 \\ 3.1 \cdot y[2] &\geq 2.9 \\ 4.5 \cdot y[3] &\geq 3.1 \end{aligned}$$

Three types of constraint expressions are allowed in Pyomo:

- *inequality constraints* have the form

  $$expr_1 \leq expr_2 \quad \text{or} \quad expr_1 \geq expr_2$$

  where $expr_1$ and $expr_2$ may be non-constant expressions. (Note that $<$ and $>$ are not supported.)
- *equality constraints* have the form

  $$expr_1 = expr_2$$

  where $expr_1$ and $expr_2$ may be non-constant expressions.
- *range constraints* have the form

  $$lower \leq expr_1 \leq upper \quad \text{or} \quad upper \geq expr_1 \geq lower$$

  where $lower$ and $upper$ are constant expressions and $expr_1$ is a non-constant expression.

In some optimization models, a constraint may not be defined for all indices. For example, particular indices may not be physically realizable. The rule function can return `Constraint.Skip` (or

`Constraint.NoConstraint`) to indicate that no constraint is associated with a particular index. For example, the consider the declaration of 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, rule=Pred_rule)
```

The value `Constraint.Skip` indicates that no constraint is being generated, and the corresponding index value is skipped. An alternative to this approach is to construct a sparse index set that specifies only the valid indices in the constraint. However, this may not be practical in complex models.

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. Similarly, the value `Constraint.Infeasible` indicates that the constraint generated by the specified index is infeasible. This might be used, for example, if a particular combination of parameter values produced an invalid constraint. For this value, Pyomo raises an exception to inform the user, because this typically indicates an error in the model.

4.5.2 Working with `Constraint` Objects

After a constraint is declared, the constraint expression is processed to identify the elements of the logical tuple: (*lower*, *body*, *upper*), where the non-constant expressions are pushed to the body. Hence, the `lower` and `upper` attributes are constant expressions or `None`, and the `body` attribute contains a Pyomo expression. If a constraint expression an equality, then the `equality` attribute is `True`, and the `lower` and `upper` attributes have the same value.

The value of the constraint body can be evaluated using the `value` function. Similarly, the `lslack` and `uslack` methods can be used to compute slack values (difference between the current expression value and the lower or upper bound), as shown in the following example:

```
model = ConcreteModel()
model.x = Var(initialize=1.0)
model.y = Var(initialize=1.0)

model.c1 = Constraint(expr= model.y - model.x <= 7.5)
model.c2 = Constraint(expr=-2.5 <= model.y - model.x)
model.c3 = Constraint(expr=-3.0 <= model.y - model.x <= 7.0)

print(value(model.c1.body)) # 0.0

print(model.c1.lslack()) # -inf
print(model.c1.uslack()) # 7.5
print(model.c2.lslack()) # 2.5
print(model.c2.uslack()) # inf
print(model.c3.lslack()) # 3.0
print(model.c3.uslack()) # 7.0
```

4.6 Set Data

A set is a collection of data, possibly including numeric data (e.g., real or integer values) as well as symbolic data (e.g., strings) that is typically used to specify the valid indices for an indexed components. Several classes can be used to define sets in Pyomo models:

`Set`	A generic component for declaring sets
`RangeSet`	A component that defines a range of numbers
`SetOf`	A component that creates a set from external data without copying the data

4.6.1 `Set` Declarations

The following code provides a simple declaration of a `Set` object:

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

Named and un-named arguments are supported, and Table 4.5 provides a list of the common arguments that can be passed when declaring the `Set` component

An indexed set can also be specified by providing other sets or Python lists as un-named arguments in the declaration, and multi-dimensional indexed sets can be declared by including a list of sets as options to the `Set` object:

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

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`initialize`	provides initial values for the variables	a scalar value, Python dictionary, or rule function
`within` `domain`	specifies the valid domain or values for a variable	a Pyomo `Set` object, Python list, or rule function
`ordered`	specifies whether or not order of the set should be preserved	True/False
`virtual`	specifies that a set does not contain data explicitly (creates a set that is typically used for validation only)	True/False
`bounds`	provide lower and upper bounds for the valid values in the set	a 2-tuple, a Python dictionary, or a rule function

Table 4.5: Common Declaration Arguments for `Set` Component

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)
```

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

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

The `initialize` keyword 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)])
```

A Python dictionary can also be used to specify the elements for each index of an indexed set:

```
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)
```

and the `initialize` keyword can also be used to specify a rule function that provides elements for an indexed set. The function accepts the indices and model and returns the set for that set index:

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

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 the set elements that should be omitted. The `filter` keyword can be used to specify a function that returns `True` when an element belongs in a set, and `False` otherwise. For example:

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

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

After an indexed set is constructed in a concrete model, sets can be added for specific indices using the Python equal operator:

```
model.R = Set([1,2,3])
model.R[1] = [1]
model.R[2] = [1,2]
```

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

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

When an element is added to the set `B`, it is checked to confirm that it also belongs to `A`. This ensures that `B` is a subset of `A`.

Validation of set data can also be performed by passing a rule to the `validate` keyword argument. The rule function should return `True` if the element that is passed in belongs in this set, and `False` otherwise (Pyomo will throw an exception). For example, the following `C_validate` function mimics the `within` keyword argument:

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

Finally, note that if both the `within` and `validate` keyword arguments are specified, then the logic specified by both are applied to validate set elements.

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 and preserve the order in which set elements are declared. We can declare a set to be ordered with the `ordered` keyword:

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

Sets may contain data elements that are either singletons or *k*-tuples. The `dimen` keyword is used to specify the expected dimension of the data. The default value is one, indicating that the set will contain singleton data. In some cases, the appropriate value of the dimension can be determined from other keyword values, but in general the user is required to specify this keyword for tuple set data.

Pyomo set components can contain concrete data; however, they can also be *virtual* sets. A virtual set does not contain data explicitly, but it supports operations like set iteration and/or set membership validation. Predefined virtual sets are shown in Table 4.2. Virtual sets can be declared by setting the `virtual` keyword argument to `True`. Virtual sets are typically used for validation (i.e., as arguments to the `within` or `validate` keyword of another set).

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. This option may be inferred from the `within` option, when that set is ordered.

The `RangeSet` component defines an ordered virtual 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` is defined with a single argument, then the argument 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)
```

If a `RangeSet` 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` 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)
```

4.6.2 Working with `Set` Objects

The `len()` function returns the number of elements in the set:

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

print(len(model.A)) # 3
```

The elements can be directly accessed with the `data()` method, which returns the underlying set data. Note that this will be a Python `set` object for simple ordered or unordered sets. For indexed sets, this returns a dictionary of set objects:

```
model.A = Set(initialize=[1,2,3])
model.B = Set(initialize=[3, 2, 1], ordered=True)
model.C = Set(model.A, initialize={1:[1], 2:[1,2]})

print(type(model.A.data()) is set) # True
print(type(model.B.data()) is set) # True
print(type(model.C.data()) is dict) # True
print(sorted(model.A.data())) # [1,2,3]
for index in sorted(model.C.data().keys()):
  print(sorted(model.C.data()[index]))
# [1]
# [1,2]
```

The `clear()` method is used to clear the data in a Pyomo set. The `discard()` and `remove()` methods are used to remove a single element from a set; the `discard()` method ignores elements that are not in the set, while the `remove()` method throws an exception for missing elements.

Set comparison and membership tests are supported with a variety of Python special methods:

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

# Test is an element is in the set
print(1 in model.A) # True

# Test if sets are equal
print([1,2] == model.A) # False

# Test if sets are not equal
print([1,2] != model.A) # True

# Test if a set is a subset of or equal to the set
print([1,2] <= model.A) # True

# Test if a set is a subset of the set
print([1,2] < model.A) # True

# Test if a set is a superset of the set
print([1,2,3] > model.A) # False

# Test if a set is a superset of the set
print([1,2,3] >= model.A) # True
```

Set iteration also works as expected for simple and indexed sets:

```
model.A = Set(initialize=[1,2,3])
model.C = Set(model.A, initialize={1:[1], 2:[1,2]})

print(sorted(e for e in model.A)) # [1,2,3]
for index in model.C:
   print(sorted(e for e in model.C[index]))
# [1]
# [1,2]
```

Ordered sets include a variety of methods that reflect the ordering in the set:

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

print(model.A.first()) # 3
print(model.A.last()) # 1
print(model.A.next(2)) # 1
print(model.A.prev(2)) # 3
print(model.A.nextw(1)) # 3
print(model.A.prevw(3)) # 1
```

The `first()` and `last()` methods respectively return the values of the first and last elements in an ordered set. The `next()` method takes a value and returns the next value in the set. Similarly, the `prev()` method returns the previous value. The `nextw()` and `prevw()` methods operate similarly, except that they wrap around the ends of the set. In this example, the value of `nextw(1)` is 3 because 1 is the last element of the set, and 3 is the next element if the set indices wrap around. The `ord()` method can be used to find the position index of an element in an ordered set, and the `[]` operator can be used to access an element given a position index:

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

print(model.A.ord(3)) # 1
print(model.A.ord(1)) # 3
print(model.A[1]) # 3
print(model.A[3]) # 1
```

Note that position indices start at one. The order of the set is determined by the sequence of the data provided when it is instantiated.

4.7 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 value, an array of values or a multi-dimensional array of values. An unindexed `Param` component looks a lot like a scalar value, and an indexed `Param` component looks a lot like a Python dictionary of values. However, the `Param` component supports advanced features like mutability and sparse representations with default values.

4.7.1 `Param` Declarations

The following code provides a simple declaration of a `Param` object:

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

Named and un-named arguments are supported, and Table 4.6 provides a list of the common arguments that can be passed when declaring the `Param` component.

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`initialize`	provides initial value(s) for the parameter	a scalar value, Python dictionary, or rule function
`default`	provides default value(s) to use for the parameter if no value has been set	a scalar value, Python dictionary, or rule function
`validate`	specifies a function that is called to determine if a particular value is valid for the parameter	a function that returns True or False given a particular value
`mutable`	specifies whether or not the parameter values may change between calls to a solver	True/False

Table 4.6: Common Declaration Arguments for `Param` Component

An indexed parameter can be specified by providing sets as unnamed arguments to the `Param` declaration:

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

The `initialize` keyword can be used to specify the value of a parameter:

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

The `initialize` keyword also accepts a rule function that returns the initial value for a scalar parameter or a value for a specified index of an indexed parameter.

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

If ordered sets are used to define the index for an indexed parameter, then the initialization function can reference previously defined 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]
model.XX = Param(model.A, model.A, initialize=XX_init)
```

The `default` option can be used to specify parameter values for all valid indices that have not been explicitly initialized. For example, we can define an indexed parameter that represents a 3×3 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)
```

A `Param` object can contain any value: string, floating point, etc. The default domain for parameters is `Any`, so by default no domain validation is performed. However, it is often valuable to specify the space of valid parameter values to provide checking of the input data. Similar to the `Set` component, there are two ways to validate parameter values. First, the domain of feasible parameter values can be specified using the `within` option:

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

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 and `False` if it is not (Pyomo will throw an exception). 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 indexed parameters is performed similarly. The `validate` option specifies a function whose arguments are the model, parameter value, and the parameter indices:

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

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

The `Param` component typically represents constant values that can be used in Pyomo models; however, mutability is also supported. In the following example, Pyomo generates the expression for the objective in this model with the form:

$$x_1 + 4x_2 + 9x_3.$$

Specifically, Pyomo has treated parameter values as fixed constants, and its expressions simply contain the numeric constants.

```
model = ConcreteModel()
p = {1:1, 2:4, 3:9}

model.A = Set(initialize=[1,2,3])
model.p = Param(model.A, initialize=p)
model.x = Var(model.A, within=NonNegativeReals)

model.o = Objective(expr=sum(model.p[i]*model.x[i] for i \
    in model.A))
```

Note that this "conversion" happens as soon as the expression is first created. The fact that these values come from a `Param` component is lost, and we only have the numerical values (done for efficiency). Consequently, these values cannot be changed once the expression is created.

However, this behavior is different if the `mutable` option is specified while constructing the model. If this option is `True`, then the parameter values are not treated as constants. Consider the previous example again where the `p` parameter is now mutable:

```
model = ConcreteModel()
p = {1:1, 2:4, 3:9}

model.A = Set(initialize=[1,2,3])
model.p = Param(model.A, initialize=p, mutable=True)
model.x = Var(model.A, within=NonNegativeReals)

model.o = Objective(expr=summation(model.p, model.x))

model.p[2] = 4.2
model.p[3] = 3.14
```

When Pyomo generates the expression for the objective in this model, it keeps knowledge of the `Param` component and now has the form:

$$p_1 x_1 + p_2 x_2 + p_3 x_3,$$

where the values p_i are `Param` objects with references to the parameter values. Here, Pyomo treats the parameter values as mutable values that may later be changed by the user. In this example, the parameter values are changed *after* the objective expression is defined, and the resulting objective is

$$x_1 + 4.2x_2 + 3.14x_3.$$

The parameters are only replaced with their numerical values when calling the solver. Therefore, their values can be changed between consecutive calls to a solver.

Mutable parameters require some additional overhead for memory and they require additional processing when translating Pyomo expressions into a form that a solver understands. Consequently, parameters are immutable by default.

4.7.2 Working with `Param` Objects

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 required to be 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.

As mentioned earlier, a default value can also be provided with the `default` keyword argument. If a default value is provided, and a model tries to access a value that has not been initialized, the default value is used (instead of throwing an exception). Note that the parameter data is stored with a sparse representation, even if the default value is specified. This is supported for memory efficiency. It provides a convenient way for the modeler to reference sparse values without adopting a specialized data structure.

Because of this sparse representation, several methods that consider the valid keys of an indexed parameter require specialized behavior. Let the *valid index set* refer to the complete list of all valid indices (whether initialized or not), and let the *effective index set* denote only the set of initialized key values in an indexed component. If no default value is declared, then the the `len` function returns the size of the effective index set, and the `in` operator tests if a specified value is in the effective index set. Iteration is supported over values in the effective index set, and the Python `[]` operator can be used to access individual elements (which is the parameter value in this example).

If a default value is declared, then all indices are equally valid in the model, whether explicitly indexed or not. Therefore, the `len()` function returns the size of the full index set, iteration and the `in` operator consider the full index set. Thus, when a default value is specified, the parameter appears to be densely populated with values, even if the underlying data structure is kept sparse for efficiency. This is illustrated in the following example:

```
model = ConcreteModel()
model.p = Param([1,2,3], initialize={1:1.42, 3:3.14})
model.q = Param([1,2,3], initialize={1:1.42, 3:3.14}, \
    default=0)

# Demonstrating the len() function
print(len(model.p)) # 2
print(len(model.q)) # 3
```

```
# Demonstrating the 'in' operator (checks against \
    component keys)
print(2 in model.p) # False
print(2 in model.q) # True

# Demonstrating iteration over component keys
print([key for key in model.p]) # [1,3]
print([key for key in model.q]) # [1,2,3]
```

The methods `sparse_keys()`, `sparse_values()`, `sparse_items()`, `sparse_iterkeys()`, `sparse_itervalues()`, and `sparse_iteritems()` define sparse versions of the corresponding methods that are defined in the`IndexedComponent` class. These methods return values only for the defined parameter values, whether or not a default value is specified.

4.8 Named Expressions

Pyomo expressions are mathematical statements that contain numbers, parameters, and variables combined using operators such as +,−,∗,/, etc. Such expressions form the basis of the algebraic representation of a model, and are stored inside constraint and objective components on that model.

The `Expression` component provides a mechanism for storing a Pyomo expression on a model so that the expression can be re-used in multiple contexts, such as a common sub-expression in one or more constraints, without the overhead of regenerating the expression each time. In addition, the Pyomo expression stored by the `Expression` component can be changed at a later time, thereby updating any constraint or objective expressions that reference it. This provides a powerful approach for modifying a model between calls to a solver.

The following sections describe the syntax for declaring and working with named expressions.

4.8.1 `Expression` Declarations

The following code provides a simple declaration of a single, non-indexed `Expression` object:

```
model.e = Expression()
```

Named and un-named arguments are supported, and Table 4.7 provides a list of the common arguments that can be passed when declaring the `Expression` component.

The `expr` or `rule` keywords can be used to initialize a named expression when it is declared, as shown in the following example:

```
model.x = Var()
model.e1 = Expression(expr=model.x + 1)
def e2_rule(model):
   return model.x + 2
model.e2 = Expression(rule=e2_rule)
```

keyword	description	acceptable values
<un-named>	reserved for specifying index sets	any number of Pyomo `Set` objects or Python lists
`expr`	provides the expression to store	any valid Pyomo expression
`rule`	the rule function that will be called to provide the expression to store	a function that returns a Pyomo expression or Expression.Skip

Table 4.7: Common Declaration Arguments for `Expression` Component

As with the other core modeling components, the `Expression` component can be indexed by declaring it with one or more arguments that represent indexing sets. The following example declares an indexed `Expression` component over all members of the index set except for the first. Indices that should be left out of the indexed `Expression` container are signified by returning the `Expression.Skip` attribute from the initialization rule.

```
N = [1,2,3]
model.x = Var(N)
def e_rule(model, i):
   if i == 1:
      return Expression.Skip
   else:
      return model.x[i]**2
model.e = Expression(N, rule=e_rule)
```

4.8.2 Working with `Expression` Objects

A simple use for the `Expression` component declares a single expression and uses it inside an objective and a constraint declaration:

```
model.x = Var()
model.e = Expression(expr=(model.x - 1.0)**2)
model.o = Objective(expr=0.1*model.e + model.x)
model.c = Constraint(expr=model.e <= 1.0)
```

The value of the named expression can be computed using the `value` function. Additionally, the expression stored in the named Expression component can be up-

dated. As the following example shows, updating the named expression has the effect of updating the objective and constraint expressions where it is used:

```
model.x.set_value(2.0)
print(value(model.e)) # 1.0
print(value(model.o)) # 2.1
print(value(model.c.body)) # 1.0

model.e.set_value((model.x - 2.0)**2)
print(value(model.e)) # 0.0
print(value(model.o)) # 2.0
print(value(model.c.body)) # 0.0
```

The `Expression` component does not require an expression when it is declared on a model, but it must be assigned one before the model is solved if the named expression is used in any active objectives or constraints. Furthermore, named expressions that are used in objectives or constraints should not store relational Pyomo expressions, that is, expressions using one or more of the operators <=, <, >=, >, and ==.

4.9 Suffix Components

Suffixes provide a mechanism for annotating a model with auxiliary data that is not strictly related to the model declaration and structure. Suffixes are commonly used by solver plugins to store extra information about the solution of a model. More generally, suffixes can be used to

- import information from a solver about the solution to a mathematical program (e.g., constraint duals, variable reduced costs, basis information),
- export information to a solver or algorithm to configure the solution process (e.g., warm-starting information, variable branching priorities), and
- tag model components with local data for later use in advanced scripting algorithms.

This functionality is made available to the modeler through the `Suffix` component class, which provides an interface for annotating Pyomo modeling components with additional data.

4.9.1 `Suffix` Declarations

The following code provides a simple declaration of a suffix labeled `foo`:

```
model.foo = Suffix()
```

Named and un-named arguments are supported, and Table 4.8 provides a list of the common arguments that can be passed when declaring the `Suffix` component

keyword	description	acceptable values
`direction`	specifies if a suffix is an input to or an output from a solver	`Suffix.LOCAL`, `Suffix.IMPORT`, `Suffix.EXPORT`, `Suffix.IMPORT_EXPORT` (more details given below)
`datatype`	specifies the particular type of data being stored in the suffix	`Suffix.FLOAT`, `Suffix.INT`, `None` (more details given below
`initialize`	provides initial values for the suffix	a rule function

Table 4.8: Common Declaration Arguments for `Suffix` Component

The `Suffix` component is a not an indexed component, and hence it cannot be declared with un-named positional arguments. The `direction` keyword argument is used to specify the information flow for a suffix when interfacing with a solver. This argument can be one of four possible values:

- `Suffix.LOCAL`: Suffix data is local to the model. It is not imported or exported by solver plugins (default).
- `Suffix.IMPORT`: Suffix data will be imported from solvers to the model by solver plugins.
- `Suffix.EXPORT`: Suffix data will be exported from the model to the solver by the plugins.
- `Suffix.IMPORT_EXPORT`: Suffix data is both imported and exported by solver plugins.

Not all solver plugins are guaranteed to manage suffix information flow, but the user controls this information flow by configuring suffix components.

The `datatype` keyword argument specifies the type of data that will be held in the suffix. This argument can be one of three possible values:

- `Suffix.FLOAT`: floating point data (default).
- `Suffix.INT`: integer data.
- `None`: any type of data.

This argument may be optional for a solver interface, but exporting suffix data with solvers that use Pyomo's `nl` file interface requires that all active export suffixes have a strict datatype (i.e., the `datatype` keyword cannot be `None`).

The following example illustrates various suffix declarations:

```
# Export integer data
model.priority = Suffix(direction=Suffix.EXPORT,
                        datatype=Suffix.INT)

# Export and import floating point data
model.dual = Suffix(direction=Suffix.IMPORT_EXPORT)
```

Suffixes are not guaranteed to be compatible with all solver plugins in Pyomo. Whether a given suffix is acceptable or not depends on both the solver and solver interface being used. In some cases, a solver plugin will raise an exception if it encounters a suffix type that it does not handle, but this is not true in every situation. For example, the `nl` file interface is generic to all AMPL-compatible solvers, so there is no way to validate that a suffix of a given name, direction, and datatype is appropriate for a solver. One should be careful in verifying that suffix declarations are being handled as expected when switching to a different solver or solver interface.

The `initialize` keyword argument can be used to define suffix values. This argument specifies a function that is executed when the model is constructed. This function returns a list or iterable of (component, value) tuples.

```
model = AbstractModel()
model.x = Var()
model.c = Constraint(expr=model.x >= 1)

def foo_rule(m):
   return ((m.x, 2.0), (m.c, 3.0))
model.foo = Suffix(initialize=foo_rule)
```

4.9.2 Working with Suffixes

Consider the following example:

```
model = ConcreteModel()
model.x = Var()
model.y = Var([1,2,3], dense=True)
model.foo = Suffix()
```

This examples includes two variable components (indexed and non-indexed) along with a suffix component. Conceptually, the declaration of the suffix `foo` allows the association of `foo` with each component in the model. For example:

```
# Assign the value 1.0 to suffix 'foo' for model.x
model.x.set_suffix_value('foo', 1.0)

# Assign the value 2.0 to suffix model.foo for model.x
model.x.set_suffix_value(model.foo, 2.0)

# Get the value of suffix 'foo' for model.x
print(model.x.get_suffix_value('foo')) # 2.0
```

Suffix values can be assigned with `set_suffix_value` and they can be accessed with `get_suffix_value`. This example illustrates two ways of specifying the same suffix: with a name and with a suffix component object.

Suffix values for indexed components can also be assigned with `set_suffix_value`:

```
# Assign the value 3.0 to suffix model.foo for model.y
model.y.set_suffix_value(model.foo, 3.0)

# Assign the value 4.0 to suffix model.foo for model.y[2]
model.y[2].set_suffix_value(model.foo, 4.0)

# Get the value of suffix 'foo' for model.y
print(model.y.get_suffix_value(model.foo)) # None
print(model.y[1].get_suffix_value(model.foo)) # 3.0
print(model.y[2].get_suffix_value(model.foo)) # 4.0
print(model.y[3].get_suffix_value(model.foo)) # 3.0
```

This example illustrates how `set_suffix_value` is used to set the value for an indexed component and a single component data object. When `set_suffix_value` is called for an indexed component, by default it sets suffix values for all elements or indices of the component, rather than the component itself. Because of this, when we try to retrieve the suffix value for the `model.y` component, we find that is is `None`.

Suffix values can also be cleared, which is equivalent to setting the value `None`:

```
model.y.clear_suffix_value(model.foo, expand=False)
model.y[3].clear_suffix_value(model.foo)

print(model.y.get_suffix_value(model.foo)) # None
print(model.y[1].get_suffix_value(model.foo)) # 3.0
print(model.y[2].get_suffix_value(model.foo)) # 4.0
print(model.y[3].get_suffix_value(model.foo)) # None
```

4.10 Build Components

When solving a model with our own Python script (see Chapter 14), one can insert Python code anywhere in the process. One can, for example, fix a particular combination of variables, print the value of a parameter, or throw an exception if a particular combination of parameter values is not valid. However, when building an `AbstractModel` and solving the problem with the `pyomo` command, one does not have control over the workflow of the solution process. Fortunately, Pyomo supports a set of components that allow for execution of Python code during the build process.

The `BuildAction` component can be defined in the model to inject actions (defined through Python code) into the model construction process. Similarly, the `BuildCheck` component is used to perform a user-defined test (again, through Python code) during the model construction process and halt construction if the test fails. These components are added to a model in the same manner as other components, but their role is to allow a user to insert scripting-like code into the model construction process.

Consider the following abstract model (defined in `buildactions.py`) that

illustrates the use of `BuildAction` and `BuildCheck` components to define error checks and diagnostic output based on our warehouse location example defined in Sections 3.2 and 3.3.7.

```
# buildactions.py: Warehouse location problem showing \
    build actions
from pyomo.environ import *

model = AbstractModel()

model.N = Set() # Set of warehouses
model.M = Set() # Set of customers
model.d = Param(model.N,model.M)
model.P = Param()

model.x = Var(model.N, model.M, bounds=(0,1))
model.y = Var(model.N, within=Binary)

def checkPN_rule(model):
   return model.P <= len(model.N)
model.checkPN = BuildCheck(rule=checkPN_rule)

def obj_rule(model):
   return sum(model.d[n,m]*model.x[n,m] for n in model.N \
       for m in model.M)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(model, m):
   return sum(model.x[n,m] for n in model.N) == 1
model.one_per_cust = Constraint(model.M, \
    rule=one_per_cust_rule)

def warehouse_active_rule(model, n, m):
   return model.x[n,m] <= model.y[n]
model.warehouse_active = Constraint(model.N, model.M, \
    rule=warehouse_active_rule)

def num_warehouses_rule(model):
   return sum(model.y[n] for n in model.N) <= model.P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

def printM_rule(model):
   model.M.pprint()
model.printM = BuildAction(rule=printM_rule)
```

In this example, we have added a `BuildCheck` component with the rule `CheckPN_rule`. This rule will check to make sure that the total number of warehouses we can place is not more than the number of available warehouse locations. We have also added a `BuildAction` component with the rule `printM_rule` that prints the set of customer locations.

We created a `.dat` file where the parameter P is larger than the available number of warehouse locations (so it would fail the `CheckPN_rule` build check:

```
# buildactions_fails.dat: Pyomo format data file for the \
    warehouse location problem
# Note: parameter P is larger than the number of warehouse \
    locations

set N := Harlingen Memphis Ashland ;
set M := NYC LA Chicago Houston;

param d :=
    Harlingen NYC 1956
    Harlingen LA  1606
    Harlingen Chicago  1410
    Harlingen Houston  330
    Memphis NYC  1096
    Memphis LA  1792
    Memphis Chicago  531
    Memphis Houston  567
    Ashland NYC  485
    Ashland LA  2322
    Ashland Chicago  324
    Ashland Houston  1236
;

param P := 4 ;
```

Solving this with the `pyomo` command:

```
pyomo solve --solver=glpk buildactions.py \
    buildactions_fails.dat
```

gives us output similar to the following:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
ERROR: Constructing component 'checkPN' from data=None
    failed:
        ValueError: BuildCheck 'checkPN' identified error
[    0.01] Pyomo Finished
ERROR: Unexpected exception while running model:
        BuildCheck 'checkPN' identified error
```

As with other components, the `BuildAction` and `BuildCheck` components can be indexed, which allows actions and checks to be customized based on specific data.

4.11 Other Modeling Components

This chapter presented details for some of the most common modeling components supported by Pyomo. There are other modeling components that were not thoroughly discussed in this chapter. These include:

`Block`: The `Block` component provides a mechanism to declare models with repeated or nested structure (e.g., separate `Block` objects may exist on a model to represent different time points in a multi-period optimization). A `Block` consists of a collection of Pyomo modeling components. More discussion of blocks is provided in Chapter 8.

`Model`: The `Model` component provides a container for grouping Pyomo modeling components to form the definition of an optimization problem. Pyomo supports both abstract and concrete modeling representations. While "model" objects were widely used in this book (they are required to formulate and solve an optimization problem in Pyomo), we have not discussed the fact they are components themselves. In fact, they inherit from the `Block` component.

`Complementarity`: This component is used to define complementarity conditions in a mathematical program with equilibrium constraints (MPEC). Several forms of the complementarity conditions are supported. This component is documented further in Chapter 12.

`Disjunct`: This component supports the Generalized Disjunctive Programming (GDP) capability within Pyomo. A `Disjunct` component is a container for an indicator variable and a set of constraints that should be active when that indicator variable is True. This component is documented further in Chapter 9.

`Disjunction`: This component supports the Generalized Disjunctive Programming (GDP) capability within Pyomo. A `Disjunction` component contains a set of `Disjunct` objects connected by a logical "OR" operator. This component is documented further in Chapter 9.

`SubModel`: A `SubModel` component is used as part of the bilevel optimization capability within Pyomo. It is used to define a subproblem for the lower-level decisions in a bilevel programming problem. This component is documented further in Chapter 13.

`Piecewise`: This component supports piecewise modeling of general functions. It supports several different transformations to produce mixed-integer representations for the piecewise functions. More documentation on this component can be found at the Pyomo website.

`SOSConstraint`: Special ordered sets (SOS) can be defined in Pyomo through the `SOSConstraint` component. Pyomo supports special ordered sets of type 1 and 2 (SOS1 and SOS2). More documentation on this component can be found at the Pyomo website.

Chapter 5
The Pyomo Command

Abstract This chapter describes Pyomo's command-line interface, which includes a variety of subcommands that support common workflows and provide information about Pyomo and its installation.

5.1 Overview

The Pyomo software includes the `pyomo` command, which was introduced in Section 3.4.1. The `pyomo` command provides a collection of subcommands that support common workflows and provide information about Pyomo and its installation. The following subcommands are supported in Pyomo 5.1:

`check`
: This subcommand checks a model for errors. This is particularly useful for evaluating the logic of rules in abstract models.

`convert`
: This subcommand is used to convert a Pyomo model into another format, such as an `lp` or `nl` file.

`help`
: Print information about the configuration and installation of Pyomo. For example, the `-s` option is particularly useful:

```
pyomo help -s
```

: This command interrogates the local environment to provide information about available solvers.

`install-extras`
: Install "extra" packages that Pyomo can leverage. For example, this subcommand installs `pyyaml`, which adds YAML support within Pyomo. Also, this subcommand installs `suds`, which is needed to launch solvers on NEOS.

`run`
: Execute a command from the Pyomo bin (or Scripts) directory. For example, this provides a handy mechanism for launching Python with Pyomo installed:

```
pyomo run python
```

`solve`
: Construct and optimize a model.

`test-solvers`
: Execute a variety of tests to verify solver capabilities.

The following sections illustrate the use of the `check`, `convert`, `help` and `solve` subcommands, which can be customized with a variety of options.

5.2 The `check` Subcommand

Pyomo models are regular Python files that define a Pyomo model. Consequently, there are many possible ways that a user might either fail to correctly define a model or define a model in a manner that is not supported by Pyomo. An example of the former is to incorrectly refer to a model variable:

```
# bad1.py
from pyomo.environ import *

model = AbstractModel()
model.A = Set(initialize=[1,2,3])
model.x = Var(model.A)

def x_rule(M):
    return sum(M.x[i] for i in model.A) >= 0
model.c = Constraint(rule=x_rule)

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

The rule function references the variable `model`, but the model instance passed into the rule is `M`. Consequently, the attempt to iterate over the set `model.A` generates an error, because the `model` object has not been initialized with data.

An example of the latter problem is illustrated by the following example:

```
# bad2.py
from pyomo.environ import *

model = AbstractModel()
model.q = Param(initialize=0, mutable=True)
model.A = Set(initialize=[1,2,3])
model.x = Var(model.A)

def x_rule(model):
    if model.q > 0:
        return sum(model.x[i] for i in model.A) >= 1
```

```
    else:
        return sum(model.x[i] for i in model.A) >= 0
model.c = Constraint(rule=x_rule)

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

Here, a mutable parameter q is used in a conditional expression. Since q is mutable, Pyomo does not treat it as a constant value, but rather it generates a Pyomo expression object. But this creates an error because Pyomo cannot implicitly evaluate this expression.

Both of these examples illustrate how subtle errors can arise in Pyomo models. The check subcommand provides a facility for automatically scanning a model file for these types of errors. For example, the command

```
pyomo check bad1.py
```

generates the following output:

```
[model::ModelAccess] bad1.py:9: Expression 'model.A' may
    access a model variable that is outside of the function
    scope
[model::ModelArgument] bad1.py:8: Model variable 'model' is
    used in the rule, but this variable is not first
    argument in the rule argument list
```

Pyomo supports a variety of checkers, each of which prints a warning with an associated line number and other context information. The command

```
pyomo help --check
```

describes the checkers that are installed with Pyomo.

5.3 The convert Subcommand

Many optimizers supported by Pyomo read a a temporary file that Pyomo generates in a standard problem format. For example, 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.

It is often useful to generate these problem files directly, both to diagnose issues with a model as well as to directly manage the execution of a solver. The convert subcommand can be used to convert a Pyomo model into a standard file format. For example, consider the command:

```
pyomo convert --format=lp concrete1.py
```

This command converts the model in concrete1.py into an LP file format, which is stored in the file unknown.lp:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
Model written to file 'unknown.lp'
[    0.00] Pyomo Finished
```

The `--output` option can also be used to specify a filename, for which the filename suffix specifies the file format. For example, the command

```
pyomo convert --output=concrete1.lp concrete1.py
```

creates the file `concrete1.lp`, which represents the model from `concrete1.py` in the `LP` file format.

The command

```
pyomo help -w
```

summarizes the file formats supported by Pyomo.

5.4 The `help` Subcommand

The `help` subcommand prints information about Pyomo's capabilities, including information about installed plugins as well as available solvers. The `-h` option prints information about the different information that is available, including:

`--checkers`
: This prints the model checkers that are installed with Pyomo.

`--commands`, `-c`
: This prints the commands that are installed with Pyomo. Although much of Pyomo's functionality can be accessed through the `pyomo` command, some functionality is developed separately. Currently, most of commands supported by Pyomo relate to the functionality in `pyomo.pysp`.

`--components`
: This prints the modeling components and virtual sets that are available in Pyomo.

`--data-managers`, `-d`
: This prints the data interfaces that are supported by the `DataPortal` class. Data can also be imported through these interfaces using Pyomo data files.

`--info`, `-i`
: This prints information about the user's PATH environment and Python installation. This command helps diagnose issues with the execution of Pyomo.

`--solvers`, `-s`
: This prints information about solvers and solver managers that can be used the optimize Pyomo models. Note that information about NEOS solvers will be included if Pyomo can connect to the NEOS server.

`--transformations`, `-t`
: This prints the model transformations that are supported by Pyomo.

--writers, -w
This prints the model writers that are supported by Pyomo. Specifically, this summarizes the different file formats that a Pyomo model can be converted to.

5.5 The solve Subcommand

The solve subcommand 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.
7. Display the solver results.

The model construction step requires a Pyomo model file, which is a Python file that defines a Pyomo model object. Thus, the solve subcommand 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 wl_concrete.py defined in Section 3.3.5 using the glpk solver:

```
pyomo solve --solver=glpk wl_concrete.py
```

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

```
pyomo solve --solver=glpk wl_abstract.py wl_data.dat
```

The solve subcommand 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.

However, the solve subcommand can also be executed with a YAML or JSON configuration file[1], which eliminates the need to specify command-line options. Consider the following configuration file:

```
# concrete1.yaml
model:
  filename: concrete1.py
solvers:
  - solver name: glpk
```

This configuration file can be used to configure the executions of the pyomo subcommand as follows:

[1] YAML and JSON are data serialization standards. JSON is supported natively in Python, and information about JSON is available at www.json.org. YAML configuration files are supported if the PyYAML package is installed, and information about YAML is available at www.yaml.org.

```
pyomo solve concrete1.yaml
```

This configuration file defines the same logic as the first command in the previous paragraph, and the following configuration file defines the same logic as the second command:

```
# abstract5.yaml
model:
  filename: abstract5.py
data:
  files:
    - abstract5.dat
solvers:
  - solver name: glpk
```

No command-line options are required when using a configuration file, because all command-line options have corresponding elements in a configuration file. Furthermore, there are configuration options that can only be expressed in a configuration file. A template configuration file can be generated with the `--generate-config-template` option .

The `--help` and `--generate-config-template` options for the `solve` subcommand require the `--solver` option. These two options provide solver-specific summaries respectively for command-line options and configuration files. For example, you could execute the following command to get command-line options that are suitable for the `glpk` solver:

```
pyomo solve --solver=glpk --help
```

Table 5.1 summarizes key options for the `solve` subcommand that are commonly used.

5.5.1 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; a model file can execute an arbitrary Python script, but the expectation is that it generate an object that contains the Pyomo model. Within the `solve` subcommand, 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 following simple LP:

Option	Description
-c, --catch-errors	Trigger failures for exceptions and print the program stack.
--json	Store results in JSON format.
-k, --keepfiles	Keep temporary files.
-l, --log	Print the solver logfile after performing optimization.
--logfile FILE	Redirect output to the specified file.
--logging LEVEL	Specify the logging level: quiet, warning, info, verbose, debug.
--model-name NAME	The name of the model object that is created in the specified Pyomo module.
--path PATH	Give a path that is used to find Pyomo python files.
--report-timing	Report various timing statistics during model construction.
--results-format FORMAT	Specify the results format: json or yaml.
--save-results FILE	The filename to which the results are saved.
--show-results	Print the results object after optimization.
--solver SOLVER	Specify the solver name.
--solver-executable FILE	The executable used by the solver interface.
--solver-io FORMAT	The type of IO used to execute the solver. Different solvers support different types of IO, but the following are common options: lp - generate LP files, nl - generate NL files, python - direct Python interface.
--solver-manager TYPE	The technique that is used to manage solver executions.
--stream-output	Stream the solver output to provide information about the solver's progress.
--solver-options STRING	String describing solver options.
--solver-suffix SUFFIXES	Solution suffixes that will be extracted by the solver (e.g., rc, dual, or slack).
--summary	Summarize the final solution after performing optimization.
--symbolic-solver-labels	When interfacing with the solver, use symbol names derived from the model. For example, "my_special_variable[1_2_3]" instead of "v1". When using the ASL solvers, this option generates corresponding .row (constraints) and .col (variables) files.
--tempdir TEMPDIR	Specify the directory where temporary files are generated.

Table 5.1: Commonly used options for the pyomo solve subcommand.

```
# abstract5.py
from pyomo.environ 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)
```

This is an abstract Pyomo model that 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. For example, we can adapt the previous example to store the model in `Model`:

```
# abstract6.py
from pyomo.environ 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)
```

This model can be optimized with the following command:

```
pyomo solve --solver=glpk --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 `solve` subcommand is executed.

5.5.2 Selecting Data with Namespaces

Section 6.7 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 `solve` subcommand 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 solve --solver=glpk --namespace=data1 abstract5.py \
                  abstract5-ns1.dat
```

creates and optimizes the abstract model in abstract5.py using the following data commands:

```
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 ;
}
```

This command specifies the data1 namespace, which has an optimal solution of 0.8. Similarly, the command

```
pyomo solve --solver=glpk --namespace=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 following data commands:

```
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 ;
}
```

This includes four namespaces and data commands outside of a namespace. The command

```
pyomo solve --solver=glpk --namespace=c1 --namespace=data2 \
            abstract5.py abstract5-ns2.dat
```

creates and optimizes the abstract modeling in abstract5.py 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.

5.5.3 Customizing Pyomo's Workflow

The different steps that are executed by the solve subcommand represent a generic workflow for model construction and optimization. This workflow can be customized using a variety of callback functions that are defined within a Pyomo model file. These callback functions allow the user to define additional analysis steps, as well as replace some of the default steps in the workflow.

Table 5.2 summarizes the callback functions and the functionality that they support. Each callback function takes one or more keyword arguments in the form keyword = value. Consequently, there are two different ways that a callback function can be defined in Python. Consider the pyomo_print_results callback 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 callback 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 callback 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 callback functions will always pass in their expected arguments, so no additional error checking is required.

There are several standard arguments for the callback functions described in Table 5.2. The options argument is an enhanced Python dictionary that contains the command-line options sent to the solve subcommand. 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 ConcreteModel, then the model and instance arguments are the same object. Other arguments are described with their associated callback functions.

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 `DataPortal` 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 5.2: Callback functions that can be used in a Pyomo model file to customize the workflow in the `solve` subcommand.

pyomo_preprocess

This callback function is executed before model construction to perform preprocessing steps. This function has one argument: `options`. For example, the following callback function simply prints the command-line options:

```
def pyomo_preprocess(options=None):
   print("Here are the options that were provided:")
   if options is not None:
      options.display()
```

pyomo_create_model

This callback 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 callback 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):
   sys.path.append(abspath(dirname(__file__)))
   abstract6 = __import__('abstract6')
   sys.path.remove(abspath(dirname(__file__)))
   return abstract6.Model
```

pyomo_create_modeldata

This callback 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 DataPortal object. For example, the following callback function creates a DataPortal object from the file abstract6.dat:

```
def pyomo_create_dataportal(options=None, model=None):
   data = DataPortal(model=model)
   data.load(filename='abstract6.dat')
   return data
```

pyomo_print_model

This callback 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['runtime']['logging']:
      model.pprint()
```

pyomo_modify_instance

This callback function modifies the model instance after it has been constructed. This function has three arguments: options, model, and instance. The following callback fixes a variable after the model is constructed:

```
def pyomo_modify_instance(options=None, model=None,
                                instance=None):
   instance.x[1].value = 0.0
   instance.x[1].fixed = True
```

pyomo_print_instance

This callback 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['runtime']['logging']:
      instance.pprint()
```

pyomo_save_instance

This callback 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')
   OUTPUT.write(str(pickle.dumps(instance)))
   OUTPUT.close()
```

pyomo_print_results

This callback 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 callback function can simply print this data:

```
def pyomo_print_results(options=None, instance=None,
                                  results=None):
   print(results)
```

However, the `solve` subcommand includes the `--print-results` option, which performs this operation. More generally, this callback function is included to allow users to provide problem-specific summaries of their optimization results.

pyomo_save_results

This callback function is used to save the results generated from optimization. This function has three arguments: `options`, `instance`, and `results`. This callback function can simply print the results to a file:

```
def pyomo_save_results(options=None, instance=None,
                                  results=None):
   OUTPUT = open('abstract7.results','w')
   OUTPUT.write(str(results))
   OUTPUT.close()
```

The `solve` subcommand includes the `--save-results` option, which performs this operation. More generally, this callback function is included to allow users to save problem-specific summary of their optimization results.

pyomo_postprocess

This callback 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):
    instance.solutions.load_from(results, \
             allow_consistent_values_for_fixed_vars=True)
    print("Solution value "+str(value(instance.obj)))
```

5.5.4 Customizing Solver Behavior

The generic workflow supported by the solve subcommand 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 solve subcommand 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.

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 solve --solver=glpk --solver-options='mipgap=0.01' \
    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 information 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 solve subcommand, 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 solve --solver=glpk --solver-suffix='.*' 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.

5.5.5 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.

5.5.6 Managing Diagnostic Output

The `solve` subcommand 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 `solve` subcommand 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 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.

```
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']
      for obj in results.solution[i].objective:
         f[i] = results.solution[i].objective[obj]['Value']
         break
   #
   # 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")
   OUTPUT = open('results.csv', 'w')
   writer = csv.writer(OUTPUT)
   writer.writerows(rows)
   OUTPUT.close()
```

Fig. 5.1: A post-processing plugin that writes final solutions in a CSV file.

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 `--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 Pyomo and PyUtilib. By default, logging messages that represent Pyomo 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 Pyomo and PyUtilib. The `--info` option enables informative, warning and error log messages for Pyomo and PyUtilib.

The `--verbose` option enables debugging log messages for Pyomo and PyUtilib. This option can be specified multiple times to enable logging messages for different parts of Pyomo and PyUtilib: (1) debugging for just Pyomo, (2) debugging for all Pyomo packages, and (3) debugging for all Pyomo and PyUtilib packages. The `--debug` option enables debugging logging, and it allows exceptions to trigger a failure in which the program stack is printed.

5.6 Discussion

The long-term goal is to make the `pyomo` command the single portal for all workflows that are provided with Pyomo. Consequently, scripts like `pyomo2lp` have been removed and replaced with the `convert` subcommand. This integration has been completed for most packages in Pyomo. The major exception is the `pyomo.pysp` package, which has a variety of custom scripts (e.g., `runph`) that will be eventually supported within the `pyomo` command.

Checkers are probably most useful for abstract models and simple concrete models. Python scripts that perform complex operations on concrete models could also be analyzed with this subcommand, but errors in such scripts could be much harder to detect. Model checking is particularly useful for new users, so this may be the default in future versions of Pyomo.

Chapter 6
Data Command Files

Abstract Data command files allow users to define set and parameter data with a high-level language. 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, including table files, CSV files, spreadsheets, and databases.

6.1 Model Data

The `Set` and `Param` components of a Pyomo model are used to define data values used to construct constraints and objectives. Previous chapters have illustrated that these components are not necessary to develop complex models. However, The `Set` and `Param` components can be used to define abstract data declarations, where no data values are specified. For example:

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

Data command files can be used to initialize data declarations in Pyomo models, and in particular they are useful for initializing abstract data declarations.

Pyomo's data command files employ a domain-specific language whose syntax closely resembles the syntax of AMPL's data commands [2]. A data command file consists of a sequence of commands that specify set and parameter data, or specify where such data is to be obtained from external sources. The following 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 the parameter data.
- The `table` command declares a two-dimensional table of parameter data.

- The `load` command defines how set and parameter data is loaded from external data sources, including ASCII table files, CSV files, XML files, YAML files, JSON 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 processed immediately.
- The `data` and `end` commands do not perform any actions, but they provide compatibility with AMPL scripts that define data commands.

Finally, 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 closely match AMPL's syntax and semantics. However, these commands only support a subset of the corresponding declarations in AMPL. However, it is not possible to support other AMPL commands because Pyomo treats data commands as data declarations while AMPL treats data 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 `load` command supports much simpler mappings. Complex mappings are accomplished in Pyomo via scripting.

The remaining sections in this chapter describe the syntax Pyomo's 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 data commands 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:

```
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 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 five 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 * 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;
```

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.

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.

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 while using different index values.

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 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 the 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
;
```

Similarly, 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 `table` Command

The `table` data command explicitly specifies a two-dimensional array of parameter data. This command provides a more flexible and complete data declaration than

is possible with a `param` declaration. This command has a similar syntax to the `load` command, but it includes a complete specification of the table data.

The following example illustrates a simple `table` command that declares data for a single parameter:

```
table M(A) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

The parameter `M` is indexed by column `A`, which must be pre-defined unless declared separately (see below). The column labels are provided after the colon and before the `:=`. Subsequently, the table data is provided. Note that the syntax is not sensitive to whitespace. Thus, the following is an equivalent `table` command:

```
table M(A) :
A  B  M   N :=
A1 B1 4.3 5.3 A2 B2 4.4 5.4 A3 B3 4.5 5.5 ;
```

Multiple parameters can be declared by simply including additional parameter names. For example:

```
table M(A) N(A,B) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

This example declares data for the `M` and `N` parameters. As this example illustrates, these parameters may have different indexing columns.

The indexing columns represent set data, which is specified separately. For example:

```
table A={A} Z={A,B} M(A) N(A,B) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

This example declares data for the `M` and `N` parameters, along with the `A` and `Z` indexing sets. The correspondence between the index set `Z` and the indices of parameter `N` can be made more explicit by indexing `N` by `Z`:

```
table A={A} Z={A,B} M(A) N(Z) :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Set data can also be specified independent of parameter data:

```
table Z={A,B} Y={M,N} :
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

NOTE: If a table command does not explicitly indicate the indexing sets, then these are assumed to be initialized separately. A table command can separately initialize sets and parameters in a Pyomo model, and there is no presumed association between the data that is initialized. For example, the table command initializes a set Z and a parameter M that are not related:

```
table Z={A,B} M(A):
A  B  M   N :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Finally, simple parameter values can be specified with a simple table command:

```
table pi := 3.1416 ;
```

This

The previous examples considered examples of the table command where column labels are provided. The table command can also be used without column labels. For example, the first example can be revised to omit column labels as follows:

```
table columns=4 M(1)={3} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

The columns=4 is a keyword-value pair that defines the number of columns in this table; this must be explicitly specified in unlabeled tables. The default column labels are integers starting from 1; the labels are columns 1, 2, 3, and 4 in this example. The M parameter is indexed by column 1. The braces syntax declares the column where the M data is provided.

Similarly, set data can be declared referencing the integer column labels:

```
table columns=4 A={1} Z={1,2} M(1)={3} N(1,2)={4} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Declared set names can also be used to index parameters:

```
table columns=4 A={1} Z={1,2} M(A)={3} N(Z)={4} :=
A1 B1 4.3 5.3
A2 B2 4.4 5.4
A3 B3 4.5 5.5
;
```

Finally, we compare and contrast the `table` and `param` commands. Both commands can be used to declare parameter and set data, and both commands can be used to declare a simple parameter. However, there are some important differences between these data commands:

- The `param` command can declare a single set that is used to index one or more parameters. The `table` command can declare data for any number of sets, independent of whether they are used to index parameter data.
- The `param` command can declare data for multiple parameters only if they share the same index set. The `table` command can declare data for any number of parameters that are may be indexed separately.
- The `table` syntax unambiguously describes the dimensionality of indexing sets. The `param` command must be interpreted with a model that provides the dimension of the indexing set.

This last point provides a key motivation for the `table` command. Specifically, the `table` command can be used to reliably initialize concrete models using Pyomo's `DataPortal` object. By contrast, the `param` command can only be used to initialize concrete models with parameters that are indexed by a single column (i.e., a simple set).

6.5 The `load` Command

The `load` command provides a mechanism for loading data from a variety of external tabular data sources. This command loads a table of data that represents set and parameter data in a Pyomo model. The table consists of rows and columns for which all rows have the same length, all columns have the same length, and the first row represents labels for the column data.

The `load` command can load data from a variety of different external data sources:

- **TAB File**: A text file format that uses whitespace to separate columns of values in each row of a table.
- **CSV File**: A text file format that uses comma or other delimiters to separate columns of values in each row of a table.
- **XML File**: An extensible markup language for documents and data structures. XML files can represent tabular data.
- **Excel File**: A spreadsheet data format that is primarily used by the Microsoft Excel application.

- **Database**: A relational database.

This command uses a *data manager* that coordinates how data is extracted from a specified *data source*. In this way, the load 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 load command when used to load data from TAB files. Next, we discuss the command syntax that is used to load data from TAB, CSV and XML files, which are loaded in a similar manner. In Section 6.5.4 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.

6.5.1 Simple Load Examples

The simplest illustration of the load 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 load command loads the parameter data:

```
load 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 load command loads both the parameter data as well as the index set A:

```
load 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:

```
load 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 `load` syntax will load this data into set `C`:

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

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

6.5.2 Load Syntax Options

The syntax of the `load` 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 a database
`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
`table`	The database table that is requested

The `format` option is the only option that is required for all data managers. This option specifies how a relational table is interpreted to represent set and parameter data. A complete set of examples for this option is provided in Section 6.5.3. 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 6.5.4.

The second part of the `load` 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 load 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:

```
load 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:

```
load 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 load command also allows for the specification of the index set:

```
load 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:

```
load 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 load 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 loaded using columns B, C and D:

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

When the using option is omitted the data manager is inferred from the filename suffix. However, the filename suffix does not always reflect the format of the data it contains. For example, consider the relational table in the file ABCD.txt:

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

We can specify the using option to load from this file into parameter D and set Z:

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

NOTE: The data managers supported by Pyomo can be listed with the `pyomo help` subcommand:

```
pyomo help --data-managers
```

The following data managers are supported in Pyomo 5.1:

```
Pyomo Data Managers
-------------------
  csv
      CSV file interface
  dat
      Pyomo data command file interface
  json
      JSON file interface
  pymysql
      pymysql database interface
  pyodbc
      pyodbc database interface
  pypyodbc
      pypyodbc database interface
  sqlite3
      sqlite3 database interface
  tab
      TAB file interface
  xls
      Excel XLS file interface
  xlsb
      Excel XLSB file interface
  xlsm
      Excel XLSM file interface
  xlsx
      Excel XLSX file interface
  xml
      XML file interface
  yaml
      YAML file interface
```

6.5.3 Interpreting Tabular Data

By default, a 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:

array	The table is a matrix representation of a two dimensional parameter.
param	The data is a simple 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 two dimensional 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 table with a single value can be interpreted as a simple parameter using the param format value. Suppose that Z.tab contains the following "table":

```
1.1
```

The following load command then loads this value into parameter p:

```
load 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 load command loads data into set B:

```
load 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 load command loads this value into parameter U with a 2-dimensional index using the array format value.

```
load 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:

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

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

6.5.4 Loading from Spreadsheets and Relational Databases

Many of the options for the `load` command are specific to spreadsheets and relational databases. The `range` option is used to specify the range of cells that are loaded from a spreadsheet. The range of cells represents a table in which 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 loads this data to initialize parameter `D` and index `Z`:

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

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

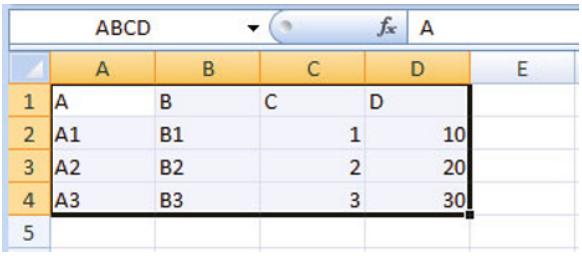


Fig. 6.1: A snapshot of the `ABCD.xls` spreadsheet, which defines a relational table in the `ABCD` range.

When loading 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.

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 a database. For example, the `pyodbc` interface can be used to connect to Excel spreadsheets. The following command loads data from the Excel spreadsheet `ABCD.xls` using the `pyodbc` interface. The command loads this data to initialize parameter `D` and index `Z`:

```
load 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:

```
load "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 load command can be tailored to the details of the database that a user is working with.

The previous examples specified the table option, which declares the name of a relational table in a database. Many databases support the Structured Query Language (SQL), which can be used to dynamically compose a relational table from other tables in a database. The classic diet problem will be used to illustrate the use of SQL queries to initialize a Pyomo model. In this problem, 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 following is a Pyomo model for this problem:

```
# diet1.py
from pyomo.environ import *

infinity = float('inf')
MAX_FOOD_SUPPLY = 20.0 # There is a finite food supply

model = AbstractModel()

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

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

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)
```

Suppose that the file `diet1.sqlite` be a SQLite database file that contains 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 database. The file `diet.sqlite.dat` specifies a `load` command that uses that `sqlite3` data manager and embeds a SQL query to retrieve the data:

```
# File diet.sqlite.dat

load "diet.sqlite"
     using=sqlite3
     query="SELECT FOOD,cost,f_min,f_max FROM Food"
     : 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.

6.6 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 `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.

6.7 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. Consider the following data command file:

```
set C := 1 2 3 ;

namespace ns1
{
    set C := 4 5 6 ;
}

namespace ns2
{
    set C := 7 8 9 ;
}
```

This data file defines two namespaces: `ns1` and `ns2` that initialize a set `C`. By default, data commands contained within a namespace are ignored during model construction; when no namespaces are specified, the set `C` has values `1,2,3`. When namespace `ns1` is specified, then set `C` values are overridden with the set `4,5,6`. See Section 5.5.2 for an example of how namespaces are selected with the `pyomo` command.

6.8 Discussion

Pyomo data commands are processed with Pyomo's `DataPortal` class. Thus, there is a direct correspondence between the `set`, `param` and `table` commands and the data formats supported by the `load` method in the `DataPortal` class.

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 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 directly supported, but the `load` 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 10).

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). 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.

Part II
Advanced Features and Extensions

Chapter 7
Nonlinear Programming with Pyomo

Abstract This chapter describes the nonlinear programming capabilities of Pyomo. It presents 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.

7.1 Introduction

It is not possible to adequately represent many applications without modeling nonlinear relationships. Fortunately, Pyomo has the ability to represent general nonlinear programming (NLP) problems in a straightforward manner. However, the solution of this class of problems presents several challenges that do not exist for linear 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. Additionally, 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 7.2, we describe the nonlinear expressions supported in Pyomo and then illustrate how to build a basic nonlinear problem formulation within Pyomo. In Section 7.3, we discuss the solver interface that allows for 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.

7.2 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.

7.2.1 Nonlinear Expressions

Formulating nonlinear optimization problems in Pyomo is no different from formulating linear or mixed-integer problems. All the Pyomo modeling components that we have described throughout the book are used in the same way (e.g., `Objective`, `Constraint`) except that they may include nonlinear expressions.

Table 7.1 lists the operators that are currently supported to formulate expressions, 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 7.2. These are Pyomo implementations of nonlinear functions that can be used in Pyomo expressions. However, Python nonlinear functions (e.g., from the `math` package) cannot be used to build nonlinear expressions in Pyomo!

NOTE: If you import Python functions and attempt to build nonlinear expressions, this can cause problems that are difficult to debug. For example, the following code will certainly cause problems since the functions from the `math` package will be used instead of the functions from the `pyomo` package.

```
from pyomo.environ import *
from math import *
```

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 division is equivalent to `expr = expr / model.x`.

[3] The example given for in-place exponentiation is equivalent to `expr = expr ** model.x`.

Table 7.1: 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 7.2: Functions supported by Pyomo for the definition of nonlinear expressions.

7.2.2 The Rosenbrock Problem

In this section we present a short example to illustrate the formulation and solution of a nonlinear Pyomo model. We consider the unconstrained minimization of the two-variable Rosenbrock function, which is a classic problem that is frequently used as an example for discussion of unconstrained nonlinear optimization algorithms (see, for example, [65]). This 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 (See Figure 7.1).

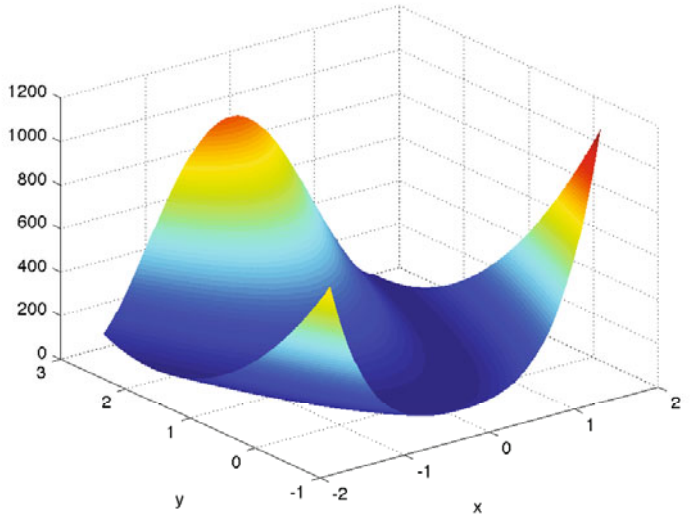


Fig. 7.1: Contours of the Rosenbrock function $f(x,y)=(1-x)^2+100\left(y-x^2\right)^2$. The minimum is in the bottom of a banana shaped valley at the point x=1, y=1.

Consider the following Pyomo model for this problem:

```
# rosenbrock.py
# A Pyomo model for the Rosenbrock problem
from pyomo.environ import *

model = AbstractModel()
model.x = Var(initialize=1.5)
model.y = Var(initialize=1.5)

def rosenbrock(model):
    return (1.0-model.x)**2 \
        + 100.0*(model.y - model.x**2)**2
model.obj = Objective(rule=rosenbrock, sense=minimize)
```

This example illustrates that defining a nonlinear model is really no different from defining a linear model. The model creates two variables `x` and `y` and initializes each of them to a value of 1.5. Notice that there is no need to provide any indication that the variables will later appear in a nonlinear expression; this will be deduced by Pyomo before solving the problem. The construction rule for the objective function simply returns a nonlinear expression.

The following `pyomo` command solves this optimization problem using the IPOPT solver:

```
pyomo solve --solver=ipopt --summary rosenbrock.py
```

This produces output similar to the following:

```
[    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: None
      Status: optimal
      Function Value: 7.013645951336496e-25
    Solver results file: results.yml

==========================================================
Solution Summary
==========================================================

Model unknown

  Variables:
    x : Size=1, Index=None
        Key  : Lower : Value              : Upper : Fixed :
    Stale : Domain
        None :  None : 1.0000000000008233 :  None : False :
    False :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value              : Upper : Fixed :
    Stale : Domain
        None :  None : 1.0000000000016314 :  None : False :
    False :  Reals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 7.013645951336496e-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 has only a single nonlinear objective and two scalar variables, all the modeling components that were discussed earlier can also be used.

NOTE: The `pyomo` command may result in output similar to the following:

```
[    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.

7.3 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. Thus, an appropriate nonlinear solver must be installed before Pyomo can be used to analyze a nonlinear programming model.

7.3.1 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 leverages the solver interface provided by the AMPL Solver Library (ASL) for solving NLP problems [33]. 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 immediately available for use with Pyomo without any need to develop another interface. Additionally, 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) [34]. 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 solution

values to a `.sol` file, which is read by Pyomo to obtain the solution. Further details concerning these file formats are provided in Gay et al. [29, 33].

NOTE: *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 the examples in this chapter, we have made use of IPOPT[48], 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/`.

7.3.2 Additional 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 formulation and solution of nonlinear programming problems.

Variable Initialization

Solvers for nonlinear programming problems often require the initialization of problem 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, large problems cannot be solved by state-of-the-art solvers. Consequently, one is often forced to employ a solver that only provides a guarantee of local optimality, for which 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.

Undefined Evaluations

Several nonlinear functions 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 and initial values on the variables.

It is also important to note that many nonlinear solvers use 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 to guarantee convergence. 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). Nocedal and Wright [65] 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.

7.4 Nonlinear Programming Examples

In this section we present several additional nonlinear examples that illustrate the capabilities of Pyomo.

7.4.1 Variable Initialization for a Multimodal Function

The following example illustrates the importance of effective variable initialization. Consider the minimization of the following multimodal function:

$$f(x) = (2 - \cos \pi x - \cos \pi y)\, x^2 y^2,$$

which has multiple local minima. The following Pyomo model for this problem initializes the variables at x=y=0.25.

```
# multimodal_init1.py
from pyomo.environ 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)
```

We can solve this problem using the following `pyomo` command:

```
pyomo solve --solver=ipopt --summary multimodal_init1.py
```

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))
```

then IPOPT finds a different local solution at x=y=2.0.

NOTE: Recall that we recommended not using the Python built-in math functions since they can override the Pyomo functions suited for Pyomo expressions. In particular,

```
from math import *
```

is very problematic since it will blindly override the internal Pyomo functions. In the example above, we *explicitly* import `pi` from the math package, but not any other functions. We could also have imported the Python math module with a name to make sure we did not override any of the internal Pyomo math functions:

```
import math as mt
```

7.4.2 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, harvesting too many bucks within a population can limit 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 consider a model adapted from Bailey [5] that describes the dynamics of the deer population. The deer population in a given area can be divided into three sub-populations: bucks, does, and fawns. Additionally, 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 [5], resulting in the following set of difference equations,

$$f_{y+1} = p_1 br_y \left(\frac{p_2}{10} f_y + p_3 d_y\right) + h_y^f \tag{7.1}$$

$$d_{y+1} = p_4 d_y + \frac{p_5}{2} f_y - h_y^d \tag{7.2}$$

$$b_{y+1} = p_6 b_y + \frac{p_5}{2} f_y - h_y^b \tag{7.3}$$

$$br_y = 1.1 + 0.8 \frac{p_s - c_y}{p_s} \tag{7.4}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y \tag{7.5}$$

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 7.3. 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 7.3: Parameter values used by 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 the same 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,

$$\max \; w_b h_y^b + w_f h_y^f + w_d h_y^d \tag{7.6}$$

$$f_y = p_1 br_y \left(\frac{p_2}{10} f_y + p_3 d_y \right) + h_y^f \tag{7.7}$$

$$d_y = p_4 d_y + \frac{p_5}{2} f_y - h_y^d \tag{7.8}$$

$$b_y = p_6 b_y + \frac{p_5}{2} f_y - h_y^b \tag{7.9}$$

$$br_y = 1.1 + 0.8 \frac{p_s - c_y}{p_s} \tag{7.10}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y \tag{7.11}$$

$$c_y \leq p_s \tag{7.12}$$

$$b_y \geq \frac{1}{5} (0.4 f_y + d_y) \tag{7.13}$$

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 7.3, it is assumed that the value of a buck tag is 10 times the value of a doe or fawn tag. Equation (7.12) ensures that the amount

of consumed food cannot be more than the available supply, thereby restricting the overall size of the population. Equation (7.13) ensures that the number of bucks is large enough for effective, sustainable breeding.

The following abstract Pyomo model represents the optimal deer harvesting problem:

```
# DeerProblem.py
from pyomo.environ import *

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)
```

The following data file represents the parameters in Table 7.3:

```
# DeerProblem.dat
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;
```

This problem can be optimized with the following command:

```
pyomo solve --solver=ipopt --summary DeerProblem.py \
    DeerProblem.dat
```

This produces the following output:

```
[ 0.00] Setting up Pyomo environment
[ 0.00] Applying Pyomo preprocessing actions
[ 0.00] Creating model
[ 0.02] Applying solver
[ 0.04] Processing results
   Number of solutions: 1
   Solution Information
     Gap: None
     Status: optimal
     Function Value: 659.224784497
   Solver results file: results.json

==========================================================
Solution Summary
==========================================================

Model unknown

  Variables:
    f : Size=1, Index=None
```

```
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 189.605592667 : None : False : False : PositiveReals
  d : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 196.006401042 : None : False : False : PositiveReals
  b : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 54.3697276124 : None : False : False : PositiveReals
  hf : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 0.0 : None : False : False : PositiveReals
  hd : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 37.8450171569 : None : False : False : PositiveReals
  hb : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 62.137976734 : None : False : False : PositiveReals
  br : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 1.09999999201 : None : False : False : PositiveReals
  c : Size=1, Index=None
      Key : Lower : Value : Upper : Fixed : Stale : Domain
      None : 0 : 700000.00699 : None : False : False : PositiveReals

Objectives:
  obj : Size=1, Index=None, Active=True
      Key : Active : Value
      None : True : 659.224784497

Constraints:
  f_bal : Size=1
      Key : Lower : Body : Upper
      None : 0.0 : 8.98742769095e-09 : 0.0
  d_bal : Size=1
      Key : Lower : Body : Upper
      None : 0.0 : 1.42108547152e-14 : 0.0
  b_bal : Size=1
      Key : Lower : Body : Upper
      None : 0.0 : 7.1054273576e-15 : 0.0
  food_cons : Size=1
      Key : Lower : Body : Upper
      None : 0.0 : 2.91038304567e-11 : 0.0
  supply : Size=1
      Key : Lower : Body : Upper
      None : None : 700000.00699 : 700000
  birth : Size=1
      Key : Lower : Body : Upper
      None : 0.0 : 0.0 : 0.0
  minbuck : Size=1
      Key : Lower : Body : Upper
      None : None : 9.34787180995e-09 : 0.0

[ 0.04] Applying Pyomo postprocessing actions
[ 0.04] Pyomo Finished
```

The solution summary shows 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 is restricting 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 14 gives more discussion of this functionality.

7.4.3 Estimation of Infectious Disease Models

Effective widespread vaccination programs 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.

NOTE: Typically, we refer to *parameters* as fixed data in our optimization problem. However, in this example, the parameters in our infectious disease model are not yet known, and we want to estimate them from existing data. Because of this, our model parameters β and α will become Pyomo variables in the model (since they are to be estimated with the optimization).

In this example, we use least-squares to estimate the parameters from simulated data. Let SI be 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. The full problem formulation is given by,

$$
\begin{aligned}
&\min \sum_{i \in SI} \left(\varepsilon_i^I\right)^2 \\
&I_i = \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N} \ \forall \ i \in SI \setminus \{1\} \\
&S_i = S_{i-1} - I_i \ \forall \ i \in SI \setminus \{1\} \\
&C_i = I_i + \varepsilon_i^I \\
&0 \leq I_i, \ Si \leq N \\
&0.5 \leq \beta \leq 70 \\
&0.5 \leq \alpha \leq 1.5
\end{aligned}
$$

The following listing shows an abstract model for this nonlinear least-squares estimation problem:

```
# disease_estimation.py
from pyomo.environ 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)

def pyomo_postprocess(options=None, instance=None, \
                results=None):
   print(' ***')
   print(' *** Optimal beta Value: %.2f' % \
       value(instance.beta))
   print(' *** Optimal alpha Value: %.2f' % \
       value(instance.alpha))
   print(' ***')
```

The Pyomo data file that contains the data for an instance of this model is given by:

```
# disease_estimation.dat

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 default 0.0 :=
  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
;
```

The following command can be used to optimize this model:

```
pyomo solve --solver=ipopt --logging=quiet \
    disease_estimation.py disease_estimation.dat
```

Note that we have included a post-processing callback that produces output similar to the following:

```
***
*** Optimal beta Value: 1.99
*** Optimal alpha Value: 1.00
***
```

We generated the data with β=2 and α=1, so these results look quite reasonable.

7.4.4 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 7.2, 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, so 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.

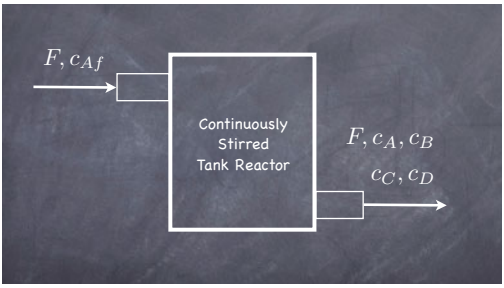


Fig. 7.2: Continuously stirred tank reactor system producing desired product B, and undesired products C, and D from A.

Consider the following reactor problem that was adapted from Bequette [9]. 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,

$$0 = \frac{F}{V}c_{Af} - \frac{F}{V}c_A - k_1 c_A - 2k_3 c_A^2$$
$$0 = -\frac{F}{V}c_B + k_1 c_A - k_2 c_B$$
$$0 = -\frac{F}{V}c_C + k_2 c_B$$
$$0 = -\frac{F}{V}c_D + k_3 c_A^2$$

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.

The following is a concrete model for the reactor design problem:

```
import pyomo.environ
from pyomo.core 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))
```

This can be solved with the following command:

```
pyomo solve --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


******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
       For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

This is Ipopt version 3.12.3, running with linear solver ma27.

Number of nonzeros in equality constraint Jacobian...: 11
Number of nonzeros in inequality constraint Jacobian.: 0
Number of nonzeros in Lagrangian Hessian.............: 5

Total number of variables............................: 5
                     variables with only lower bounds: 5
                variables with lower and upper bounds: 0
                     variables with only upper bounds: 0
Total number of equality constraints.................: 4
Total number of inequality constraints...............: 0
      inequality constraints with only lower bounds: 0
  inequality constraints with lower and upper bounds: 0
      inequality constraints with only upper bounds: 0

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
   0 -2.0000000e+03 7.50e+03 6.25e-01 -1.0 0.00e+00 - 0.00e+00 0.00e+00 0
   1 -1.0801475e+03 5.54e+02 2.46e+00 -1.0 1.39e+03 - 5.54e-01 1.00e+00h 1
   2 -1.0763574e+03 8.86e+01 1.87e+02 -1.0 3.66e+02 - 9.31e-01 1.00e+00h 1
   3 -1.0727252e+03 1.07e+01 5.26e+00 -1.0 9.35e+01 - 9.71e-01 1.00e+00h 1
   4 -1.0726714e+03 4.01e+00 1.22e-01 -1.0 6.03e+01 - 9.68e-01 1.00e+00h 1
   5 -1.0724371e+03 3.44e-04 2.93e-05 -2.5 4.19e-01 - 1.00e+00 1.00e+00h 1
   6 -1.0724372e+03 1.63e-08 3.93e-09 -3.8 3.85e-03 - 1.00e+00 1.00e+00h 1
   7 -1.0724372e+03 5.46e-11 2.70e-11 -5.7 2.24e-04 - 1.00e+00 1.00e+00h 1
   8 -1.0724372e+03 4.55e-13 3.56e-14 -8.6 2.78e-06 - 1.00e+00 1.00e+00h 1

Number of Iterations....: 8

                                   (scaled)  (unscaled)
Objective...............: -1.0724372001086319e+03 -1.0724372001086319e+03
Dual infeasibility......: 3.5606115145031445e-14 3.5606115145031445e-14
Constraint violation....: 4.5474735088646414e-14 4.5474735088646412e-13
Complementarity.........: 2.5059065225790179e-09 2.5059065225790179e-09
Overall NLP error.......: 2.5059065225790179e-09 2.5059065225790179e-09


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) = 0.002
Total CPU secs in NLP function evaluations = 0.000

EXIT: Optimal Solution Found.

Ipopt 3.12.3: Optimal Solution Found
[ 0.07] Processing results
   Number of solutions: 1
   Solution Information
     Gap: None
     Status: optimal
     Function Value: 1072.43720011
   Solver results file: results.yml

============================================================
Solution Summary
============================================================

Model unknown
```

```
Variables:
  sv : Size=1, Index=None
     Key : Lower : Value : Upper : Fixed : Stale : Domain
     None : 0 : 1.34381176107 : None : False : False : PositiveReals
  ca : Size=1, Index=None
     Key : Lower : Value : Upper : Fixed : Stale : Domain
     None : 0 : 3874.25886723 : None : False : False : PositiveReals
  cb : Size=1, Index=None
     Key : Lower : Value : Upper : Fixed : Stale : Domain
     None : 0 : 1072.43720011 : None : False : False : PositiveReals
  cc : Size=1, Index=None
     Key : Lower : Value : Upper : Fixed : Stale : Domain
     None : 0 : 1330.09353341 : None : False : False : PositiveReals
  cd : Size=1, Index=None
     Key : Lower : Value : Upper : Fixed : Stale : Domain
     None : 0 : 1861.60519963 : None : False : False : PositiveReals

Objectives:
  obj : Size=1, Index=None, Active=True
     Key : Active : Value
     None : True : 1072.43720011

Constraints:
  ca_bal : Size=1
     Key : Lower : Body : Upper
     None : 0.0 : 0.0 : 0.0
  cb_bal : Size=1
     Key : Lower : Body : Upper
     None : 0.0 : -2.27373675443e-13 : 0.0
  cc_bal : Size=1
     Key : Lower : Body : Upper
     None : 0.0 : 0.0 : 0.0
  cd_bal : Size=1
     Key : Lower : Body : Upper
     None : 0.0 : -4.54747350886e-13 : 0.0

[ 0.07] Applying Pyomo postprocessing actions
[ 0.07] 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 following Python script:

```
from pyomo.environ 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))

# run the sequence of square problems
solver = SolverFactory('ipopt')
model.sv.fixed = True
sv_values = [1.0 + v * 0.05 for v in range(1, 20)]
print(" %s %s" % (str('sv'.rjust(10)), str('cb'.rjust(10))))
for sv_value in sv_values:
   model.sv = sv_value
   solver.solve(model)
   print(" %s %s" %(str(model.sv.value).rjust(10),\
      str(model.cb.value).rjust(15)))
```

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, producing 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 14 for more scripting examples and further description of these capabilities.

Chapter 8
Structured Modeling with Blocks

Abstract This chapter documents how to express hierarchically-structured models using Pyomo's `Block` component. Many models contain significant hierarchical structure; that is, they are composed of repeated groups of conceptually related modeling components. Pyomo allows the modeler to define fundamental building blocks, and then construct the overall problem by connecting these building blocks together in an object-oriented manner. In this chapter, we describe the fundamental `Block` component along with common examples of its use, including repeated components and managing model scope.

8.1 Introduction

Optimization solvers typically rely on getting a model in a standardized form. For example, linear solvers accept models built on a standard form similar to:

$$\begin{aligned} &\min\ c^T x \\ &\text{s.t.}\ \ Ax \leq b\ \ . \\ &\qquad x \geq 0 \end{aligned}$$

Here, the variables are lumped together into a single vector x, and constraints are represented in simplified matrix form. While this form is convenient for algorithms that directly manipulate these matrices, it is not an easy form for a modeler to generate, manipulate, or debug. Algebraic Modeling Languages (AMLs) directly address this challenge by allowing modelers to provide distinguishing names to modeling components (e.g., variables or constraints) and to define the model over index sets. Since models are often composed of repeated mathematical expressions, this allows the expression of large models with relatively few lines of code, which are also easier to document, understand, modify, and debug.

As models become larger and more complex, however, we often want to carry this concept further and group variables and constraints that are conceptually re-

lated. That is, not that the constraints are connected by a common expression generator, but rather that the variables and constraints describe a certain (often physical) concept. For example, the group of variables and constraints could represent the operating behavior of an electric generator (ramp-up limits, ramp-down limits, cost curves) or chemical process equipment like a distillation column (the mass, equilibrium, and energy balance equations). Other examples include multi-period optimization problems where the same fundamental model is repeated over many time periods, or stochastic programming problems where the same basic model is repeated over different scenarios with different parameters. In Pyomo, we use the `Block` component to support object-oriented construction of hierarchical models like those described above.

The `Block` component is a container for organizing groups of variables and constraints, and it can contain any number of named Pyomo components in exactly the same way that models do. In fact, `ConcreteModel` and `AbstractModel` inherit from the `Block` component. Additionally, `Block` components can themselves be added to a model, allowing for hierarchical model construction based on the fundamental building blocks in an object-oriented manner.

This concept is illustrated in Figure 8.1, which provides an overview of the capabilities of blocks for an electrical grid model. Individual block can be used to define the necessary variables and equations to describe generator, bus, and transmission line elements. Then, these can be put together to form the entire (single time period)

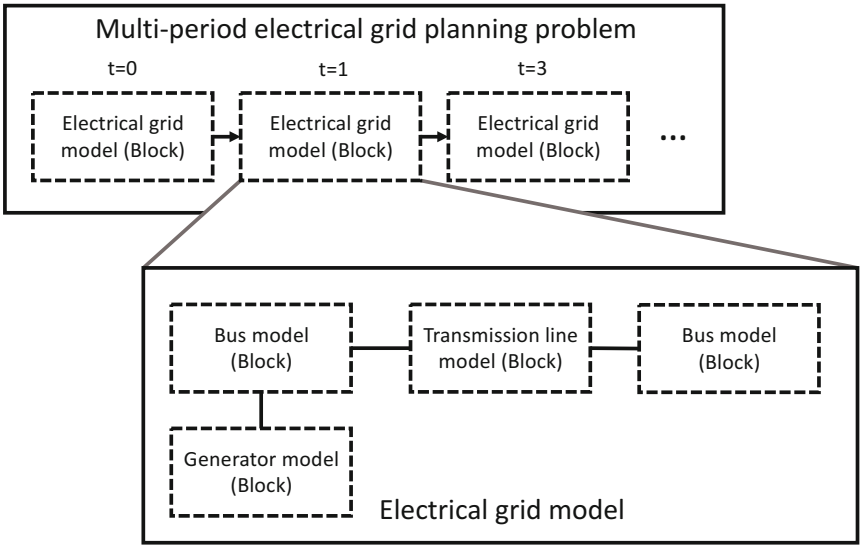


Fig. 8.1: The electrical grid model can be composed from individual blocks representing the generators, buses, and transmission lines. Furthermore, a multi-period model can be constructed in a hierarchical manner treating the electrical grid model as its fundamental building blocks.

electrical grid model. This model could be solved on its own. However, it can also be used as a building block for a higher-level model, like the multi-period planning problem illustrated in this figure. In this manner, Pyomo can represent very complex models out of smaller, less complex pieces.

8.2 Block structures

The Pyomo `Block` component can be treated in much the same way as a model: components are added directly to the block as attributes. Since `Block` components may contain any other Pyomo modeling components, including other blocks, it is possible to construct arbitrarily nested hierarchical structures.

When discussing the block hierarchy, we adopt terminology from tree structures and refer to the block one level up the hierarchy (toward the top-level Model) as the *parent*, and all components contained within the block as *children*. The *root* of the block hierarchy is always the current model. The following code snippet shows some basic capabilities of blocks.

```
model = ConcreteModel()
model.x = Var()
model.P = Param(initialize=5)
model.S = RangeSet(model.P)
model.b = Block()
model.b.I = RangeSet(model.P)
model.b.x = Var(model.b.I)
model.b.y = Var(model.S)
model.b.b = Block()
model.b.b.x = Var()
```

Here, `model` contains a variable (`model.x`), a parameter (`model.P`), and a set (`model.S`). It also contains a `Block` given by `model.b`. This block itself contains a set, two variables, and yet another block. You will notice that components in a block can reference components in other blocks or in the parent model. The range set `model.b.I` is defined using a parameter from the parent model, the variable `model.b.x` uses a set from it's same block, and the variable `model.b.y` references a set from the parent block. Components and expressions can reference components from anywhere within the hierarchy.

NOTE: Within one block, Pyomo supports references to components from any other block. However, it is generally good object-oriented practice to only reference components from the current or lower levels in the hierarchy. This promotes resusability of your blocks within other models without strong assumptions about the structure of the owning or parent blocks.

Note that, in the above code snippet, each block defines its own *component namespace*; that is, while component names must be unique within a single block,

they do not need to be globally unique. This allows blocks to be constructed safely without concerns about definitions in one block colliding or interfering with definitions in other blocks. This also leads to all components having two forms of a name: the local name (which may be repeated elsewhere in the model) and a globally-unique *fully qualified* name that includes the names of the parent block(s) separated by periods:

```
print(model.x.local_name)      # x
print(model.x.name)            # x
print(model.b.x.local_name)    # x
print(model.b.x.name)          # b.x
print(model.b.b.x.local_name) # x
print(model.b.b.x.name)        # b.b.x
```

`Block` components can also be created and populated, and then later added to a model. The code below shows creation of a block that is later added to a model. It also illustrates how a parent model can make use of sets and parameters contained in a child block.

```
new_b = Block()
new_b.x = Var()
new_b.P = Param(initialize=5)
new_b.I = RangeSet(10)

model = ConcreteModel()
model.b = new_b
model.x = Var(model.b.I)
```

NOTE: In this example, the `Block` object `new_b` is not initialized when it is declared. In this manner, it is abstract until it is added to the `ConcreteModel` object. At that point, it is immediately initialized. Similarly, when a `Block` object is added to an `AbstractModel`, it is initialized when the entire model object is initialized.

8.3 Blocks as Indexed Components

As with other Pyomo components, `Block` components may also be indexed and initialized using a construction rule. However, block construction rules follow a slightly different convention: the first argument to to a block rule is the block to be populated rather than the owning block. Within a rule, one can either directly populate the block by assigning components to it, or create a new block and return it from the rule.

The following example illustrates the use of construction rules for blocks:

```
model = ConcreteModel()
model.P = Param(initialize=3)
model.T = RangeSet(model.P)

def xyb_rule(b, t):
   b.x = Var()
   b.I = RangeSet(t)
   b.y = Var(b.I)
   b.c = Constraint(expr = b.x == 1.0 - sum(b.y[i] for i \
       in b.I))
model.xyb = Block(model.T, rule=xyb_rule)
```

Here, we have defined an indexed `Block` component that will contain one block for each element of the set `model.T`. In the construction rule, we create two variables and a set. These construction rules are just as flexible as those for other components. The set `b.I` created in this example is different for each block, and consequently the variable `b.y` is also a different length for each block. This illustrates another feature of blocks. Often, different blocks contain exactly the same model structure, just with different data. However, it is also possible to construct blocks that have a different structure based on data available in the construction rule.

We can extend this example to print the constraint body `c` for each of the blocks:

```
for t in model.T:
   print(model.xyb[t].c.body)
```

The constraints expand appropriately and they contain fully qualified names of variables in each of the subblocks:

```
-1.0 + xyb[1].x + xyb[1].y[1]
-1.0 + xyb[2].x + xyb[2].y[1] + xyb[2].y[2]
-1.0 + xyb[3].x + xyb[3].y[1] + xyb[3].y[2] + xyb[3].y[3]
```

8.4 Construction Rules within Blocks

Like model objects, blocks can contain other modeling components, including `Set` and `Param` objects. Additionally, blocks can be initialized with modeling components which themselves are constructed by rules. However, doing this exposes a subtelty of Pyomo component construction rules.

Up to this point we have frequently referred to the first argument of a component rule as the "model", but this is not completely correct. The first argument to component rules is actually the *owning block* of the component being constructed. For "flat" models (models without any sub-blocks), the owning block is indeed the model, but this will not be the case for hierarchically-structured models. If needed, the model object can be obtained from the owning block using the aforementioned `model()` method.

For example, consider the following declaration of the xyb block from the previous example:

```
def xyb_rule(b, t):
   b.x = Var()
   b.I = RangeSet(t)
   b.y = Var(b.I, initialize=1.0)
   def _b_c_rule(_b):
      return _b.x == 1.0 - sum(_b.y[i] for i in _b.I)
   b.c = Constraint(rule=_b_c_rule)
model.xyb = Block(model.T, rule=xyb_rule)
```

In this example, the xyb block includes a constraint that is defined with the rule _b_c_rule. The owning block _b passed to this rule is the same as b. However, Pyomo the _b variable is defined locally. This allows the rule to be used even if the owning block is constructed in a different manner.

Since the owning block (or model) is NOT passed into a block construction rule, the modeler may need another mechanism to access components on the parent or other blocks in the hierarchy. The component methods parent_block and model facilitate moving up the block hierarchy. The parent_block() of any component or component data object is the block that the component is attached to. The model() method on any component or component data object returns the block object at the root of the tree.

8.5 Extracting values from hierarchical models

While blocks support a convenient mechanism for expressing composite concepts (e.g., a time-period, a scenario), this results in some data becoming more spread out across your model. However, we can access the components by explicitly iterating over blocks and their associated variables:

```
for t in model.xyb:
  for i in model.xyb[t].y:
    print("%s %f" % (model.xyb[t].y[i], \
        value(model.xyb[t].y[i])))
```

Additionally, Pyomo's slice notation can be used to dynamically extract a subset of the blocks or variable values:

```
for y in model.xyb[:].y[:]:
   print("%s %f" % (y, value(y)))
```

8.6 Blocks Example: Optimal Multi-Period Lot-Sizing

We now demonstrate a complete model based on blocks using a well-known multi-period optimization problem for optimal lot-sizing [43]. Our goal in the lot-sizing

problem is to determine the optimal production X_t in each time period $t \in T$ given known demands d_t. We let y_t be a binary variable indicating whether or not there is any production in time period t, and assume that there is a fixed cost c_t if we decide to produce in time period t. The inventory I_t at the end of each time period is a function of the previous inventory, production, and sales,

$$I_t = I_{t-1} + X_t - d_t.$$

If we allow the inventory to be negative (meaning we did not meet demands and we have a backlog of orders), we can represent the inventory as $I_t = I_t^+ - I_t^-$ where we restrict both I_t^+ and I_t^- to be non-negative. Here, I_t^+ represents inventory that we are holding, and I_t^- represents a backlog of orders. We can assign an inventory holding cost and a shortage cost (cost of keeping a backlog) as h_t^+ and h_t^- respectively.

With this description, the optimization problem can be formulated as,

$$\min \sum_{t \in T} c_t y_t + h_t^+ I_t^+ + h_t^- I_t^- \tag{LS.1}$$

$$\text{s.t.} \quad I_t = I_{t-1} + X_t - d_t \qquad \forall t \in T \tag{LS.2}$$

$$I_t = I_t^+ - I_t^- \qquad \forall t \in T \tag{LS.3}$$

$$X_t \leq P y_t \qquad \forall t \in T \tag{LS.4}$$

$$X_t,\, I_t^+,\, I_t^- \geq 0 \qquad \forall t \in T \tag{LS.5}$$

$$y_t \in \{0, 1\} \qquad \forall t \in T \tag{LS.6}$$

where Equation (LS.4) is a constraint that only allows production in time period t if the indicator variable $y_t = 1$. The data for our problem is provided in Table 8.1.

Table 8.1: Data for Lot-Sizing Problem

Parameter	Description	Value
c	fixed cost of production	4.6
I_0^+	initial value of positive inventory	5.0
I_0^-	initial value of backlogged orders	0.0
h^+	cost (per unit) of holding inventory	0.7
h^-	shortage cost (per unit)	1.2
P	maximum production amount (big-M value)	5
d	demand	$[5, 7, 6.2, 3.1, 1.7]$

8.6.1 A Formulation Without Blocks

We can formulate the lot-sizing problem *without* blocks using the Pyomo code shown below.

```
from pyomo.environ import *

model = ConcreteModel()
model.T = RangeSet(5) # time periods

i0 = 5.0 # initial inventory
c = 4.6 # setup cost
h_pos = 0.7 # inventory holding cost
h_neg = 1.2 # shortage cost
P = 5.0 # maximum production amount

# demand during period t
d = {1: 5.0, 2:7.0, 3:6.2, 4:3.1, 5:1.7}

# define the variables
model.y = Var(model.T, domain=Binary)
model.x = Var(model.T, domain=NonNegativeReals)
model.i = Var(model.T)
model.i_pos = Var(model.T, domain=NonNegativeReals)
model.i_neg = Var(model.T, domain=NonNegativeReals)

# define the inventory relationships
def inventory_rule(m, t):
   if t == m.T.first():
      return m.i[t] == i0 + m.x[t] - d[t]
   return m.i[t] == m.i[t-1] + m.x[t] - d[t]
model.inventory = Constraint(model.T, rule=inventory_rule)

def pos_neg_rule(m, t):
   return m.i[t] == m.i_pos[t] - m.i_neg[t]
model.pos_neg = Constraint(model.T, rule=pos_neg_rule)

# create the big-M constraint for the production indicator variable
def prod_indicator_rule(m,t):
   return m.x[t] <= P*m.y[t]
model.prod_indicator = Constraint(model.T, rule=prod_indicator_rule)

# define the cost function
def obj_rule(m):
   return sum(c*m.y[t] + h_pos*m.i_pos[t] + h_neg*m.i_neg[t] for t in m.T)
model.obj = Objective(rule=obj_rule)

# solve the problem
solver = SolverFactory('glpk')
solver.solve(model)

# print the results
for t in model.T:
   print('Period: {0}, Prod. Amount: {1}'.format(t, value(model.x[t])))
```

This example uses standard Pyomo syntax as discussed in early chapters of the book. If we were considering the lot-sizing problem over a single time period only, our variable declarations would have looked like,

```
# define the variables
model.y = Var(domain=Binary)
model.x = Var(domain=NonNegativeReals)
model.i = Var()
model.i_pos = Var(domain=NonNegativeReals)
model.i_neg = Var(domain=NonNegativeReals)
```

In the multi-period case, we have the same fundamental variables and constraints defined over each time period. Here, the variable declarations looked like,

```
# define the variables
model.y = Var(model.T, domain=Binary)
model.x = Var(model.T, domain=NonNegativeReals)
model.i = Var(model.T)
model.i_pos = Var(model.T, domain=NonNegativeReals)
model.i_neg = Var(model.T, domain=NonNegativeReals)
```

If we were considering a multi-period and multi-scenario problem (e.g., a stochastic programming formulation for lot-sizing under uncertainty), the variable declarations would have looked like,

```
# define the variables
model.y = Var(model.T, model.S, domain=Binary)
model.x = Var(model.T, model.S, domain=NonNegativeReals)
model.i = Var(model.T, model.S,)
model.i_pos = Var(model.T, model.S, domain=NonNegativeReals)
model.i_neg = Var(model.T, model.S, domain=NonNegativeReals)
```

In each of these examples, when we add new complexity, or an additional *layer* onto the model, we add a new index to the variables and constraints. This approach is very common in the field of operations research. Unfortunately, it requires that we completely redefine the model with each new layer, and it does not readily support construction of hierarchical models with reusable code. Blocks provide another approach that easily supports model reuse in an object-oriented fashion.

8.6.2 A Formulation With Blocks

We now show how blocks can be used to write this problem. Most of the constraints in the multi-period lot-sizing problem are defined over $t \in T$, and they can be logically grouped together by time. Pyomo allows us to define blocks, each with the variables and constraints for a single time period only, and then link them together to form the overall model.

Considering the lot-sizing problem again, we can define the variables and constraints within a rule that provides a block for a single period of the lot-sizing problem:

```
# create a block for a single time period
def lotsizing_block_rule(b, t):
    # define the variables
    b.y = Var(domain=Binary)
    b.x = Var(domain=NonNegativeReals)
    b.i = Var()
    b.i0 = Var()
    b.i_pos = Var(domain=NonNegativeReals)
    b.i_neg = Var(domain=NonNegativeReals)

    # define the constraints
    b.inventory = Constraint(expr=b.i == b.i0 + b.x - d[t])
    b.pos_neg = Constraint(expr=b.i == b.i_pos - b.i_neg)
    b.prod_indicator = Constraint(expr=b.x <= P * b.y)
model.lsb = Block(model.T, rule=lotsizing_block_rule)
```

Here, we are defining the variables and constraints for a single time period t within the rule. The `Block` component is then indexed over the set `model.T`, and the declaration constructs a lot-sizing block for each entry in `model.T`. Therefore, our `model` object now contains a block for each time period t. All that remains is to provide constraints linking the blocks together (setting the initial inventory of one block equal to the final inventory of the previous block), and to define the objective function over all the blocks. The full code listing for the block version of the multi-period lot-sizing problem is shown below.

```
from pyomo.environ import *

model = ConcreteModel()
model.T = RangeSet(5) # time periods

i0 = 5.0 # initial inventory
c = 4.6 # setup cost
h_pos = 0.7 # inventory holding cost
h_neg = 1.2 # shortage cost
P = 5.0 # maximum production amount

# demand during period t
d = {1: 5.0, 2:7.0, 3:6.2, 4:3.1, 5:1.7}

# create a block for a single time period
def lotsizing_block_rule(b, t):
    # define the variables
    b.y = Var(domain=Binary)
    b.x = Var(domain=NonNegativeReals)
    b.i = Var()
    b.i0 = Var()
    b.i_pos = Var(domain=NonNegativeReals)
    b.i_neg = Var(domain=NonNegativeReals)

    # define the constraints
    b.inventory = Constraint(expr=b.i == b.i0 + b.x - d[t])
    b.pos_neg = Constraint(expr=b.i == b.i_pos - b.i_neg)
    b.prod_indicator = Constraint(expr=b.x <= P * b.y)
```

```
model.lsb = Block(model.T, rule=lotsizing_block_rule)

# link the inventory variables between blocks
def i_linking_rule(m, t):
   if t == m.T.first():
      return m.lsb[t].i0 == i0
   return m.lsb[t].i0 == m.lsb[t-1].i
model.i_linking = Constraint(model.T, rule=i_linking_rule)

# construct the objective function over all the blocks
def obj_rule(m):
   return sum(c*m.lsb[t].y + h_pos*m.lsb[t].i_pos + \
       h_neg*m.lsb[t].i_neg for t in m.T)
model.obj = Objective(rule=obj_rule)

### solve the problem
solver = SolverFactory('glpk')
solver.solve(model)

# print the results
for t in model.T:
   print('Period: {0}, Prod. Amount: {1}'.format(t, \
       value(model.lsb[t].x)))
```

This formulation is small, so it can be difficult to see the benefit of blocks. However, as models grow in size and complexity, this object-oriented modeling concept allows us to define small pieces of the model in self-contained chunks of code, and then build the large model by pulling these pieces together. This example was selected in part because it is a heavily studied, classic multi-stage inventory model. One can easily imagine extensions to the model that include additional constraints and costs. In fact, many such models have appeared in the academic literature and in practical application. In large models, it is common to write methods or classes that define individual blocks and reuse that code within several different, high-level optimization formulations.

Chapter 9
Generalized Disjunctive Programming

Abstract This chapter documents how to express and solve Generalized Disjunctive Programs (GDPs). GDP models provide a structured approach for describing logical relationships in optimization models. We show how Pyomo blocks provide a natural base for representing disjuncts and forming disjunctions, and we how to solve GDP models through the use of automated problem transformations.

9.1 Introduction

A common feature of many discrete optimization problems is the selection among two or more discrete choices. These decisions imply additional restrictions on the feasible problem space. For example, a *semi-continuous variable* either has value zero or must be above a given value. This can be expressed algebraically as either $x = 0$ or $L \leq x \leq U$, where $L > 0$ and U is allowed to be infinite. When U is finite, this property can be enforced by defining a boolean indicator variable y and defining the following constraints:

$$\begin{aligned} Ly &\leq x \\ x &\leq Uy \end{aligned}$$

When U is infinite, then the second equation is replaced with a so-called "Big-M" equation:

$$x \leq My$$

for a value M that is sufficiently big to not limit the space of feasible solutions.

This approach to formulating a switching decision can be generalized to switch between different groups of constraints. For example, consider the Unit Commitment Problem, where the goal of the optimization problem is to determine the minimal cost schedule for turning on and off a fleet of generators in order to meet

expected demand. In this case, a generator can exist in one of several states: on, off, starting up, and shutting down. Selecting a particular state implies additional constraints on the generator operation. If a generator is "on", then the output power is bounded between the minimum (nonzero) and maximum power levels. In contrast, when the generator is "off", the output power must be 0. Further, the output power in any time period must be within "ramp limits" of output power in the previous time period.

Common implementations of these state selection rules express all the constraints and relax them based on the state variables using so-called "Big-M" terms:

$$Power_{g,t} \leq MaxPower_g \cdot GenOn_{g,t} \quad (9.1)$$

$$Power_{g,t} \geq MinPower_g \cdot GenOn_{g,t} \quad (9.2)$$

$$Power_{g,t} \leq Power_{g,t-1} + RampUpLimit_g + M_g \cdot (1 - GenOn_{g,t}) \quad (9.3)$$

$$Power_{g,t} \geq Power_{g,t-1} - RampDownLimit_g - M_g \cdot (1 - GenOn_{g,t}) \quad (9.4)$$

$$Power_{g,t} \leq MaxPower_g \cdot (1 - GenOff_{g,t}) \quad (9.5)$$

$$Power_{g,t-1} \leq ShutDownRampLimit_g + MaxPower_g \cdot (1 - GenOff_{g,t}) \quad (9.6)$$

$$Power_{g,t} \leq StartUpRampLimit_g + MaxPower_g \cdot (1 - GenStartUp_{g,t}) \quad (9.7)$$

$$GenOn_{g,t} + GenOff_{g,t} + GenStartUp_{g,t} = 1 \quad (9.8)$$

$$GenOn_{g,t} \leq GenOn_{g,t-1} + GenStartUp_{g,t-1} \quad (9.9)$$

$$GenStartUp_{g,t} \leq GenOff_{g,t-1} \quad (9.10)$$

$$GenOn_{g,t}, GenOff_{g,t}, GenStartUp_{g,t}, GenShutDown_{g,t} \in \{0,1\} \quad (9.11)$$

$$Power_{g,t} \in (0, MaxPower_g) \quad (9.12)$$

This approach to formulating a switching decision has two significant limitations. First, the relationships between the binary selection variable and the corresponding constraints that the binary variable selects is somewhat obfuscated. Second, the use of "Big-M" relaxations is only one of several possible approaches to formulating the problem. By hard-coding that relaxation into your model, you are effectively precluding the possibility of exploring alternative approaches (like a convex hull relaxation) without significant effort rewriting the model.

Generalized Disjunctive Programming [75] represents an alternative approach to representing problems with significant logical structure. It generalizes the concepts of Disjunctive Programming [6] for integer linear problems to also include nonlinear systems. The canonical GDP model [55] augments the objective, variables, and constraints of a typical MI(N)LP with Boolean variables, disjunctions, and logical constraints:

$$\min \quad \sum_{k \in K} c_k + f(x) \tag{9.13}$$

$$s.t. \quad r(x) \leq 0 \tag{9.14}$$

$$\bigvee_{j \in J_k} \begin{bmatrix} Y_{jk} \\ g_{jk}(x) \leq 0 \\ c_k = \gamma_{j,k} \end{bmatrix} \quad \forall k \in K \tag{9.15}$$

$$\Omega(Y) = True \tag{9.16}$$

$$x \geq 0, c_k \geq 0, Y_{jk} \in \{True, False\} \tag{9.17}$$

In this framework, the logical decisions are represented as sets of disjunctions (Eqn. 9.15) and logical constraints (Eqn. 9.16). Each disjunction contains a number of terms (*disjuncts*) connected by an "OR" operator. Each disjunct contains a Boolean indicator variable (Y) and a set of constraints that is only enforced when Y is *True*. Additional constraints that enforce logical relationships among the indicator variables are imposed through Eqn. 9.16.

Recasting the generator state model above as a GDP yields the following disjunction:

$$\begin{bmatrix} Y_{g,on} \\ MinPower_g \leq Power_{g,t} \leq MaxPower_g \\ -RampDownLimit_g \leq Power_{g,t} - Power_{g,t-1} \leq RampUpLimit_g \end{bmatrix}$$
$$\bigvee \begin{bmatrix} Y_{g,off} \\ Power_{g,t} = 0 \\ Power_{g,t-1} \leq ShutDownRampLimit_g \end{bmatrix}$$
$$\bigvee \begin{bmatrix} Y_{g,startup} \\ Power_{g,t} \leq StartUpRampLimit_g \end{bmatrix} \tag{9.18}$$

This modeling approach directly addresses the two limitations of typical MI(N)LP models discussed above: the relationship between the switching variable and the constraints it implies is now explicit in the model structure, and the model is no longer locked into any particular relaxation.

9.2 Modeling GDP in Pyomo

The `pyomo.gdp` package extends the core modeling environment to represent GDP models. This package defines two new constructs: the *disjunct* and the *disjunction*. We implement these constructs as two new components in `pyomo.gdp`: `Disjunct` and `Disjunction`, respectively. The components are imported from the GDP package:

```
from pyomo.gdp import *
```

A disjunct is logically a container for the indicator variable and the corresponding

constraints. Here we see the power of the hierarchical modeling approach enabled by the `Block` component: the `Disjunct` component is naturally derived from the `Block` class. As with blocks, `Disjunct` components may be arbitrarily indexed and initialized through rules. In addition, they may contain any Pyomo modeling component, including not only `Set`s, `Param`s, `Var`s, and `Constraint`s, but also `Block`s, `Disjunct`s, and `Disjunction`s. The only thing that the `Disjunct` class adds to the normal `Block` implementation is the implicit and automatic definition of the disjunct's indicator variable.

For our generator state example, the requisite three disjuncts are declared as follows:

```
model.NumTimePeriods = Param()
model.GENERATORS = Set()
model.TIME = RangeSet(model.NumTimePeriods)

model.MaxPower = Param(model.GENERATORS, \
    within=NonNegativeReals)
model.MinPower = Param(model.GENERATORS, \
    within=NonNegativeReals)
model.RampUpLimit = Param(model.GENERATORS, \
    within=NonNegativeReals)
model.RampDownLimit = Param(model.GENERATORS, \
    within=NonNegativeReals)
model.StartUpRampLimit = Param(model.GENERATORS, \
    within=NonNegativeReals)
model.ShutDownRampLimit = Param(model.GENERATORS, \
    within=NonNegativeReals)

def Power_bound(m,g,t):
   return (0, m.MaxPower[g])
model.Power = Var(model.GENERATORS, model.TIME, \
    bounds=Power_bound)

def GenOn(b, g, t):
   m = b.model()
   b.power_limit = Constraint(
      expr=m.MinPower[g] <= m.Power[g,t] <= m.MaxPower[g] )
   if t == m.TIME.first():
      return
   b.ramp_limit = Constraint(
      expr=-m.RampDownLimit[g] <= m.Power[g,t] - \
          m.Power[g,t-1] <= m.RampUpLimit[g] )
model.GenOn = Disjunct(model.GENERATORS, model.TIME, \
    rule=GenOn)

def GenOff(b, g, t):
   m = b.model()
   b.power_limit = Constraint(
      expr=m.Power[g,t] == 0 )
   if t == m.TIME.first():
      return
   b.ramp_limit = Constraint(
      expr=m.Power[g,t-1] <= m.ShutDownRampLimit[g] )
```

```
model.GenOff = Disjunct(model.GENERATORS, model.TIME, \
    rule=GenOff)

def GenStartUp(b, g, t):
   m = b.model()
   b.power_limit = Constraint(
      expr=m.Power[g,t] <= m.StartUpRampLimit[g] )
model.GenStartup = Disjunct(model.GENERATORS, model.TIME, \
    rule=GenStartUp)
```

Note that while the disjuncts may be completely self-contained, with their own local variables, parameters, and constraints, they may also reference Pyomo components outside their immediate scope.

The `Disjunction` component is used to associate a set of disjuncts. A disjunction is similar to a constraint, in that it can be indexed and defined through rules. However, unlike a `Constraint`, where the rule returns a relational expression, the rule for a `Disjunction` must return a list of `Disjuncts`. Subsequent model transformations will convert the `Disjunction` component and generate the binding constraint across the disjuncts. While the general form of a GDP relates the disjuncts using an "OR" operator, the vast majority of models actually expect an "exclusive OR" operator. This is so common that the default behavior of the `Disjunction` component is to generate the "exclusive OR" relationship. Modelers may, however, specify the original "OR" operator by providing `exclusive=False` to the `Disjunction` declaration.

The disjunction for our generator state example is an exclusive OR, and can be expressed in Pyomo using:

```
def bind_generators(m,g,t):
  return [m.GenOn[g,t], m.GenOff[g,t], m.GenStartup[g,t]]
model.bind_generators = Disjunction(model.GENERATORS, \
    model.TIME, rule=bind_generators)
```

Finally, we must be able to express any additional logical constraints on the disjunct indicator variables. In Pyomo, we support this by explicitly forming constraints over the implicit binary `indicator_var` variables. Again, for the generator state example, we can express the switching rules as:

```
def bind_generators(m,g,t):
  return [m.GenOn[g,t], m.GenOff[g,t], m.GenStartup[g,t]]
model.bind_generators = Disjunction(model.GENERATORS, \
    model.TIME, rule=bind_generators)
```

9.3 Solving GDP models

None of the optimization solvers currently interfaced with Pyomo can directly parse or manipulate disjunctive models. However, Pyomo includes the capability to *transform* a disjunctive model into an equivalent MI(N)LP model by relaxing the dis-

junctive constraints. Pyomo's GDP package provides two automated relaxations: the first relaxes the disjunctive constraints by adding so-called "Big-M" terms (recovering the original model structure from Section 9.1) and the second explicitly generates the convex hull of the individual disjunctions.

9.3.1 Big-M transformation

The Big-M transformation performs a constraint-by-constraint relaxation of the original disjunctive model. This preserves the size (number of variables and constraints) of the original model at the expense of possibly generating a weak continuous relaxation.

This transformation begins by recasting each disjunct as a normal block, modifying the individual constraints to add the Big-M term. For equality constraints and 2-sided inequality constraints (those with both upper and lower bounds), the transformation duplicates the constraint as two one-sided inequality constraints before relaxing each. The values of the M parameters can be specified through a `BigM Suffix` placed on the `Disjunct`. When transforming linear constraints over bounded variables, this value can be estimated automatically by the transformation.

Finally, the Big-M transformation recasts the `Disjunctions` by converting them into their equivalent algebraic form; that is either

$$\sum_{k \in K} d_k.indicator\_var = 1 \tag{9.19}$$

for exclusive disjunctions (the default), or

$$\sum_{k \in K} d_k.indicator\_var \geq 1 \tag{9.20}$$

for non-exclusive disjunctions.

The transformation name `gdp.bigm` is used to apply the Big-M transformation.

9.3.2 Convex hull transformation

The convex hull transformation relaxes the original disjunctive model by generating the set of constraints defining the convex hull of each disjunction. This increases the overall size of the model (both the number of variables and constraints), but gives a tighter continuous relaxation than the Big-M transformation. However, all variables must be bounded to apply the convex hull disjunction.

The transformation follows the procedure of Balas [6] for linear disjunctions and Lee and Grossmann [55] (with modifications from Sawaya and Grossmann [77]) for

nonlinear disjunctions. In both cases, variables appearing in any constraint within a disjunction are disaggregated by replacing them with a disjunct-specific variable, and then adding an additional constraint to the model that equates the original variable with the sum of the disaggregated variables.

Finally, the convex hull transformation recasts the `Disjunctions` by converting them into their equivalent algebraic form; that is either

$$\sum_{k \in K} d_k.indicator\_var = 1 \tag{9.21}$$

for exclusive disjunctions (the default), or

$$\sum_{k \in K} d_k.indicator\_var \geq 1 \tag{9.22}$$

for non-exclusive disjunctions.

The transformation name `gdp.chull` is used to apply the convex hull transformation.

9.4 A mixing problem with semi-continuous variables

The following model illustrates a simple mixing problem with three semi-continuous variables (x_1, x_2, x_3) which represent quantities that are mixed to meet a volumetric constraint. In this simple example, the number of sources is minimized:

```
# scont.py
from pyomo.environ import *
from pyomo.gdp import *

L = [1,2,3]
U = [2,4,6]
index = [0,1,2]

model = ConcreteModel()
model.x = Var(index, within=Reals, bounds=(0,20))

# Each disjunct is a semi-continuous variable
# x[k] == 0 or L[k] <= x[k] <= U[k]
def d_rule(block, k, i):
   m = block.model()
   if i == 0:
      block.c = Constraint(expr=m.x[k] == 0)
   else:
      block.c = Constraint(expr=L[k] <= m.x[k] <= U[k])
model.d = Disjunct(index, [0,1], rule=d_rule)

# There are three disjunctions
def D_rule(block, k):
   model = block.model()
```

```
   return [model.d[k,0], model.d[k,1]]
model.D = Disjunction(index, rule=D_rule)

# Minimize the number of x variables that are nonzero
model.o = Objective(expr=sum(model.d[k,1].indicator_var \
    for k in index))

# Satisfy a demand that is met by these variables
model.c = Constraint(expr=sum(model.x[k] for k in index) \
    >= 7)
```

There are three ways to apply either the Big-M or convex hull transformation to solve this model:

1. through the `pyomo` command line,
2. through a scripting interface, or
3. through a `BuildAction`.

On the pyomo command line, the `--transform` command line option is used to apply a transformation:

```
pyomo solve scont.py --transform gdp.bigm --solver=glpk
```

The equivalent approach when developing custom scripts is to create the transformation before applying it to the model:

```
xfrm = TransformationFactory('gdp.bigm')
xfrm.apply_to(model)

solver = SolverFactory('glpk')
status = solver.solve(model)
```

Finally, there are situations where you will want to inject transformations into models that are generated and manipulated in environments other than the `pyomo` command or custom scripts (e.g., the `runph` script). In this case, you can trigger the transformation by adding a `BuildAction` to the model:

```
def transform_gdp(m):
  xfrm = TransformationFactory('gdp.bigm')
  xfrm.apply_to(m)
model.transform_gdp = BuildAction(rule=transform_gdp)
```

Chapter 10
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 PySP writing the extensive form and invoking a standard deterministic solver. For more complex stochastic programs, PySP includes an implementation of Rockafellar 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 and the embedding of the base deterministic model in that language, PySP provides completely generic and highly configurable solver implementations.

10.1 Introduction

From the earliest days of using computers for optimization problems, it was recognized that input data is uncertain in most real-world decision problems [19]. In cases where information becomes available in a few decision stages, stochastic programming is an appropriate and widely studied mathematical framework to express and solve uncertain decision problems [10, 52, 54, 80, 84]. 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.

The `pyomo.pysp` package can express a stochastic program in a generic manner. 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 [11, 32].

Once the deterministic and scenario tree models have been specified, PySP pro-

vides two paths for solving the corresponding stochastic program. The first alternative involves PySP writing the extensive form and optionally invoking a deterministic (mixed-integer) linear solver. For more complex stochastic programs, a generic implementation of Rockafellar and Wets' Progressive Hedging algorithm [76] 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 and the embedding of the base deterministic model in that language. Additionally, PySP leverages Pyomo to provide access to the full range of solvers supported by Pyomo. Consequently, a broad range of model types can be addressed by PySP.

10.2 Stochastic Programming: Definition and Notation

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 [10], King and Wallace [54], Shapiro et al. [80], and Wallace and Ziema [84].

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 knowledge of the future 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* and we use *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 relevant index i. 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. We refer to the terminal nodes as *leaf nodes.* Scenario trees are discussed in more detail in Section 10.3.2.

10.3 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 [10] "farmer" stochastic program. Mirroring several other approaches to modeling stochastic programs (e.g., see Thénié et al. [82]), we require the specification of two related components: the base model and the scenario tree. In Section 10.3.1 we discuss the specification of the deterministic reference model and some associated test data; Section 10.3.2 details the model and data underlying PySP scenario tree specification. The mechanisms for specifying uncertain parameter data are discussed in Section 10.3.3.

10.3.1 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 expects that the reference model – specified in Pyomo – is contained in a file named `ReferenceModel.py`, but this can be changed by specifying the path to the file (including the file name) instead of just the directory when specifying the model directory.

The following is the complete reference model file for Birge and Louveaux's farmer problem:

```
# ReferenceModel.py
#
# Farmer: rent out version has a scalar root node var
# note: this will minimize

from pyomo.environ 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))

# Constraints

def ConstrainTotalAcreage_rule(model):
   return summation(model.DevotedAcreage) <= \
       model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = \
    Constraint(rule=ConstrainTotalAcreage_rule)

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, \
    rule=EnforceCattleFeedRequirement_rule)

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, \
```

```
    rule=LimitAmountSold_rule)

def EnforceQuotas_rule(model, i):
   return (0.0, model.QuantitySubQuotaSold[i], \
       model.PriceQuota[i])

model.EnforceQuotas = Constraint(model.CROPS, \
    rule=EnforceQuotas_rule)

# Stage-specific cost computations

def ComputeFirstStageCost_rule(model):
   return summation(model.PlantingCostPerAcre, \
       model.DevotedAcreage)

model.FirstStageCost = \
    Expression(rule=ComputeFirstStageCost_rule)

def ComputeSecondStageCost_rule(model):
   expr = summation(model.PurchasePrice, \
       model.QuantityPurchased)
   expr -= summation(model.SubQuotaSellingPrice, \
       model.QuantitySubQuotaSold)
   expr -= summation(model.SuperQuotaSellingPrice, \
       model.QuantitySuperQuotaSold)
   return expr

model.SecondStageCost = \
    Expression(rule=ComputeSecondStageCost_rule)

# PySP Auto-generated Objective

# minimize: sum of StageCosts
#
# An active scenario objective equivalent to that \
    generated by PySP is
# included here for informational purposes.
def total_cost_rule(model):
   return model.FirstStageCost + model.SecondStageCost
model.Total_Cost_Objective = \
    Objective(rule=total_cost_rule, sense=minimize)
```

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 `Expression` (a singleton variable or an element of a variable array can also be used, but named `Expression` components are preferred). See Section 4.8 for more information about named Pyomo `Expression` components. In the reference model we simply label the first and second stage cost `Expression` components as `FirstStageCost` and `SecondStageCost`, respectively.

The following is a data file for the farmer reference model:

```
# above mean scenario

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 real 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 ;
```

Although Pyomo supports various data file formats, the example illustrates the use of a data command file.

10.3.2 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 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 by PySP with a scenario tree model, which itself happens to be a Pyomo model (but users of PySP do not really need to know that). The semantics for each of the parameters (or sets) indicated in the scenario tree model file 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 by 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 10.3.3).

`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 variable names indicate 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.

`StageDerivedVariables` An indexed set mapping stage names to sets of variable names in the reference model. The sets of variable names indicate variables that are associated with the given stage that are derived from other variables (i.e., auxiliary variables). This explicitly exempts these variables from inclusion in anticipativity constraints.

`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 10.3.3 for further details.

`StageCost` A parameter indexed by stages giving the `Expression` for costs in each stage that is defined in the reference model.

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 the `ScenarioStructure.dat` data command file:

```
# 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 StageCost := FirstStage FirstStageCost
               SecondStage SecondStageCost ;
```

This example illustrates how PySP provides a simple "wildcard" syntax to specify subsets of indexed variables. The asterisk 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 wildcard syntax allows for 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.

NOTE: When using PH, it is particularly important to declare variables that are implied by other variables as `StageDerivedVariables`.

10.3.3 Scenario Parameter Specification

10.3.3.1 Abstract Models

For abstract models, the 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 approach 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 even though there is redundancy in this encoding of the model parameters. Scenario-based data specification is the default behavior in PySP.

Node-based parameter specification is provided as an alternative to the default scenario-based approach, principally to eliminate 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:

- `A`boveAverageNode.dat
- `A`verageNode.dat
- `B`elowAverageNode.dat

The choice between node- and scenario-based data input is an aesthetic choice as much as a computational one, or perhaps more so. Scenario-based method offers the advantage that is very easy to solve any particular scenario instance using the `pyomo` command for debugging purposes.

10.3.3.2 Concrete Models

For concrete models, scenario data is provided by a `pysp_instance_creation_callback` function inside the `ReferenceModel.py` file.

For example, the following code can be added to the `ReferenceModel.py` for the farmer example.

```
#
# Stochastic Data
#
Yield = {}
Yield['BelowAverageScenario'] = \
   {'WHEAT':2.0,'CORN':2.4,'SUGAR_BEETS':16.0}
Yield['AverageScenario'] = \
   {'WHEAT':2.5,'CORN':3.0,'SUGAR_BEETS':20.0}
Yield['AboveAverageScenario'] = \
   {'WHEAT':3.0,'CORN':3.6,'SUGAR_BEETS':24.0}

def pysp_instance_creation_callback(scenario_name, \
    node_names):

   instance = model.clone()
   instance.Yield.store_values(Yield[scenario_name])

   return instance
```

This function is called for each of the scenarios specified in the `ScenarioStructure.dat` file. The first argument to the function is the name of the scenario and the second argument is the list of scenario tree node names for the scenario. Note that if `ScenarioBasedData` is assigned the value `True` in `ScenarioStructure.dat` it will be ignored for concrete models because the scenarios are given as `ConcreteModel` instances. There is a great deal of flexibility in how the scenario instances are created, but there must be a separate model instance for each callback invocation.

In the farmer example, a single concrete model is constructed outside of the callback, which uses mutable `Param` components for the stochastic data. Each time the callback is invoked the model is cloned (this is more efficient than constructing from scratch), and the mutable model parameters are loaded with their scenario specific data. In these examples, the data for the base model, plus the data for the scenarios is provided by constants hard-coded in the model file; however, the data could have been read from files using standard Python facilities.

Users of the `phpyro` solver manager should be aware that each `phsolverserver` independently imports the model file into an isolated python process. This means that it is possible that the callback will only be invoked over a subset of the scenarios (within each process).

10.4 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 10.2 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-location=MODEL_DIRECTORY`
: Specifies the directory in which the reference model (`ReferenceModel.py`) is stored. The default is the current working directory. If a file name is also supplied, that is used instead of `ReferenceModel.py`.

`--scenario-tree-location=INSTANCE_DIRECTORY`
: Specifies the directory in which 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 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**. To cause `runef` to invoke a solver, this option must be specified.

`--solver=SOLVER_TYPE`
: Specifies the type of solver for solving the extensive form, if a solve is requested. The default is `cplex` because `glpk` does not support quadratic objectives. Note that (`glpk` could be used in conjunction with `--linearize-nonbinary-penalty-terms`.

`--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 (perhaps with dashes added by Pyomo). So for most MIP solvers, the mip gap can be set using `--solver-options= "mipgap=0.01 "`

Multiple options are separated by a space. Options that do not take an argument should be specified with the equals sign followed by either a space or the end of the string.
For example, to specify that the solver is GLPK, then to specify a mipgap of two percent and the GLPK cuts option, use `--solver=glpk --solver-options="mipgap=0.02 cuts="`
If there are multiple "levels" to the keyword, as is the case for some Gurobi and CPLEX options, the tokens are separated by underscore. For example, "mip cuts all" would be specified as `mip_cuts_all`. For another example, to set the solver to be CPLEX, then to set a mip gap of one percent and to specify 'y' for the sub-option "numerical" to the option "emphasis" use `--solver=cplex --solver-options="mipgap=0.001 emphasis_numerical=y"`

`--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.

All options begin with a double dash prefix. The full set of arguments for `runef` can be obtained using the `--help` option.

NOTE: It is common to replace `--model-location` with its alias `-m` and `--scenario-tree-location` with `-s`. The aliases make use of only a single dash and do not use an equal sign.

For example, to write and solve the farmer problem (provided with the Pyomo installation, in the directory `pyomo/examples/pysp/farmer`), the command is:

```
runef -m models -s 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:
            QuantityPurchased[CORN]=48.0
            QuantitySubQuotaSold[SUGAR_BEETS]=4000.0
            QuantitySubQuotaSold[WHEAT]=140.0

      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 cost of (sub)tree rooted at node=-275900.0000

      Name=AverageNode
      Stage=SecondStage
      Parent=RootNode
      Conditional probability=0.3333
      Children:
            None
      Scenarios:
            AverageScenario
      Expected cost of (sub)tree rooted at node=-218250.0000

      Name=BelowAverageNode
      Stage=SecondStage
      Parent=RootNode
      Conditional probability=0.3333
      Children:
            None
      Scenarios:
            BelowAverageScenario
      Expected cost of (sub)tree rooted at node=-157720.0000

      Name=RootNode
      Stage=FirstStage
      Parent=None
      Conditional probability=1.0000
```

```
      Children:
            AboveAverageNode
            AverageNode
            BelowAverageNode
      Scenarios:
            AboveAverageScenario
            AverageScenario
            BelowAverageScenario
      Expected cost of (sub)tree rooted at node=-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
```

The output file format for the extensive form problem instance is controlled by the `--solver-io=SOLVER_IO` option. Different solvers support different types of IO, but the following are common options: lp - generate LP files, nl - generate NL files, python - direct Python interface, os - generate OSiL XML files. If the filename given by the `--output-file` has a filename that is `.lp` or `.nl` , then that will determine the format.

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 [79]. To activate this, use the option `--generate-weighted-cvar` along with `--risk-alpha=RISK_ALPHA` to set the probability threshold associated with CVaR (the default is 0.95) and `--cvar-weight=CVAR_WEIGHT` to give the weight associated with the CVaR term in the risk-weighted objective formulation. The default is 1.0 (which is completely arbitrary for most problem instances). If the weight is 0, then *only* a non-weighted CVaR cost will appear in the EF objective and the expected cost component will be dropped.

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 because the presolvers in commercial solver packages (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`.

NOTE: The `runef` command does not call a solver unless the `--solve` option is given. To specify the solver, use the `--solver=X` option where X is replaced by the solver name.

10.5 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 [81]. In contrast, *horizontal* strategies decompose a stochastic program by scenario; Rockafellar and Wets' Progressive Hedging algorithm [76] and Caroe and Schultz's Dual Decomposition (DD) algorithm [13] are two notable methods in this class.

Progressive Hedging (PH) was initially introduced as a decomposition strategy for solving large-scale stochastic linear programs [76]. 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. Numerous applications to stochastic mixed-integer programs have been reported, such as [18, 21, 47, 57, 59]. 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 accelerate 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 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.

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. 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.

Figure 10.1 provides a formal description of the PH algorithm (with step numbering that matches 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-

1. $k \longleftarrow 0$
2. For all scenario indexes, $s \in \mathscr{S}$:

$$X^{(0)}(s) \longleftarrow \text{argmin} f_s(X(s)) : X(s) \in \Omega_s \tag{10.1}$$

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

$$X^k(s) \longleftarrow \text{argmin} f_s(X(s)) + w^{(k)}(s)X(s) + \rho/2 \left\| X(s) - \overline{X}^{k-1} \right\|^2 : X(s) \in \Omega_s. \tag{10.2}$$

6. If the termination criteria are not met (e.g., solution discrepancies quantified via a metric $g^{(k)}$), then go to Step 3.

Fig. 10.1: A formal description of the Progressive Hedging algorithm.

dependent strategies for computing ρ are often required; mechanisms to support various strategies for setting ρ are described in Section 10.5.1.

10.5.1 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-location` and `--scenario-tree-location` options (and their aliases `-m` and `-s`) are used to specify the PySP problem instance, 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 maximum 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. This option is required.

`--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 0.0001, which is too low for most applications. The default convergence metric is the expected deviation from the mean

$$g^k = \sum_{s \in \mathscr{S}} \Pr(s) ||X^{(k)}(t;s) - \bar{X}^{(k)}(\mathscr{A})||$$

summed across all variables and divided by the number of variables. The option `--enable-normalized-termdiff-convergence` corresponds to this; other options are possible and can be seen by using the `runph --help` command. Most of the convergence metrics that have been implemented in PySP consider only primal values.

For any real application, experimentation and analysis should be applied to obtain a computationally effective configuration of options.

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 --default-rho=1
```

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 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 the default 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 using a ρ value of 1, for example, convergence is achieved in about 50 iterations. It should be noted that for many stochastic linear and small mixed-integer programs (including the farmer example), PH may require *much more computational effort* than the extensive form, primarily because of the overhead associated with communicating with solvers for each scenario, for each PH iteration. However, this overhead can be 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 10.6.

10.5.1.1 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 solutions may "over-shoot" the optimal variable value, resulting in oscillation.

We have developed various strategies for computing variable-specific ρ values [85]. 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 callback 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 [50]:

```
def ph_rhosetter_callback(ph, scenario_tree, scenario):

  MyRhoFactor = 1.0

  root_node = scenario_tree.findRootNode()

  si = scenario._instance
  sm = si._ScenarioTreeSymbolMap

  for i in si.ProductSizes:

    ph.setRhoOneScenario(
      root_node,
      scenario,
      sm.getSymbol(si.NumProducedFirstStage[i]),
      si.UnitProductionCosts[i] * MyRhoFactor * 0.001)

    for j in si.ProductSizes:
      if j <= i:
        ph.setRhoOneScenario(
          root_node,
          scenario,
          sm.getSymbol(si.NumUnitsCutFirstStage[i,j]),
          si.UnitReductionCost * MyRhoFactor * 0.001)
```

See the `rhosetter.py` file in the PySP `sizes` examples directory. 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. The customization strategy underlying the PySP variable-specific ρ mechanism is a limited form of callback function, in which the core PH code temporarily hands control back to a user script to set specific model parameters.

10.5.1.2 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. 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 provides an option for 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. (However, note that in the relaxed problem these terms are not linear.) 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 vari-

able 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 causes half of the BPTS breakpoints to be on either side of the observed variable average at the corresponding node in the scenario tree, with an 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 10.7).

10.5.1.3 Solution Values

Upon termination, the `runph` script attempts to identify an admissible, implementable solution that can be used to compute an expected value for the objective function and to be displayed as the solution. Various strategies can be used, but if the PH algorithm has converged to the point where all scenarios are the same, then it is easy to find such a solution.

At each iteration, PH computes an average value for each variable over the nodes of the scenario tree. We refer to this as $\overline{X}$ in Step 4 of the algorithm given in Figure 10.1. For many problems, particularly those with integer restrictions, $\overline{X}$ might not be feasible for every scenario unless PH happens to be fully converged (in the primal variables). Consequently, the software computes a solution system $\hat{X}$ that is more likely to be feasible for every scenario and will be equivalent to $\overline{X}$ under full convergence. This solution is reported upon completion of PH and its expected value is reported if it is feasible for all scenarios.

Methods for computing $\hat{X}$ are controlled by the `--xhat-method` command-line option. For example `--xhat-method=closest-scenario` causes $\hat{X}$ to be set to the scenario that is closest to $\overline{X}$ (in a z-score sense). Other options, such as `voting` and `rounding`, assign values of $\overline{X}$ to $\hat{X}$ except for binary and general integer variables. The `voting` choice sets the integer values according to probability weighted voting across the scenarios and `rounding` simply rounds $\overline{X}$ to obtain integers. Note that for binary variables these methods are equivalent.

10.5.1.4 Setting Variable Bounds

In some situations, it is helpful to set variable bounds in a callback, perhaps for the purpose of reading data from multiple scenarios. The following example sets

bounds for the networkflow example in the Pyomo distribution. One adds something like this to the `runph` command line:
`--bounds-cfgfile=networkflow/config/xboundsetter.py`

```
# We only need to set upper bounds on first-stage \
    variables, i.e., those
# being blended.

def ph_boundsetter_callback(ph, scenario_tree, scenario):

   # x is a first stage variable
   root_node = scenario_tree.findRootNode()
   scenario_instance = scenario._instance
   symbol_map = scenario_instance._ScenarioTreeSymbolMap
   for arc in scenario_instance.Arcs:
      scenario_instance.x[arc].setlb(0.0)
      scenario_instance.x[arc].setub(scenario_instance.M)
```

10.6 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 10.6.2, 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.

10.6.1 Bundling

The idea behind bundling is simple: instead of having just one scenario as a sub-problem, combine multiple scenarios and solve the resulting EF as a sub-problem (sometimes called a *super-scenario*. Various strategies have been, and are being proposed, for how to form the bundles (see, e.g., [17]) Bundling is specified in the `ScenarioStructure.dat` file.

An example is include with Pyomo in the `SIZES10WithBundles` sub-directory of the `sizes` example. The following lines appear at the bottom of the `ScenarioStructure.dat` file:

```
param Bundling := True ;

set Bundles := EvenBundle OddBundle ;

set BundleScenarios[OddBundle] :=
   Scenario1 Scenario3 Scenario5 Scenario7 Scenario9 ;

set BundleScenarios[EvenBundle] :=
   Scenario2 Scenario4 Scenario6 Scenario8 Scenario10 ;
```

The line `param Bundling := True ;` is required to signal that bundles are to be used. The rest of the lines will not cause bundling to occur unless this line is present. So by commenting out this line, or by changing the right hand side of it to `False`, bundling can be toggled.

The set names `Bundles` and `BundleScenarios` are reserved words, but the names of the bundles (and, of course, the names of the scenarios) are user specified. Any string can be used as the name of a bundle. There can be any number of bundles and they need not contain the same number of scenarios; however, every scenario must be in some bundle.

An alternative to adding these lines to the ScenarioStructure.dat file is to put them in another file and then give that file name as an argument to `--scenario-bundle-specification` on the runph command line. If the specified name ends with a `.dat` suffix, the argument is interpreted as a filename. Otherwise, the name is interpreted as a file in the instance directory, constructed by adding the .dat suffix automatically.

The option `--create-random-bundles=X` creates X random bundles, thus obviating the need for any file to specify the bundles. Note that the option `--scenario-tree-seed=` allows the user to specify the random seed used for bundle formation so that experiments are repeatable. For example, adding the following to the command line results in creation of 10 bundles using the seed 7734:

```
create-random-bundles=10 --scenario-tree-seed=7734
```

10.6.2 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 [85] describe and introduce many of these extensions.

The PySP implementation of PH provides these extensions in the form of a *plugin*, 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) plugin generalizes the accelerator and cycle-avoidance mechanisms described in Watson and Woodruff [85]. The Python module implementing this plugin 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 plugin:

`--enable-ww-extensions`
Enable the Watson-Woodruff PH extensions plugin. This defaults to False.

`--ww-extension-cfgfile`=WW_EXTENSION_CFGFILE
The name of a configuration file for the Watson-Woodruff PH extensions plugin. A good name to use is "wwph.cfg".

`--ww-extension-suffixfile`=WW_EXTENSION_SUFFIXFILE
The name of a variable suffix file for the Watson-Woodruff PH extensions plugin. A good name to use is "wwph.suffixes". Note that these suffixes are not the same as the suffix component as discussed in 4.9.

As discussed in Section 10.6.5, 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, these values frequently do 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, this strategy must be used carefully. In particular, for problems where the constraints effectively limit variables 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 [87] 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 [44]. 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.

10.6.2.1 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.

`InitialMipGap` 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 in section 10.6.2.2) 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`. Examples are shown in the `config` subdirectories of the examples distributed with Pyomo, such as in the `networkflow` example.

10.6.2.2 General Variable Fixing and Slamming Parameters

Variable fixing is often an empirically effective heuristic for accelerating PH convergence. Fixing strategies implicitly rely on strong correlations 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 a 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 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 [85]. 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* [85]. 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 a family 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.

10.6.2.3 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) [85]. 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 [2]. Global defaults are established using the mechanisms described in Section 10.6.2.2, 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 10.6.2.2:

```
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.

Suffixes are supplied to the WW PH extension in a file, which is specified using the `--ww-extension-suffixfile` option. For example, the following yaml format suffix file employs the variables used in the notional examples discussed above:

```
# allow slamming of b0 to any agreed-up value.
RootNode: # or FirstStage
 b0[*,*]:
   CanSlamToLB: True
   CanSlamToMin: True
   CanSlamToAnywhere: True
   CanSlamToMax: True
   CanSlamToUB: True
   SlammingPriority: 1 # equal priority for now.
```

10.6.3 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 [59]. 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 values are currently fixed in any scenario sub-problem are automatically preprocessed into constant terms in any referencing constraints or the objective. As noted in Section 10.4, 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.

10.6.4 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 10.5.1). 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 10.6.3. Note that multiple criteria can be specified.

10.6.5 User-Defined Extensions

The Watson-Woodruff PH extensions described in Section 10.6.2 rely on a simple, general callback framework that allows user-defined extensions to enhance the functionality of the core PH algorithm. While 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 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 pyomo.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 Pyomo user plugins 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., callback) 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.

10.7 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 Pyomo library [42]. Pyomo integrates the third-party, open-source Pyro (Python Remote Objects) package [74] to manage the communication between machines, and Pyomo 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: `pyomo_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 `pyomo_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 `pyomo_ns >& pyomons.log`. The `runph` command is 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 10.5). However, high-throughput computing is often a more important driver for stochastic programming applications than parallel efficiency.

To get better performance for some problems, the `pyro_mip_server` can be replaced by the `phsolverserver` command. The command `pyrosolverserver --help` gives a list of options. If you use the phsolversover, then use `--solver-manager=phpyro` as an argument to runph rather than `--solver-manager=pyro`. Usually, there needs to be one phsolverserver per subproblem.

10.8 Bounds

Using the notation developed in Section 10.5 bounds can be computed after any iteration, k, of PH by computing

$$z^k(s) \longleftarrow \min_{X(s)} f_s(X(s)) + w^{(k)}(s)X(s) : X(s) \in \Omega_s.$$

for all scenario indices, $s \in \mathscr{S}$. A valid bound is then given by

$$\sum_{s \in \mathscr{S}} z^k(s)$$

A proof this result is given in [30] and an application is shown in [38].

A plugin to compute bounds in this way can be invoked by adding the following to a `runph` command:

```
--user-defined-extension=pyomo.pysp.plugins.phboundextension
```

When PH terminates, this plugin will report a history of bounds to the terminal as well as to the file phbound.txt. Setting parameters and file names for this plugin will change in subsequent versions of Pyomo, but at present, the default is to compute the bounds after every PH iteration. Since computing the bounds requires considerable computational effort, it may be desirable to compute the bounds less often. This can be done in either of two ways: by having a file named PHB_.DAT that contains an integer giving the desired frequency, or by creating an operating system environment variable named PHBOUNDINTERVAL and setting its value to the desired frequency of bound computation. For example, the bound would be computed every 10 PH iterations if the file PHB_.DAT contains the number 10 or the system environment variable PHBOUNDINTERVAL=10. If the file exists, the environment variable is not used. Note that the file name has an underscore character before the period.

An *outer bound*. is a lower bound for minimization problems and it is an upper bound for maximization problems. Sometimes, `runph` is used to try to find bounds and it is desirable to terminate when the bounds are good enough. The options `--enable-outer-bound-convergence` and `--outer-bound-convergence-threshold=VAL` cause PH to terminate when the outer bound has reached VAL (i.e., is at or above VAL for a minimization problem).

Chapter 11
Differential Algebraic Equations

Abstract This chapter documents how to formulate and solve optimization problems with differential and algebraic equations (DAEs). The `pyomo.dae` package allows users to easily incorporate detailed dynamic models within an optimization framework and is flexible enough to represent a wide variety of differential equations. We also demonstrate several automated solution techniques included in `pyomo.dae` that apply a simultaneous discretization approach to solve dynamic optimization problems.

11.1 Introduction

In order to develop a better understanding of real-world phenomena, scientists and engineers often develop dynamic, or differential equation based, models. High fidelity simulation of these models can often be difficult and computationally expensive and is still an active research area in many fields. But after a model suitable for simulation has been developed, the next goal is often to optimize a particular aspect of the dynamic system. (e.g., model parameter estimates given dynamic data, or control of the dynamic system to a desired set point). For example, consider the small optimal control problem from [49]:

$$\begin{aligned}
\min \quad & x_3(t_f) && (11.1)\\
\text{s.t.} \quad & \dot{x}_1 = x_2 && (11.2)\\
& \dot{x}_2 = -x_2 + u && (11.3)\\
& \dot{x}_3 = x_1^2 + x_2^2 + 0.005 \cdot u^2 && (11.4)\\
& x_2 - 8 \cdot (t - 0.5)^2 + 0.5 \leq 0 && (11.5)\\
& x_1(0) = 0, x_2(0) = -1, x_3(0) = 0, t_f = 1 && (11.6)
\end{aligned}$$

where the objective is to minimize the value of x_3 at the final time point by finding the optimal values for the input variable u. This problem includes three differential equations as constraints and also includes an inequality constraint restricting the profile of x_2, also known as a path constraint.

While it is easy to write down optimization problems including dynamic models, solving them is hard. Off-the-shelf optimization solvers cannot handle differential equations directly. Therefore, optimization problems including differential equations as constraints, or dynamic optimization problems, must be reformulated in order to be solved. Common solution approaches include single or multiple shooting methods or a full discretization approach. Regardless of the solution strategy, the implementation of the technique is often entwined with the particular model or problem being solved which makes it time-consuming to apply these solution techniques to new dynamic optimization problems or experiment with different solution strategies on the same model.

The `pyomo.dae` package addresses several of these challenges. It provides users the ability to separate the dynamic optimization formulation from the solution strategy used to solve it. This is done by introducing modeling components for representing continuous domains and derivative terms directly. `pyomo.dae` also includes implementations of the simultaneous discretization solution technique which can be applied automatically to a Pyomo model with differential equations.

The remainder of this chapter provides a brief overview of how to use the `pyomo.dae` package. We refer the reader to Nicholson et al. [64] for a more detailed description and information about the design and novelty of this package. This package is still under active development and expansion. Please refer to the online Pyomo documentation for the most up-to-date documentation on new features.

11.2 Pyomo DAE Modeling Components

In order to represent DAE models in Pyomo, the `pyomo.dae` package defines two new components:

- the `ContinuousSet` represents continuous domains over which a derivative can be taken, and
- the `DerivativeVar` represents the derivative of a `Var` with respect to a `ContinuousSet`.

The package is explicitly imported to access these modeling components:

```
from pyomo.environ import *
from pyomo.dae import *
```

The `ContinuousSet` component functions similarly to the regular Pyomo `Set`. It can be used to index other Pyomo components such as `Var`, `Constraint`, or `Expression`. A `ContinuousSet` can be thought of as a bounded virtual set. In order to construct a `ContinuousSet` you must supply numeric values repre-

senting the upper and lower bounds of the continuous domain being represented. For our optimal control example, the continuous domain t is declared as follows.

```
m.tf = Param(initialize=1)
m.t = ContinuousSet(bounds=(0,m.tf))
```

A separate `ContinuousSet` must be declared for each continuous domain in the model. After declaration, it can be used to index other Pyomo components and to declare derivatives in conjunction with the `DerivativeVar` component. A `DerivativeVar` must be declared for each derivative that appears in the dynamic model. Furthermore, you can only take the derivative of a `Var` with respect to a `ContinuousSet` that is included as an indexing set of the variable. The variables and derivatives for our optimal control examples can be declared using:

```
m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)

m.dx1 = DerivativeVar(m.x1, wrt=m.t)
m.dx2 = DerivativeVar(m.x2, wrt=m.t)
m.dx3 = DerivativeVar(m.x3)
```

Notice that the positional argument supplied to a `DerivativeVar` component is the `Var` being differentiated. The indexing sets for a `DerivativeVar` are inherited from and identical to those of the `Var` being differentiated. If a variable is indexed by multiple `ContinuousSets` then the 'wrt' or 'withrespectto' keyword argument is used to specify the desired derivative. In addition, high-order derivatives can also be declared with the `DerivativeVar` component. For example, a second order derivative can be specified with:

```
m.dx1dt2 = DerivativeVar(m.x1,wrt=(m.t,m.t))
```

Differential equations can be formulated using standard Pyomo constraint components. For example, the differential equations for our optimal control example are implemented with:

```
def _x1dot(m, t):
    if t == m.t.first():
        return Constraint.Skip
    return m.dx1[t] == m.x2[t]
m.x1dotcon = Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    if t == m.t.first():
        return Constraint.Skip
    return m.dx2[t] == -m.x2[t]+m.u[t]
m.x2dotcon = Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    if t == m.t.first():
        return Constraint.Skip
    return m.dx3[t] == m.x1[t]**2+m.x2[t]**2+0.005*m.u[t]**2
m.x3dotcon = Constraint(m.t, rule=_x3dot)
```

Because differential equations are formulated as constraints, the `pyomo.dae` package does not impose a particular form or structure on the differential equations. The differential equations will by default be enforced at the boundaries of the continuous domain. Depending on the dynamic model, this might not be desired. You can use `Constraint.Skip` to override the enforcement of a differential equation at one or more bounds of a continuous domain.

The last important aspect of any dynamic optimization problem is the specification of initial or boundary conditions. This can be achieved either by explicitly indexing a `Var` or `DerivativeVar` with one of the bounds of a `ContinuousSet` or by using the 'first' or 'last' accessor functions on the `ContinuousSet`. For example, the initial conditions for the optimal control example can be expressed as:

```
def _init(m):
    yield m.x1[0] == 0
    yield m.x2[m.t.first()] == -1
    yield m.x3[m.t.first()] == 0
m.init_conditions = ConstraintList(rule=_init)
```

Here we have grouped all the initial conditions into a single `ConstraintList` component, but they could also be formulated as individual constraints.

The last pieces of our optimal control example, the objective function (11.1) and the path constraint (11.5) are implemented with:

```
m.obj = Objective(expr=m.x3[m.tf])

def _con(m, t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

11.3 Solving Pyomo Models with DAEs

Having formulated a Pyomo model with differential equations, we now describe how to solve it. None of the optimization solvers interfaced with Pyomo can currently handle differential equations directly. The only solution technique currently included with `pyomo.dae` is a simultaneous discretization approach, also called direct transcription. This approach discretizes the continuous domains in the model and approximates the differential equations using algebraic equations defined at the discretization points. The result of this discretization transformation is a purely algebraic model that can be solved with a standard nonlinear programming solver.

There are two types of discretization schemes included in `pyomo.dae`: finite difference and collocation. The schemes differ in the algebraic equations used to approximate the derivative but they are applied using nearly identical syntax. A discretization is applied to a particular continuous domain and propagated to each derivative and constraint over that domain. After you specify the discretization scheme and the resolution of the discretization (number of discretization points)

`pyomo.dae` will automatically add the necessary discretization points to the appropriate `ContinuousSet` and add additional constraints to the Pyomo model with the discretization equations. This has the effect of *transforming* the DAE model into an algebraic model. Unlike other Pyomo transformations, `pyomo.dae` transformations cannot currently be applied from the `pyomo` command line, you must create a transformation object and apply a discretization transformation to a model.

11.3.1 Finite Difference Transformation

Finite difference methods approximate the derivative at a particular point using a difference equation and are among the simplest discretization schemes to conceptually understand and implement. There are many variations which differ in the choice of points used to approximate the derivative. The backward difference method, also called implicit or backward Euler, is the most common variation. To illustrate the discretization equations associated with this method we first define the following derivative and differential equation (constraint):

$$\left(\frac{dx(t)}{dt}, f(x(t), u(t))\right) = 0, \quad t \in [0, T]. \tag{11.7}$$

After applying the backward difference method to the continuous domain t, the resulting derivative and constraint pair is

$$\left.\frac{dx}{dt}\right|_{t_{k+1}} = \frac{x_{k+1} - x_k}{h}, \quad k = 0, ..., N-1 \tag{11.8}$$

$$g\left(\left.\frac{dx}{dt}\right|_{t_{k+1}}, f(x_{k+1}, u_{k+1})\right) = 0, \quad k = 0, ..., N-1 \tag{11.9}$$

where $x_k = x(t_k)$, $t_k = kh$, and h is the step size between discretization points or the size of each finite element. When a finite difference transformation is applied to a Pyomo model, the discretization equations such as (11.8) are automatically generated and added to the Pyomo model as equality constraints.

The code required to apply the backward finite difference method to our optimal control example is as follows:

```
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, nfe=20, wrt=m.t, scheme='BACKWARD')
```

The `nfe` keyword argument stands for "number of finite elements" and is used to specify the number of discretization points to be used in the discretization. The `scheme` keyword specifies which finite difference method to apply. There currently are three finite difference schemes included in `pyomo.dae`: backward finite difference ('BACKWARD'), central finite difference ('CENTRAL'), and forward finite

difference ('FORWARD').

11.3.2 Collocation Transformation

The second type of discretization included in `pyomo.dae` is collocation or more specifically, orthogonal collocation over finite elements. This approach works by first breaking a continuous domain into $N-1$ segments known as finite elements. Over each of these segments, the profiles of the differential variables (variables whose derivatives appear in the model) are approximated using polynomials. The polynomials are defined using K collocation points that appear as discretization points within each finite element. Continuity is enforced at the finite element boundaries for the differential variables. To provide a formal, mathematical representation of this approach applied to the derivative and differential equation (11.7) we have:

$$\left.\frac{dx}{dt}\right|_{t_{ij}} = \frac{1}{h_i}\sum_{j=0}^{K} x_{ij}\frac{d\ell_j(\tau_k)}{d\tau}, \quad k=1,\ldots,K,\ i=1,\ldots,N-1 \tag{11.10}$$

$$0 = g\left(\left.\frac{dx}{dt}\right|_{t_{ij}}, f(x_{ik},u_{ik})\right), \quad k=1,\ldots,K,\ i=1,\ldots,N-1 \tag{11.11}$$

$$x_{i+1,0} = \sum_{j=0}^{K} \ell_j(1)x_{ij}, \quad i=1,\ldots,N-1 \tag{11.12}$$

where $t_{ij} = t_{i-1} + \tau_j h_i$, $x(t_{ij}) = x_{ij}$. Further, we note that the solution $x(t)$ is interpolated as follows:

$$x(t) = \sum_{j=0}^{K} \ell_j(\tau)x_{ij}, \quad t \in [t_{i-1}, t_i], \quad \tau \in [0,1] \tag{11.13}$$

$$\ell_j(\tau) = \prod_{k=0,\neq j}^{K} \frac{(\tau-\tau_k)}{(\tau_j-\tau_k)}. \tag{11.14}$$

Collocation methods produce significantly more accurate algebraic approximations compared to finite difference methods. However, they are much harder to implement manually. Variations of collocation methods differ in the functional representation of the differential variable profile over each finite element as well as the selection of the collocation points. As of this writing, the collocation transformations in `pyomo.dae` use Lagrange polynomials to represent differential variable profiles. Two options are available for the selection of collocation points: shifted Gauss-Radau roots ('LAGRANGE-RADAU') and shifted Gauss-Legendre roots ('LAGRANGE-LEGENDRE').

A collocation discretization can be applied to a Pyomo model using:

```
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m,nfe=7,ncp=6,scheme='LAGRANGE-RADAU')
```

The `nfe` keyword argument specifies the number of finite elements and the `ncp` argument specifies the number of collocation points within each finite element.

11.4 Additional Features

There are several advanced features included `pyomo.dae`. In this section, we briefly mention two such features that will be of interest for users interested in PDE constrained optimization or more advanced optimal control strategies.

11.4.1 Applying Multiple Discretizations

As mentioned previously, a separate discretization transformation can be applied to each `ContinuousSet` that appears in the model. This means that different finite difference or collocation schemes or a combination of the two can be applied to a single Pyomo model. For example, if a Pyomo model had two `ContinuousSets` ('model.t1' and 'model.t2'), it might be discretized with any of the following combinations of discretization schemes:

```
# Apply multiple finite difference schemes
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, wrt=m.t1, nfe=10, scheme='BACKWARD')
discretizer.apply_to(m, wrt=m.t2, nfe=100, scheme='FORWARD')
```

```
# Apply multiple collocation schemes
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m, wrt=m.t1, nfe=10, ncp=6, \
    scheme='LAGRANGE-LEGENDRE')
discretizer.apply_to(m, wrt=m.t2, nfe=100, ncp=3, \
    scheme='LAGRANGE-RADAU')
```

```
# Apply a combination of finite difference and
# collocation schemes
discretizer1 = \
    TransformationFactory('dae.finite_difference')
discretizer2 = TransformationFactory('dae.collocation')
discretizer1.apply_to(m, wrt=m.t1, nfe=10, scheme='BACKWARD)
discretizer2.apply_to(m, wrt=m.t2, nfe=100, ncp=3, \
    scheme='LAGRANGE-RADAU')
```

11.4.2 Restricting Control Input Profiles

One of the main design considerations for the `pyomo.dae` package was the extensibility of the package to include general implementations of common operations applied to dynamic optimization problems. One such common operation in the area of optimal control is restricting the control input to have a certain profile, typically piecewise constant or piecewise linear. These profiles are often desired when a model has been discretized using collocation over finite elements and the control variable is restricted to be constant over each finite element. The `pyomo.dae` package includes a function for doing this after a collocation discretization has been applied to a model. It works by reducing the number of free collocation points for a particular variable. For example, to restrict our control input u to be piecewise constant in our small optimal control problem you would add the following line right after applying a discretization transformation:

```
discretizer.reduce_collocation_points(m, var=m.u, ncp=1, \
    contset=m.t)
```

The `ncp` keyword argument specifies the number of free collocation points per finite element for the variable specified by the keyword `var`. Specifying `ncp=1` restricts u to have a single free collocation point (or degree of freedom) rendering it constant over each finite element. The function works by adding constraints to the discretized model which force any extra, undesired collocation points to be interpolated from the others.

11.4.3 Plotting

After formulating, discretizing, and solving a dynamic optimization problem, `pyomo.dae` makes it easy to plot the resulting dynamic profiles. Because a `ContinuousSet` is populated with actual numerical values, the user can directly create Python lists from it for plotting. Any variable indexed by a `ContinuousSet` will have a value for each point in the `ContinuousSet`, after the model has been solved. Therefore, creating a Python list for the variable values is just as straightforward as for a `ContinuousSet`

The Python script shown below puts everything together. Assuming the Pyomo model has been declared and saved in a separate file, the script shows how to apply a discretization and solve the model.

```
from pyomo.environ import *
from pyomo.dae import *
from path_constraint import m

# Discretize model using Orthogonal Collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m,nfe=7,ncp=6,scheme='LAGRANGE-RADAU')
```

```
discretizer.reduce_collocation_points(m, var=m.u, ncp=1, \
    contset=m.t)

solver=SolverFactory('ipopt')
results = solver.solve(m, tee=True)
```

Finally, the code below shows an example implementation of a plotter function using `matplotlib` for plotting. The resulting figure is also shown below.

```
def plotter(subplot, x, *y, **kwds):
   plt.subplot(subplot)
   for i,_y in enumerate(y):
      plt.plot(list(x), [value(_y[t]) for t in x], \
          'brgcmk'[i%6])
      if kwds.get('points', False):
         plt.plot(list(x), [value(_y[t]) for t in x], 'o')
   plt.title(kwds.get('title',''))
   plt.legend(tuple(_y.name for _y in y))
   plt.xlabel(x.name)

import matplotlib.pyplot as plt
plotter(121, m.t, m.x1, m.x2, title='Differential \
    Variables')
plotter(122, m.t, m.u, title='Control Variable', \
    points=True)
plt.show()
```

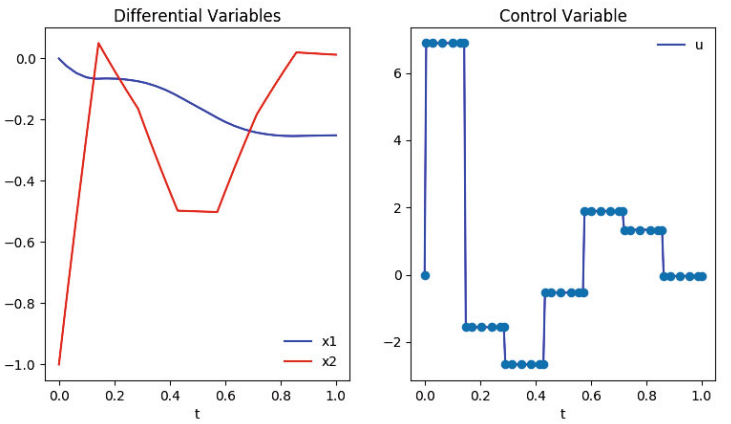


Fig. 11.1: Plot produced by `matplotlib` for the optimal control example

Chapter 12
Mathematical Programs with Equilibrium Constraints

Abstract This chapter documents how to formulate mathematical programs with equilibrium constraints (MPECs), which naturally arise in a wide range of engineering and economic systems. We describe Pyomo components for complementarity conditions, and transformation capabilities that automate the reformulation of MPEC models, and meta-solvers that leverage these transformations to support global and local optimization of MPEC models.

12.1 Introduction

Mathematical Programs with Equilibrium Constraint (MPEC) problems arise in a large number of applications in engineering and economic systems [23, 60, 68]. An MPEC is an optimization problem that includes equilibrium constraints in the form of complementarity conditions. Equilibrium constraints naturally arise as the solution to an optimization subproblem (e.g., for bilevel programs), variational inequalities, and complementarity problems [40].

Since MPEC problems frequently arise in practice, many algebraic modeling languages (AML) have integrated capabilities for expressing complementarity conditions [63], including AMLs like AIMMS [1], AMPL [2, 29], GAMS [31], MATLAB [62] and YALMIP [58]. In this chapter, we describe new functionality in Pyomo for expressing and optimizing MPEC models. MPEC models can be easily expressed with Pyomo modeling components for complementarity conditions. Further, Pyomo's object-oriented design naturally supports the ability to automate the reformulation of MPEC models into other forms (e.g., disjunctive programs). We describe Pyomo meta-solvers that transform MPECs as MIP or NLP problems, which are then optimized with standard solvers. Further, we describe interfaces to specialized mixed complementarity problem solvers, which solve MPEC problems expressed without an optimization objective.

12.2 Modeling Equilibrium Conditions

12.2.1 Complementarity Conditions

Ferris et al. [24] note that there are a few fundamental forms that account for a wide range of complementarity conditions that arise in practice. Consider a variable x and function $g(x)$. The classical form of complementarity condition can be expressed as

$$x \geq 0 \perp g(x) \geq 0,$$

which expresses the complementarity restriction that at least one of these must hold with equality. When the variable x is bounded such that $x \in [l,u]$, then a *mixed complementarity condition* can be expressed as

$$l \leq x \leq u \perp g(x),$$

which expresses the complementarity restriction that at least one of the following must hold:

$$\begin{aligned} x = l \quad & \text{and } g(x) \geq 0, \\ x = u \quad & \text{and } g(x) \leq 0, \\ \text{or } l < x < u \text{ and } & g(x) = 0. \end{aligned}$$

These forms can be generalized by substituting a function $f(x)$ for the variable x. Thus, a *generalized mixed complementarity condition* can be expressed as

$$l \leq f(x) \leq u \perp g(x),$$

which expresses the complementarity restriction that at least one of the following must hold:

$$\begin{aligned} f(x) = l \quad & \text{and } g(x) \geq 0, \\ f(x) = u \quad & \text{and } g(x) \leq 0, \\ \text{or } l < f(x) < u \text{ and } & g(x) = 0. \end{aligned}$$

For completeness, note that the complementarity condition

$$f(x) \perp g(x) = 0$$

is a special case where the function $f(x)$ is unbounded.

12.2.2 Complementarity Expressions

The design of complementarity conditions in Pyomo relies on the specification of Pyomo constraint expressions. A Pyomo constraint expression defines an equality, a simple inequality, or a pair of inequalities. For example:

$$expr_1 = expr_2$$
$$expr_1 <= expr_2$$
$$const_1 <= expr_2 <= const_2$$

where $const_i$ are constant arithmetic expressions that may only contain variables that fixed, and $expr_i$ are arithmetic expressions that contain unfixed variables.

A complementarity condition is defined with a pair of constraint expressions

$$l_1 <= expr_1 <= u_1 \perp l_2 <= expr_2 <= u_2,$$

where exactly two of the constant bounds l_1, u_1, l_2 and u_2 are finite. The non-finite bounds values are omitted in practice, so this condition directly describes a classical or mixed complementarity condition. Additionally, a complementarity condition can be expressed with a simple inequality:

$$l_1 <= expr_1 <= u_1 \perp expr_2 <= expr_3.$$

This complementarity condition is implicitly transformed to a form with constant bounds:

$$l_1 <= expr_1 <= u_1 \perp expr_2 - expr_3 <= 0.$$

12.2.3 Modeling Mixed-Complementarity Conditions

Pyomo employs an object-oriented strategy for representing models. Consequently, Pyomo's modeling capabilities can be extended by defining new modeling components. Pyomo's `pyomo.mpec` package defines the `Complementarity` component that is used to declare complementarity conditions.

For example, consider the `ralph1` problem in MacMPEC [61]:

$$\begin{aligned} \min\ & 2x - y \\ & 0 \le y \perp y \ge x \\ & x, y \ge 0 \end{aligned}$$

The following script defines a Pyomo model for `ralph1`:

```
# ralph1.py
from pyomo.environ import *
from pyomo.mpec import *

model = ConcreteModel()

model.x = Var( within=NonNegativeReals )
model.y = Var( within=NonNegativeReals )

model.f1 = Objective( expr=2*model.x - model.y )

model.compl = Complementarity(
```

```
                expr=complements(0 <= model.y,
                                 model.y >= model.x) )
```

The first lines in this script import Pyomo packages. The `pyomo.environ` packages initializes Pyomo's environment, and `pyomo.mpec` defines modeling components for complementarity conditions. The subsequent lines in this script create a model, declare variables `x` and `y`, declare an objective `f1`, and declare a complementarity condition `compl`.

The complementarity condition is declared with the `Complementarity` component. In the simplest case, this Python class takes a keyword argument `expr` that contains the value of the `complements` function. This function accepts two Pyomo constraint expressions that are used to declare a complementarity condition.

Pyomo also supports indexed components, where a set of components are initialized over an index set using a construction rule. Thus, the `Complementarity` component can be declared with an index set. For example, consider the following model, `indexed`:

$$\begin{aligned} &\min \textstyle\sum_{i=1}^{n} i(x_i - 1)^2 \\ &\quad 0 \le x_i \perp 0 \le x_{i+1} \quad i = 1, \ldots, n-1 \end{aligned}$$

The following script defines a Pyomo model for `indexed` with $n = 5$:

```
# ex1a.py
from pyomo.environ import *
from pyomo.mpec import *

n = 5

model = ConcreteModel()

model.x = Var( range(1,n+1) )

model.f = Objective(expr=sum(i*(model.x[i]-1)**2
                     for i in range(1,n+1)) )

def compl_(model, i):
    return complements(model.x[i] >= 0, model.x[i+1] >= 0)
model.compl = Complementarity( range(1,n), rule=compl_ )
```

The complementarity conditions are defined with a single `Complementarity` component that is indexed over the set $1, \ldots, n-1$ and initialized with a construction rule `compl_`. This rule is a function that accepts a model instance and an index, and returns the i-th complementarity condition.

The declared set of indexes may be a superset of the indices that define complementarity conditions. If a construction rule returns `Complementarity.Skip`, then the corresponding index is skipped. For example:

```
# ex1d.py
from pyomo.environ import *
from pyomo.mpec import *

n = 5

model = ConcreteModel()

model.x = Var( range(1,n+1) )

model.f = Objective(expr=sum(i*(model.x[i]-1)**2
                     for i in range(1,n+1)) )

def compl_(model, i):
    if i == n:
        return Complementarity.Skip
    return complements(model.x[i] >= 0, model.x[i+1] >= 0)
model.compl = Complementarity( range(1,n+1), rule=compl_ )
```

This example can also be expressed with the `ComplementarityList` component:

```
# ex1b.py
from pyomo.environ import *
from pyomo.mpec import *

n = 5

model = ConcreteModel()

model.x = Var( range(1,n+1) )

model.f = Objective(expr=sum(i*(model.x[i]-1)**2
                     for i in range(1,n+1)) )

model.compl = ComplementarityList()
model.compl.add(complements(model.x[1]>=0, model.x[2]>=0))
model.compl.add(complements(model.x[2]>=0, model.x[3]>=0))
model.compl.add(complements(model.x[3]>=0, model.x[4]>=0))
model.compl.add(complements(model.x[4]>=0, model.x[5]>=0))
```

This component defines a list of complementarity conditions. The list index can be used in Pyomo, but this component simplifies the declaration of models for which the index values are not important. The `ComplementarityList` component can also be defined with a rule that iteratively returns complementarity conditions:

```
# ex1c.py
from pyomo.environ import *
from pyomo.mpec import *

n = 5

model = ConcreteModel()
```

```
model.x = Var( range(1,n+1) )

model.f = Objective(expr=sum(i*(model.x[i]-1)**2
                    for i in range(1,n+1)) )

def compl_(model):
    yield complements(model.x[1] >= 0, model.x[2] >= 0)
    yield complements(model.x[2] >= 0, model.x[3] >= 0)
    yield complements(model.x[3] >= 0, model.x[4] >= 0)
    yield complements(model.x[4] >= 0, model.x[5] >= 0)
model.compl = ComplementarityList( rule=compl_ )
```

Similarly, the construction rule may be a list expression that generates a sequence of complementarity conditions:

```
# ex1e.py
from pyomo.environ import *
from pyomo.mpec import *

n = 5

model = ConcreteModel()

model.x = Var( range(1,n+1) )

model.f = Objective(expr=sum(i*(model.x[i]-1)**2
                    for i in range(1,n+1)) )

model.compl = ComplementarityList(
    rule=(complements(model.x[i] >= 0, model.x[i+1] >= 0)
          for i in range(1,n)) )
```

12.3 MPEC Transformations

Pyomo's object-oriented design supports the structured transformation of models. Pyomo can iterate through model components as well as nested model blocks. Thus, model components can be easily transformed locally, and global data can be collected to support global transformations. Further, Pyomo components and blocks can be activated and deactivated, which facilitates *in place* transformations that do not require the creation of a separate copy of the original model.

Pyomo's `pyomo.mpec` package defines several model transformations that can be easily applied. For example, if `model` defines an MPEC model (as in our previous examples), then the following example illustrates how to apply a model transformation:

```
xfrm = TransformationFactory("mpec.simple_nonlinear")
transformed = xfrm.create_using(model)
```

In this case, the `mpec.simple_nonlinear` transformation is applied. The fol-

lowing sections describe the transformations currently supported in `pyomo.mpec`.

12.3.1 Standard Form

In Pyomo, a complementarity condition is expressed a pair of constraint expressions

$$l_1 <= expr_1 <= u_1 \ \perp \ l_2 <= expr_2 <= u_2,$$

where exactly two of the constant bounds l_1, u_1, l_2 and u_2 are finite. The non-finite bounds are typically omitted, but the value `None` can be used to express infinite bounds. Additionally, each constraint expression can be expressed with a simple inequality of the form

$$expr_1 <= expr_2.$$

The `mpec.standard_form` transformation reformulates each complementarity condition in a model into a standard form:

$$l_1 <= expr <= u_1 \ \perp \ l_2 <= var <= u_2,$$

where exactly two of the constant bounds l_1, u_1, l_2 and u_2 are finite, and either l_2 is zero or both l_2 or u_2 are finite.

Note that this transformation creates new variables and constraints as part of this transformation. For example, the complementarity condition

$$1 \leq x+y \ \perp \ 1 \leq 2x-y,$$

get re-expressed as the following:

$$\begin{aligned} &1 \leq x+y \\ &v = 2x-y-1 \\ &v \in \mathbb{R}, v \geq 0 \end{aligned}$$

For each complementary condition object, the new variable and constraints are added as additional components within the complementarity object. Thus, the overall structure of the MPEC model is not changed by this transformation.

12.3.2 Simple Nonlinear

The `mpec.simple_nonlinear` transformation begins by applying the `mpec.standard_form` transformation. Subsequently, a nonlinear constraint is created that defines the complementarity condition. This is a simple nonlinear transformation adapted from Ferris et al. [25], which can be described by three different cases:

- If l_1 is finite, then the following constraint is defined:

$$(expr - l_1) * v \leq \varepsilon$$

- If u_1 is finite, then the following constraint is defined:

$$(u_1 - expr) * v \leq \varepsilon$$

- If l_2 and u_2 are both finite, then the following constraints are defined:

$$(var - l_2) * expr \leq \varepsilon$$
$$(var - u_2) * expr \leq \varepsilon$$

Each of these cases ensure that the complementarity condition is met when ε is zero. For example, in the first case, we know that $0 \leq v$ and $0 \leq expr - l_1$. When ε is zero, this constraint ensures that either v is zero or $expr - l_1$ is zero.

This transformation uses the parameter `mpec_bound`, which defines the value for ε for every complementarity condition. This allows for the specification of a relaxed nonlinear problem, which may be easier to optimize with some nonlinear programming solvers. The default value of `mpec_bound` is zero.

12.3.3 Simple Disjunction

The `mpec.simple_disjunction` transformation expresses a complementarity condition as a disjunctive program. We are given a complementarity condition defined with a pair of constraint expressions

$$l_1 <= expr_1 <= u_1 \ \perp \ l_2 <= expr_2 <= u_2,$$

where exactly two of the constant bounds l_1, u_1, l_2 and u_2 are finite. Without loss of generality, we assume that either l_1 or u_1 is finite.

This transformation can be described by three different cases:

- If the first constraint is an equality, then the complementarity condition is trivially replaced by that equality constraint.
- If both bounds on the first constraint are finite but different, then the disjunction has the form:

$$\begin{bmatrix} Y_1 \\ l_1 = expr_1 \\ expr_2 \geq 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_2 \\ expr_1 = u_1 \\ expr_2 \leq 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_3 \\ l_1 \leq expr_1 \leq u_1 \\ expr_2 = 0 \end{bmatrix}$$

$$Y_1 \veebar Y_2 \veebar Y_3 = True$$

$$Y_1, Y_2, Y_3 \in \{True, False\}$$

- Otherwise, each constraint is a simple inequality. The complementarity condition is reformulated as

$$0 <= \overline{expr}_1 \perp 0 <= \overline{expr}_2,$$

and the disjunction has the form:

$$\begin{bmatrix} Y \\ 0 = \overline{expr}_1 \\ 0 \leq \overline{expr}_2 \end{bmatrix} \bigvee \begin{bmatrix} \neg Y \\ 0 \leq \overline{expr}_1 \\ 0 = \overline{expr}_2 \end{bmatrix}$$

$$Y \in \{True, False\}$$

This transformation makes use of modeling components and transformations from Pyomo's `pyomo.gdp` package. The transformation expresses each of the disjunctive terms explicitly using `Disjunct` components and the *select exactly one* logical condition using the `Disjunction` component. The transformation adds the `Disjunct` and `Disjunction` components within the objects that represent the complementarity conditions. It then recasts the modified complementarity components into simple `Block` components. This localizes all changes to the model to the individual complementarity components. Subsequent transformation of the disjunctive expressions to algebraic constraints can be effected through either Big-M (`gdp.bigm`) or Convex Hull (`gdp.chull`) transformations.

12.3.4 AMPL Solver Interface

Solvers like PATH [22] have been tailored to work with the AMPL Solver Library (ASL). AMPL uses `nl` files to communicate with solvers, which read `nl` files with the ASL. Pyomo can also create `nl` files, and the `mpec.nl` transformation processes `Complementarity` components into a canonical form that is suitable for this format [24].

12.4 Solver Interfaces and Meta-Solvers

Pyomo supports interfaces to third-party solvers as well as meta-solvers that apply transformations and third-party solvers, perhaps in an iterative manner. The `pyomo.mpec` package includes an interface to the `PATH` solver, as well as several meta-solvers. These are described in this section, and examples are provided that employ the `pyomo` command-line interface.

12.4.1 Nonlinear Reformulations

The `mpec.simple_nonlinear` transformation provides a generic way for transforming an MPEC into a nonlinear program. When the MPEC only has continuous decision variables, the resulting model can be optimized by a wide range of solvers.

For example, the `pyomo` command-line interface allows the user to specify a nonlinear solver and a model transformation that is applied to a model:

```
pyomo solve --solver=ipopt \
            --transform=mpec.simple_nonlinear ex1a.py
```

This example illustrates the use of the `ipopt` interior-point solver to solve a problem generated with the `mpec.simple_nonlinear` transformation. When a transformation is used directly like this, the results that are returned to the user include decision variables for the transformed model. Pyomo does not have general capabilities for mapping a solution back into the space from the original model. In this example, the results object includes values for the x variables as well as the variables v introduced when applying the transformation to the standard form (see above).

Pyomo includes a meta-solver, `mpec_nlp` that applies the nonlinear transformation, performs optimization, and then returns results for the original decision variables. For example, `mpec_nlp` executes the same logic as the previous `pyomo` example:

```
pyomo solve --solver=mpec_nlp ex1a.py
```

Additionally, this meta-solver can also manipulate the ε values in the model, starting with larger values and iteratively tightening them to generate a more accurate model.

```
pyomo solve --solver=mpec_nlp \
            --solver-options="epsilon_initial=0.1 \
                epsilon_final=1e-7" \
            ex1a.py
```

This approach may be useful when using a nonlinear solver that has difficulty optimizing with equality constraints.

12.4.2 Disjunctive Reformulations

The `mpec.simple_disjunction` transformation provides a generic way for transforming an MPEC into a disjunctive program. The `mpec_minlp` solver applies this transformation to create a nonlinear disjunctive program, and then further reformulates the disjunctive model using a "Big-M" transformation that is provided by the `pyomo.gdp` package. The resulting transformation is similar to the reformulation of bilevel models described by Fortuny-Amat and McCarl [28]. If the original model was nonlinear, then the resulting model is a mixed-integer nonlinear program (MINLP). Pyomo includes interfaces to solvers that use the AMPL Solver

Library (ASL), so `mpec_minlp` can optimize nonlinear MPECs with a solver like Couenne [15].

If the original model was a linear MPEC, then the resulting model is a mixed-integer linear program that can be globally optimized (e.g., see Hu et al. [46], Júdice [51]). For example, the `pyomo` command can be used to execute the `mpec_minlp` solver using a specified MIP solver:

```
pyomo solve --solver=mpec_minlp \
            --solver-options="solver=glpk" ralph1.py
```

Note that Pyomo includes interfaces to a variety of commonly used MIP solvers, including CPLEX, Gurobi, CBC, and GLPK.

12.4.3 PATH and the ASL Solver Interface

Pyomo's solver interface for the AMPL Solver Library (ASL) applies the `mpec.nl` transformation, writes an AMPL `.nl` file, executes an ASL solver, and then loads the solution into the original model. Pyomo provides a custom interface to the PATH solver [22], which simply allows the solver to be specified as `path` while the solver executable is named `pathamp`.

The `pyomo` command can execute the PATH solver by simply specifying the `path` solver name. For example, consider the `munson1` problem from MCPLIB:

```
# munson1.py
from pyomo.environ import *
from pyomo.mpec import *


model = ConcreteModel()

model.x1 = Var()
model.x2 = Var()
model.x3 = Var()

model.f1 = Complementarity(expr=complements(
                model.x1 >= 0,
                model.x1 + 2*model.x2 + 3*model.x3 >= 1))

model.f2 = Complementarity(expr=complements(
                model.x2 >= 0,
                model.x2 - model.x3 >= -1))

model.f3 = Complementarity(expr=complements(
                model.x3 >= 0,
                model.x1 + model.x2 >= -1))
```

This problem can be solved with the following command:

```
pyomo solve --solver=path munson1.py
```

12.5 Discussion

Pyomo supports the ability to model complementarity conditions in a manner that is similar to other AMLs. For example, Pyomo's `pyomo.data` package [72] includes Pyomo formulations for many of the MacMPEC [61] and MCPLIB [20] models, which were originally formulated in GAMS and AMPL. However, Pyomo does not currently support related modeling capabilities for equilibrium models, variational inequalities and embedded models, which are supported by the GAMS extended mathematical programming framework [26].

The transformations and meta-solvers currently included in Pyomo illustrate how Pyomo's MPEC modeling capability can be leveraged. We expect these capabilities to mature and expand in response to application needs. For example, the `mpec.simple_nonlinear` transformation could be expanded to support reformulations that are well-suited for sequential quadratic programming solvers [56]. Similarly, current meta-solvers could be extended to directly support the communication of suffix information from the solver back to the original model.

Chapter 13
Bilevel Programming

Abstract This chapter documents how to formulate bilevel programs, which model adversarial behavior in a general manner. We describe new modeling components that represent subproblems, modeling transformations for re-expressing models with bilevel structure in other forms, and optimize bilevel programs with meta-solvers that apply transformations and then perform optimization on the resulting model. We illustrate the breadth of Pyomo's modeling capabilities for bilevel programs, and we describe how Pyomo's meta-solvers can perform local and global optimization of bilevel programs.

13.1 Introduction

Many planning situations involve the analysis of several objectives that reflect a hierarchy of decision-makers. For example, policy decisions are made at different levels of a government, each of which has a different objective and decision space. Similarly, robust planning against adversaries is often modeled with a 2-level hierarchy, where the defensive planner makes decisions that account for adversarial response.

Multilevel optimization techniques partition control over decision variables amongst the levels. Decisions at each level of the hierarchy may be constrained by decisions at other levels, and the objectives for each level may account for decisions made at other levels. In practice, multilevel problems have proven difficult to solve, and most of the literature has focused on *bilevel programs*, which model a 2-level hierarchy [8].

Although multilevel problems arise in many applications, few algebraic modeling languages (AML) have integrated capabilities for expressing these problems. AMLs are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems [53]. AMLs provide a mechanism for defining variables and generating constraints with a concise mathematical representation, which is essential for large-scale, real-world

problems that involve thousands or millions of constraints and variables. GAMS, YALMIP and Pyomo provide explicit support for modeling bilevel programs. A variety of other AMLs support the solution of bilevel programs through the expression of Karush-Kuhn-Tucker conditions and associated reformulations using mixed-complementarity conditions, but these reformulations must be expressed by the user in these AMLs.

Pyomo provides an intuitive syntax for expressing multilevel optimization problems. Multilevel models can be easily expressed with Pyomo modeling components for submodels, which can be nested in a general manner. Further, Pyomo's object-oriented design naturally supports the ability to automate the reformulation of multilevel models into other forms. Pyomo can transform bilevel models for several broad classes of problems. We describe Pyomo meta-solvers that transform bilevel programs into mixed integer programs (MIP) or nonlinear programs (NLP), which are then optimized with standard solvers.

13.2 Motivating Problems

In multilevel optimization problems, a subset of decision variables at each level is constrained to take values associated with an optimal solution of a distinct, lower level optimization problem. For example, a general formulation for bilevel programs is

$$\begin{array}{ll} \min_{x \in X, y} & F(x,y) \\ \text{s.t.} & G(x,y) \leq 0 \\ & y \in P(x) \end{array} \tag{13.1}$$

where

$$P(x) = \arg\min_{y \in Y} \; f(x,y) \\ g(x,y) \leq 0$$

$P(x)$ defines a *lower-level problem*, which may have multiple solutions. Here x is the primary upper-level decision, and y is the anticipated lower-level decision.

When $P(x)$ contains multiple solutions, this formulation ensures that the selected value for y will minimize $F(x,y)$. Consequently, this formulation has been called *optimistic* or *cooperative*, since the selection of the lower-level decision variables minimized the upper-level objective. Most research on bilevel programming has considered this problem, since this formulation has an optimal solution under reasonable assumptions.

The following subsections describe specializations of Equation (13.1) that will be explored in greater detail throughout this chapter. These are well-studied classes of bilevel programs that that can be reformulated into other canonical optimization forms.

13.2.1 Linear Bilevel Programs with Continuous Variables

Multilevel linear programming considers the case where decision variables are continuous, and both objectives and constraints are linear. In the 2-level case, we have the following linear bilevel program:

$$\begin{array}{lll} \min_{x,y} & c_1^T x + d_1^T y & \\ \text{s.t.} & A_1 x + B_1 y \leq b_1 & \\ & x \geq 0 & \\ & \min_y & c_2^T x + d_2^T y \\ & \text{s.t.} & A_2 x + B_2 y \leq b_2 \\ & & y \geq 0 \end{array} \tag{13.2}$$

13.2.2 Quadratic Min/Max

Consider the case where the lower-level decisions y do not constrain the upper-level decisions. Let

$$X = \{x \mid A_1 x \leq b, x \geq 0\}$$

Then we have

$$\begin{array}{ll} \min_{x \in X} & \max_{y \geq 0} \; c_1^T x + d_1^T y + x^T Q y \\ & A_2 x + B_2 y \leq b_2 \end{array} \tag{13.3}$$

In our discussion below, we allow the x_i to be binary without loss of generality.

13.3 Modeling Bilevel Programs

The `pyomo.bilevel` package extends Pyomo by defining a new modeling component: `SubModel`. The `SubModel` component defines a subproblem that represents the lower level decisions in a bilevel program. This component is like a `Block` component; any components can be added to a `SubModel` object. In general, a submodel is expected to have an objective, one or more variables and it may define constraints.

The `SubModel` class generalizes the `Block` component by including constructor arguments that denote which variables in the submodel should be considered fixed or variable. When expressions in a submodel refer to variables defined outside of the submodel, the user needs to indicate whether these are fixed values defined by an upper-level problem. Fixed variables are treated as constants within the submodel, but non-fixed variables are defined by the current submodel or by a lower-level problem.

Consider the following example:

$$
\begin{array}{lll}
\min_{x,y,v} & x+y+v & \\
\text{s.t.} & x+v \geq 1.5 & \\
& 1 \leq x \leq 2 & \\
& 1 \leq v \leq 2 & \\
& \max_{y,w} & x+w \\
& \text{s.t.} & y+w \leq 2.5 \\
& & 1 \leq y \leq 2 \\
& & 1 \leq w \leq 2
\end{array}
\tag{13.4}
$$

The following Pyomo model defines four variables, `x`, `v`, `sub.y` and `sub.w`:

```
from pyomo.environ import *
from pyomo.bilevel import *

model = ConcreteModel()
model.x = Var(bounds=(1,2))
model.v = Var(bounds=(1,2))
model.sub = SubModel()
model.sub.y = Var(bounds=(1,2))
model.sub.w = Var(bounds=(-1,1))

model.o = Objective(expr=model.x + model.sub.y + model.v)
model.c = Constraint(expr=model.x + model.v >= 1.5)
model.sub.o = Objective(expr=model.x+model.sub.w, \
    sense=maximize)
model.sub.c = Constraint(expr=model.sub.y + model.sub.w <= \
    2.5)
```

Variables `x` and `v` are declared in the upper-level problem, and `v` only appears in the upper-level problem. Variables `sub.y` and `sub.w` are declared in the submodel. However, note that the `sub.y` variable appears in the upper-level problem, while the `sub.w` variable only appears in the lower-level problem.

These modeling capabilities are quite general. Expressions in submodels can be linear or nonlinear, convex or nonconvex, continuous or discontinuous, and more. Additionally, submodels can be nested to an arbitrary degree. Thus, the range of bilevel programs that can be expressed with Pyomo is quite broad. However, the real challenge is solving these models. The following sections describe two general strategies for solving the two classes of bilevel programs described in Section 13.2.

In each section we describe model transformations and their use in meta-solvers. Pyomo's object-oriented design supports the structured transformation of models. Pyomo can iterate through model components as well as nested model blocks. Thus, model components can be easily transformed locally, and global data can be collected to support global transformations. Further, Pyomo components and blocks can be activated and deactivated, which facilitates *in place* transformations that do not require the creation of a separate copy of the original model.

13.4 Solving Linear Bilevel Programs

We consider the formulation for linear bilevel programs described in Equation (13.2):

$$\begin{array}{lll} \min_{x,y} & c_1^T x + d_1^T y & \\ \text{s.t.} & A_1 x + B_1 y \leq b_1 & \\ & x \geq 0 & \\ & \min_y & c_2^T x + d_2^T y \\ & \text{s.t.} & A_2 x + B_2 y \leq b_2 \\ & & y \geq 0 \end{array} \tag{13.2}$$

Following Bard [7], we can replace the lower-level problem with corresponding optimality conditions. This transformation gives the following model:

$$\begin{array}{ll} \min & c_1^T x + d_1^T y \\ \text{s.t.} & A_1 x + B_1 y \leq b_1 \\ & d_2 + B_2^T u - v = 0 \\ & b_2 - A_2 x - B_2 y \geq 0 \perp u \geq 0 \\ & y \geq 0 \perp v \geq 0 \\ & x \geq 0, y \geq 0 \end{array} \tag{13.5}$$

This transformation results in a mathematical program with equilibrium constraints (MPEC).

Pyomo's `pyomo.bilevel` package automates the application of this model transformation. For example, if `model` defines an linear bilevel program, then the code applies this transformation to change the model in place:

```
xfrm = TransformationFactory('bilevel.linear_mpec')
xfrm.apply_to(model)
```

The `bilevel.linear_mpec` transformation modifies the model by creating a new block of variables and constraints for the optimality conditions in the lower-level problem.

A variety of solution strategies can leverage this transformation to support optimization. Pyomo defines two meta-solvers, which apply the `bilevel.linear_mpec` transformation and then perform optimization with a third-party solver.

Consider the following problem, which is Example 5.1.1. in Bard [7]:

$$\begin{array}{lll} \min_{x,y} & x - 4y & \\ & x \geq 0 & \\ \text{s.t.} & \min_y y & \\ & \text{s.t.} & -x - y \leq -3 \\ & & -2x + y \leq 0 \\ & & 2x + y \leq 12 \\ & & 3x - 2y \leq 4 \end{array} \tag{13.6}$$

Note that the last constraint in this example is negated from the text in Bard [7]; this corrects an error in the example, which is reflected in Bard's discussion of the solution to this example. The Pyomo model for this problem is:

```
# bard511.py
from pyomo.environ import *
from pyomo.bilevel import *

M = ConcreteModel()
M.x = Var(bounds=(0,None))
M.sub = SubModel()
M.sub.y = Var(bounds=(0,None))

M.o = Objective(expr=M.x - 4*M.sub.y, sense=minimize)

M.sub.o = Objective(expr=M.sub.y, sense=minimize)
M.sub.c1 = Constraint(expr=- M.x - M.sub.y <= -3)
M.sub.c2 = Constraint(expr=-2*M.x + M.sub.y <= 0)
M.sub.c3 = Constraint(expr= 2*M.x + M.sub.y <= 12)
M.sub.c4 = Constraint(expr= 3*M.x - 2*M.sub.y <= 4)
```

13.4.1 Global Optimization

Following Fortuny-amat and McCarl [28], Pyomo's `bilevel_blp_global` meta-solver chains together reformulations to generate the following sequence of models:

$$\text{BLP} \Rightarrow \text{MPEC} \Rightarrow \text{GDP} \Rightarrow \text{MIP},$$

where GDP refers to generalized disjunctive programs. Note that this leverages advanced modeling capabilities in Pyomo for MPEC [41] and GDP.

Pyomo provides general support for solving MIPs with commercial and open-source solvers. The `bilevel_blp_global` meta-solver applies these transformations, solves the resulting MIP, and translates the MIP solution back into the original linear bilevel program. The result is a globally optimal solution. For example, the `pyomo` command can be used to execute the `mpec_blp_global` solver using a specified MIP solver:

```
pyomo solve --solver=bilevel_blp_global \
            --solver-options="solver=glpk" bard511.py
```

Note that a "Big-M" transformation is used to convert the GDP model into a MIP. The default M value is very large, which may make it difficult to solve the resulting MIP problem. Hence, this solver includes a `bigM` option that can be used to specify a problem-specific value:

```
pyomo solve --solver=bilevel_blp_global \
            --solver-options="bigM=100 solver=glpk" \
            bard511.py
```

13.4.2 Local Optimization

Pyomo's `bilevel_blp_local` meta-solver chains together reformulations to generate the following sequence of models:

$$\text{BLP} \Rightarrow \text{MPEC} \Rightarrow \text{NLP}.$$

This leverages model transformations in `pyomo.gdp` to transform an MPEC into an NLP through a simple nonlinear transformation adapted from Ferris et al. [25]. For example, the complementarity condition

$$y \geq 0 \perp w \geq 0$$

is transformed to the constraints

$$\begin{aligned} y &\geq 0 \\ w &\geq 0 \\ yw &\leq \varepsilon. \end{aligned}$$

Pyomo provides general support for solving NLPs with commercial and open-source solvers. The `bilevel_blp_local` meta-solver applies these transformations, solves the resulting NLP, and translates the solution back into the original linear bilevel program. In general, the result is a locally optimal solution. For example, the `pyomo` command can be used to execute the `mpec_blp_local` solver using a specified NLP solver:

```
pyomo solve --solver=bilevel_blp_local \
            --solver-options="solver=ipopt" bard511.py
```

Note that the tolerance value ε is initially set to a small value, which some solvers may have difficulty with. This value can be explicitly set with the `mpec_bound` option:

```
pyomo solve --solver=bilevel_blp_local \
            --solver-options="mpec_bound=0.01 solver=ipopt" \
            bard511.py
```

13.5 Solving Quadratic Min-Max Bilevel Programs

We consider the formulation for quadratic min-max bilevel programs described in Equation (13.3):

$$\begin{aligned} \min_{x \in X} \ \max_{y \geq 0} \ & c_1^T x + d_1^T y + x^T Q y \\ & A_2 x + B_2 y \leq b_2 \end{aligned} \tag{13.3}$$

where

$$X = \{x \mid A_1 x \leq b, x \geq 0\}.$$

The lower-level problem is linear, so we can replace the lower-level problem with corresponding optimality conditions. Since, the objectives are opposite and the upper-level constraints do not constrain the lower level decisions, we get a single minimization problem. This transformation gives the following model:

$$\begin{aligned} \min\ & c_1^T x + (b_2 - A_2 x)^T \nu \\ \text{s.t.}\ & B_2^T \nu \geq d_1 + Q^T x \\ & A_1 x \leq b_1 \\ & x \geq 0, \nu \geq 0 \end{aligned} \tag{13.7}$$

In Pyomo, this is implemented as the `bilevel.linear_dual` transformation.

If $A_2 \equiv 0$, then the lower-level problem does not constrain the upper-level decisions. This is a simple case, where the transformation generates a linear program if the upper-level decision variables x are continuous and a MIP if some or all of the x are binary.

More generally, suppose that the upper-level decision variables x are binary and $A_2 \not\equiv 0$. We can linearize the quadratic terms in the objective using `gdp.bilinear` transformation, which creates the following disjunctive representation:

$$\begin{aligned} \min\ & c_1^T x + b_2^T \nu - 1^T z \\ \text{s.t.}\ & B_2^T \nu \geq d_1 + Q^T x \\ & A_1 x \leq b_1 \\ & \begin{pmatrix} x_i = 0 \\ z_i = 0 \end{pmatrix} \wedge \begin{pmatrix} x_i = 1 \\ z_i = A_2^T(*, i)\nu \end{pmatrix} \\ & x_i \geq \{0,1\}, \nu \geq 0 \end{aligned} \tag{13.8}$$

Subsequently, this GDP can be transformed into a MIP using a Big-M transformation.

Consider the following network interdiction problem, for which an attacker eliminates links in a network to minimize the maximum flow through the network from a fixed source s to a fixed destination t. Let N be the nodes in the network through which flow occurs. Let y_{ij} be a variable that indicates flow from node i to node j, and let c_{ij} be the maximum capacity on that arc. The attacker selects b variables x_{ij}; if x_{ij} is one then the arc is removed. This network interdiction problem can be written as:

$$\begin{aligned} \min_{x \in X}\ \max_y\ & \eta \\ & \textstyle\sum_{i \in N} y_{in} = \sum_{j \in N} y_{nj} \quad \forall n \in N && \text{(flow balance constraint)} \\ & \eta \leq \textstyle\sum_{j \in N} y_{sj} && \text{(node s flow)} \\ & \eta \leq \textstyle\sum_{i \in N} y_{it} && \text{(node t flow)} \\ & 0 \leq y_{ij} \leq t_{ij}(1 - x_{ij}) \ \forall \text{ arcs } (i,j) && \text{(capacity constraint)} \end{aligned} \tag{13.9}$$

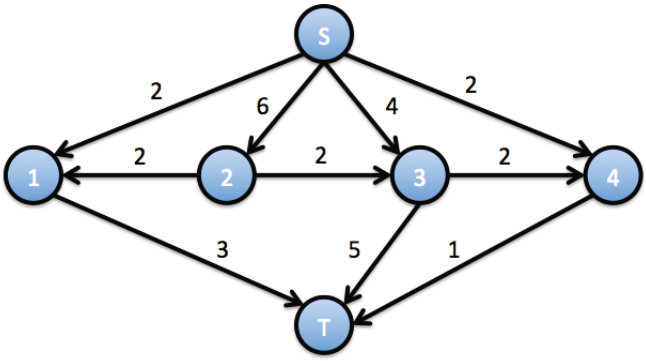
where

$$X = \left\{ x \mid x_{ij} \in \{0,1\}, \sum_{i,j} x_{ij} \leq b \right\}.$$

The following Pyomo model describes this bilevel program:

```
# interdiction.py
from pyomo.environ import *
from pyomo.bilevel import *
from interdiction_data import A, budget

INDEX = list(A.keys())

M = ConcreteModel()
M.x = Var(INDEX, within=Binary)
M.budget = Constraint(expr=summation(M.x) <= budget)
M.sub = SubModel()
M.sub.f = Var()
M.sub.y = Var(INDEX, within=NonNegativeReals)

# Min/Max objectives
M.o = Objective(expr=M.sub.f, sense=minimize)
M.sub.o = Objective(expr=M.sub.f, sense=maximize)

# Flow constraint
def flow_rule(M, n):
   return sum(M.y[i,n] for i in sequence(0,4) if (i,n) in \
       A) == sum(M.y[n,j] for j in sequence(1,5) if (n,j) \
       in A)
M.sub.flow = Constraint(sequence(1,4), rule=flow_rule)

# Source constraint
def s_rule(M):
   model = M.model()
   return model.sub.f <= sum(M.y[0,j] for j in \
       sequence(1,4) if (0,j) in A)
M.sub.s = Constraint(rule=s_rule)

# Destination constraint
def t_rule(M):
   model = M.model()
   return model.sub.f <= sum(M.y[j,5] for j in \
       sequence(1,4) if (j,5) in A)
M.sub.t = Constraint(rule=t_rule)

# Capacity constraint
def c_rule(M, i, j):
   model = M.model()
   return M.y[i,j] <= A[i,j]*(1-model.x[i,j])
M.sub.c = Constraint(INDEX, rule=c_rule)
```

In this example, the file `interdiction_data.py` defines a simple network with 6 nodes (including s and t), adapted from an example by Will Traves:



Pyomo's `bilevel_ld` meta-solver applies the `bilevel.linear_dual` transformation and then applies subsequent transformations when $A_2 \neq 0$.

```
pyomo solve --solver=bilevel_ld\
            --solver-options="bigM=100 solver=glpk"\
            interdiction.py
```

Note that this solver includes a `bigM` option that can be used to specify a problem-specific value when a MIP is generated from a GDP.

13.6 Discussion

Note that Pyomo's ability to model multilevel optimization problem extends far beyond the bilevel programs that are currently supported. For example, declarations of `SubModel` can be arbitrarily nested with clear semantics. For example, consider the following trilevel model [88]:

$$\begin{aligned}
&\min_{x,y,z} \; x - 4y + 2z \\
&\text{s.t.} \quad -x - y \leq -3 \\
&\qquad -3x + 2y - z \geq -10 \\
&\qquad\qquad \min_{y,z} \; x + y - z \\
&\qquad\qquad \text{s.t.} \quad -2x + y - 2z \leq -1 \\
&\qquad\qquad\qquad 2x + y + 4z \leq 14 \\
&\qquad\qquad\qquad\qquad \min_{z} \; x - 2y - 2z \\
&\qquad\qquad\qquad\qquad \text{s.t.} \quad 2x - y - z \leq 2
\end{aligned}$$

The following model illustrates how this trilevel model could be implemented in Pyomo:

```
from pyomo.environ import *
from pyomo.bilevel import *

M = ConcreteModel()
M.x = Var()
M.s = SubModel()
M.s.y = Var()
M.s.s = SubModel()
M.s.s.z = Var()

M.o = Objective(expr= M.x - 4*M.s.y + 2*M.s.s.z)
M.c1 = Constraint(expr= - M.x - M.s.y <= -3)
M.c2 = Constraint(expr= -3*M.x + 2*M.s.y >= -10)
M.s.o = Objective(expr= M.x + M.s.y - M.s.s.z)
M.s.c1 = Constraint(expr=-2*M.x + M.s.y - 2*M.s.s.z <= -1)
M.s.c2 = Constraint(expr= 2*M.x + M.s.y + 4*M.s.s.z <= 14)
M.s.s.o = Objective(expr= M.x - 2*M.s.y - 2*M.s.s.z)
M.s.s.c = Constraint(expr=2*M.x - M.s.y - M.s.s.z <= 2)
```

Pyomo use of object-oriented model specification makes it fundamentally different from the specification of bilevel models in GAMS and YALMIP. Both GAMS and YALMIP allow users to specify expressions for variables, objectives and constraints, and then the users specifies which of these are associated with an upper-level or lower-level problem. This design allows users to mix-and-match different modeling components in a flexible manner. However, it is limited to a strictly bilevel form. By contrast, Pyomo submodels can be nested in an arbitrary manner. This includes multilevel models, as was just illustrated. But it also allows for the specification of a tree of nested submodels. For example, Pyomo supports the specification of independent submodels at the same level, which can be used to model a single agent cooperating with decisions for two independent agents that make subsequent decisions.

Chapter 14
Scripting

Abstract This chapter illustrates the use of Python scripts for solution analysis, the development of high-level algorithms and custom workflows. We discuss how to script the standard workflow: build a model, solve the model, and then analyze the solution. Pyomo also supports development of high-level algorithms and complex workflows. This chapter also contains some larger examples, including a Sudoku solver. Together, these scripting examples illustrate how Pyomo users can go beyond the simple use of the `pyomo` command to formulate, solve, and analyze optimization models.

14.1 Introduction

In previous chapters, we have described how a generic optimization process can be executed with Pyomo to construct a model, solve the model, and display the results. The `pyomo` command can be used to apply a 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 libraries. However, this command masks much of the power underlying Pyomo.

The use of Python provides tremendous flexibility for the Pyomo modeling environment. 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 solution process giving two important high-level capabilities:

- Pyomo users can leverage existing Python libraries for analysis of data both before and after solving the optimization problem.
- Pyomo supports development of algorithms that require problem transformations and multiple solves of problems with different structure and data. When coupled with the programming capabilities of Python, this allows users to build

high-level algorithms (e.g. Bender's decomposition, MINLP solvers, multi-stage initialization strategies).

In this chapter, we will discuss the basics of scripting with Pyomo. This functionality will be demonstrated on some examples, including a Sudoku solver.

NOTE: This chapter shows the power of Pyomo that can be accessed through scripting, and examples in this chapter may make use of methods on components that are part of the core Pyomo infrastructure. The developers of Pyomo try to maintain backwards compatibility where possible. However, note that the methods described in this chapter are more likely to change than other capabilities discussed in this book.

14.2 A Basic Optimization Script

In Chapter 3 we introduced a warehouse location problem. This problem solved for the optimal locations to build warehouses to meet delivery demands. Please refer back to Section 3.2 for a detailed description.

The following example highlights the three basic pieces found in almost any Pyomo script: (1) creating a Pyomo model, (2) performing optimization of the Pyomo model using a solver interface, and (3) interrogating the solution.

```
1  from pyomo.environ import *
2  from warehouse_data import *
3
4  # create the model
5  model = ConcreteModel(name="(WL)")
6  model.x = Var(N, M, bounds=(0,1))
7  model.y = Var(N, within=Binary)
8
9  def obj_rule(model):
10    return sum(d[n,m]*model.x[n,m] for n in N for m in \
          M)
11 model.obj = Objective(rule=obj_rule)
12
13 def one_per_cust_rule(model, m):
14    return sum(model.x[n,m] for n in N) == 1
15 model.one_per_cust = Constraint(M, \
       rule=one_per_cust_rule)
16
17 def warehouse_active_rule(model, n, m):
18    return model.x[n,m] <= model.y[n]
19 model.warehouse_active = Constraint(N, M, \
       rule=warehouse_active_rule)
20
```

```
21  def num_warehouses_rule(model):
22      return sum(model.y[n] for n in N) <= P
23  model.num_warehouses = \
        Constraint(rule=num_warehouses_rule)
24
25  # solve the model
26  solver = SolverFactory('glpk')
27  solver.solve(model)
28
29  # look at the solution
30  model.y.pprint()
```

In this example, line 1 contains the standard Pyomo import, and line 2 imports the data. Lines 4 through 23 create the Pyomo model, lines 24 and 25 create the solver and perform the optimization, and line 30 prints the values of the binary variables at the solution.

Of course, Python allows scripting with Pyomo workflows that are much more powerful than this simple example. In the next three sections, we describe:

1. how to generate the model and modify the model using mutable parameters and other methods on Pyomo components
2. the mechanism for creating an interface to a solver, configuring the solver with options, and interpreting the results object
3. methods for retrieving the solution, including accessing variable values in flat or hierarchical models

NOTE: While scripting is possible with abstract models, it is most common to interact with concrete models when scripting. Therefore, this chapter uses concrete models in the examples. However, as described in Chapter 2, concrete models can be created from abstract models with the `create_instance()` method.

14.3 Creating and Modifying Pyomo Models

In early stages of model design, it is often convenient to embed all data within the model definition. This facilitates a rapid process for testing and extending a problem formulation. The following example shows a minimalistic approach where the data, model, and execution script are contained in a single file.

```
from pyomo.environ import *

model = ConcreteModel()
model.x = Var()
model.o = Objective(expr= model.x)
model.c = Constraint(expr= model.x >= 1)

solver = SolverFactory("glpk")
results = solver.solve(model)

print(results)
```

An alternative strategy for setting up a model is to used a function that takes data as inputs. The following example illustrates the warehouse location problem again, this time defining a function to create the model.

```
from pyomo.environ import *

def create_wl_model(N, M, d, P):
    # create the model
    model = ConcreteModel(name="(WL)")
    model.x = Var(N, M, bounds=(0,1))
    model.y = Var(N, within=Binary)

    def obj_rule(model):
        return sum(d[n,m]*model.x[n,m] for n in N for m in M)
    model.obj = Objective(rule=obj_rule)

    def one_per_cust_rule(model, m):
        return sum(model.x[n,m] for n in N) == 1
    model.one_per_cust = Constraint(M, \
        rule=one_per_cust_rule)

    def warehouse_active_rule(model, n, m):
        return model.x[n,m] <= model.y[n]
    model.warehouse_active = Constraint(N, M, \
        rule=warehouse_active_rule)

    def num_warehouses_rule(model):
        return sum(model.y[n] for n in N) <= P
    model.num_warehouses = \
        Constraint(rule=num_warehouses_rule)

    return model
```

This example defines a function named `create_wl_model` that takes in four arguments (the data for this problem) used to define the constraints on a `ConcreteModel` that it then returns. The model can be created and solved using the following script:

```
from warehouse_data import *
import pyomo.environ as pe
import warehouse_function as wf

# call function to create model
model = wf.create_wl_model(N, M, d, P)

# solve the model
solver = pe.SolverFactory('glpk')
solver.solve(model)

# look at the solution
model.y.pprint()
```

With this approach, multiple instantiations can easily be solved by invoking the function with different inputs inside of a for loop:

```
from warehouse_data import *
import pyomo.environ as pe
import warehouse_function as wf

for pp in [1,2,3]:
   # call function to create model
   model = wf.create_wl_model(N, M, d, pp)

   # solve the model
   solver = pe.SolverFactory('glpk')
   solver.solve(model)

   # look at the solution
   print('--- P = {0} ---'.format(pp))
   model.y.pprint()
```

This example creates the model in a loop, providing different values of `P` (the maximum number of warehouses).

Encapsulating model construction in functions can be extended to include the construction of blocks. Pyomo models or blocks can be added to other models in a hierarchical fashion. Please see Chapter 8 for more details.

14.3.1 Modifying Model Parameters

In the example above, we solved the model for different values of the parameter `P` by creating a new model every time through the loop. It is also possible to take a model that has been solved, change the value of a parameter and solve it again. This requires the use of mutable parameters, which are declared with the `mutable=True` option.

The following example illustrates the use of mutable parameters in the warehouse problem:

```
from warehouse_data import *
from pyomo.environ import *

# create the model
model = ConcreteModel(name="(WL)")
model.P = Param(initialize=P, mutable=True)
model.x = Var(N, M, bounds=(0,1))
model.y = Var(N, within=Binary)

def obj_rule(model):
   return sum(d[n,m]*model.x[n,m] for n in N for m in M)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(model, m):
   return sum(model.x[n,m] for n in N) == 1
model.one_per_cust = Constraint(M, rule=one_per_cust_rule)

def warehouse_active_rule(model, n, m):
   return model.x[n,m] <= model.y[n]
model.warehouse_active = Constraint(N, M, \
    rule=warehouse_active_rule)

def num_warehouses_rule(model):
   return sum(model.y[n] for n in N) <= model.P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

# execute the loop
for pp in [1,2,3]:
   # change the value of the parameter P
   model.P = pp

   # solve the model
   solver = SolverFactory('glpk')
   solver.solve(model)

   # look at the solution
   print('--- P = {0} ---'.format(pp))
   model.y.pprint()
```

The parameter `model.P` is declared mutable, so the loop does not create a new model instance.

14.3.2 Modifying Model Structure

Within Pyomo, it is also possible to modify the structure of a model between solves.

- Pyomo allows addition of new modeling components. For examples, constraints can be added or deleted to a component, and new components can be added to a model.

- Modeling components can be activated and deactivated without changing the data stored in the model. Pyomo simply stores a flag that indicates whether a component is processed and included in the model sent to the solver.
- Variables can be treated as fixed or unfixed (the default).

The following example illustrates these methods for modify model structure:

```
from pyomo.environ import *

model = ConcreteModel()
model.x = Var(bounds=(0,5))
model.y = Var(bounds=(0,1))
model.con = Constraint(expr=model.x + model.y == 1.0)
model.obj = Objective(expr=model.y-model.x)

# solve the problem
solver = SolverFactory('glpk')
solver.solve(model)
print(value(model.x)) # 1.0
print(value(model.y)) # 0.0

# add a constraint
model.con2 = Constraint(expr=4.0*model.x + model.y == 2.0)
solver.solve(model)
print(value(model.x)) # 0.33
print(value(model.y)) # 0.66

# deactivate a constraint
model.con.deactivate()
solver.solve(model)
print(value(model.x)) # 0.5
print(value(model.y)) # 0.0

# activate a constraint
model.con.activate()
solver.solve(model)
print(value(model.x)) # 0.33
print(value(model.y)) # 0.66

# delete a constraint
del model.con2
solver.solve(model)
print(value(model.x)) # 1.0
print(value(model.y)) # 0.0

# fix the variable
model.x.fix(0.5)
solver.solve(model)
print(value(model.x)) # 0.5
print(value(model.y)) # 0.5

# unfix the variable
model.x.unfix()
solver.solve(model)
```

```
print(value(model.x)) # 1.0
print(value(model.y)) # 0.0
```

These basic capability provide a lot of flexibility, and later examples illustrate how Pyomo supports the construction of complex workflows and meta-solvers.

14.4 Using Solvers

Pyomo models can be analyzed with a wide variety of optimization solvers, and there are several types of solver interfaces in Pyomo:

- A *shell solver* is launched as a separate sub-process by running an executable found on the user's `PATH` environment. Pyomo interfaces with these solvers through files; Pyomo generates a file description of the problem, launches the solver, and then loads the results from log files and standard output files. This is a common form of solver.
- A *direct solver* is executed as a subroutine. Pyomo interfaces with these solvers through libraries that are installed and exposed in the form of Python packages. This is an uncommon form of solver, since it relies on Python interfaces to solver libraries.
- A *meta-solver* is a Python script that executes one-or-more other solvers. These solvers are directly integrated into Pyomo, and they execute sub-solvers to perform optimization. Meta-solvers can support the application of sub-solvers to sub-problems, and they also provide a simple way of interfacing to other classes of solvers (e.g. ASL solvers).

Fortunately, the Pyomo interface is the same regardless of the solver type used.

As seen in Section 14.2 the `SolverFactory` function is used to construct a solver interface object. The argument passed to the solver factory specifies the name of the solver being used. In most cases, this is the name of the executable that will be used to solve the problem; however, Pyomo supports shorter names for some solvers. For example, the GLPK solver can be specified with

```
solver = SolverFactory('glpk')
```

The `solve()` method accepts a number of keyword arguments. Pyomo may support multiple interfaces for some solvers, and the interface type can be specified with the `solver_io` keyword argument:

```
# Construct solver object
solver = SolverFactory('gurobi')

# Apply solver and load results into model
solver.solve(model, solver_io='nl')
```

Other keyword arguments include:

- `logfile`: The filename used to store output for shell solvers.
- `solnfile`: The filename used to store the solution for shell solvers.

- `load_solutions`: If this argument is `True` (the default), then solutions are stored in the model. If `False`, then the results object keeps a *raw* representation of the solutions.
- `timelimit`: The number of seconds that a shell solver is run before it is terminated. (default is `None`)
- `report_timing`: If this argument is `True`, then timing information is report by the solver (default is `False`)
- `tee`: If this argument is `True`, then the solver output is both printed to the standard output as well as saved to the log file. If `False` (the default), then the solver output is only saved to the log file.
- `suffixes`: A list of suffixes that are exported to the solver.
- `options`: A dictionary of options to be passed to the underlying solver.

The `options` attribute can used to send solver specific options to the underlying solver. In the following example, we pass the `tee=True` keyword argument to tell Pyomo to send the solver output to the console, and we pass two solver-specific options to GLPK (sending log output to `warehouse.log`, and turning off the MIP presolver). Notice that some solver-specific options do not take values (e.g. `nointopt`), but are rather simple flags to turn on or off particular behavior. For these types of options, set the option value in the dictionary to `None`.

```
from warehouse_data import *
import pyomo.environ as pe
import warehouse_function as wf

# call function to create model
model = wf.create_wl_model(N, M, d, P)

# solve the model
solver = pe.SolverFactory('glpk')
solver_opt = dict()
solver_opt['log'] = 'warehouse.log'
solver_opt['nointopt'] = None
solver.solve(model, options=solver_opt)

# look at the solution
model.y.pprint()
```

14.5 Investigating the Solution

After a model is solved, there are two aspects of the solution to be investigated. The first is the solution status returned from the solver, and the second is the value of the variables and objective function. The solution status can usually be viewed in the console by passing `tee=True` into the solve command, but there are many times when one would like to read this status in code. This section will discuss the results object, and show how to retrieve values from variables after the solve.

14.5.1 Solver Results

The `solve()` method returns a results object that contains status information from the solver. If the solve completes successfully, the solution values are loaded directly into the model. This consists of three steps: (1) storing solutions in the `solutions` attribute of the model, (2) load the values of variables from a selected solution, and (3) remove solutions from the results object. Afterwards, the `results` object only contains meta-data about information about the model and the optimization process. For memory efficiency, it no longer contains the solution itself.

NOTE: By default, when the solver completes, the solution is automatically loaded into the model object and removed from the results object. Because of this, the results object will indicate that it has 0 solutions.

Typically, a solver only returns a single solution; however, there are cases where a solver might return multiple solutions (a pool of solutions). Because of this, the results object supports an interface that looks like a dictionary of lists containing more than one solution. However, for the most common case of a single solution, the results object supports a simple attribute-like interface. The results object returned from the `solve()` method contains a `problem` attribute and a `solver` attribute that contain information about the problem statistics and the solver status respectively.

The `results.solver` attribute contains a `SolverInformation` object. Key attributes of this object are shown in the table below. Examples illustrating the use of the results object are provided later in this chapter.

Table 14.1: Key attributes of the `SolverInformation` object

Attribute	
`status`	Returns the solver status as a PyUtilib.Enum `SolverStatus` that can be: `ok`, `warning`, `error`, `aborted`, or `unknown`.
`termination_condition`	Returns the specific termination condition as reported by the solver. This is a PyUtilib enum `TerminationCondition` that can have different values, including `optimal`, `infeasible`, or `unbounded`. There are many different solver outcomes, and depending on the solver, you may see other outcomes.
`termination_message`	String message returned by the solver summarizing the termination status.

14.5.2 Retrieving Variable Values

After performing optimization, the solver plugin automatically loads the solution into the model instance. This provides a convenient way to analyze the optimization results by accessing the solution through values stored in the model components.

It is straightforward to retrieve the value of a variable using the `value()` function. This can be used for both scalar variables and indices of indexed variables. This is shown in the example below.

```
model.y.pprint()
print(value(model.z))
print(value(model.y['Ashland']))
```

This example illustrates the use of the `.pprint()` method, which prints a preformatted set of output for the variable. In the next two lines, you see printing a scalar variable value, and the value of a particular index of an indexed variable.

NOTE: It is common to forget the `value()` function when retrieving values from Pyomo `Var` components. For example, in the code shown below, Python will actually print the representation of the variable object itself, not the value.

```
print(model.z)
```

In this case, the code will print the variable name (`z`), not the value.

This retrieval of *variable values* is also important when comparing variables. Consider the following example.

```
1  from pyomo.environ import *
2
3  model = ConcreteModel()
4  model.u = Var(initialize=1.0)
5  model.v = Var(initialize=1.0)
6
7  # comparing values
8  print(value(model.u) == value(model.v)) # True
9
10 # comparing variables
11 print(model.u == model.v) # "u == v"
12
13 # following prints "Same"
14 if model.u == model.v:
15    print('Same')
16 else:
17    print('Different')
```

In line 8, the value of two variables are compared, and as expected this generates the value `True`. However, in line 11 the variables are compared directly, which will print string `'u == v'`. In this line, Pyomo is actually creating an expression

u is equal to *v*, so the value printed is the string representation of that expression. Further confusing the situation, lines 14-17 will actually print `Same` to the console. In the condition on line 14, Pyomo creates the expression `u == v`, and then that condition is evaluated. In this example, the evaluation returns True.

This example illustrates the creation of implicit expressions, how they can lead to unintended results and how they can lead to normally expected results. It can be difficult to predict the behavior of Pyomo expressions, so users are strongly encouraged to avoid generating expressions except as part of the model construction process. Comparisons of the values of variable (e.g. line 8) do *not* generate a Pyomo expression, so this type of comparison is always safe!

The previous examples illustrate how to access variables by name or scalar index. It is also straightforward to iterate over values in an indexed variable.

```
for i in model.y:
   print('{0} = {1}'.format(model.y[i], value(model.y[i])))
```

Similarly, the following example shows how to iterate over variables in a generic manner, which is useful in many scripting contexts:

```
for v in model.component_objects(Var):
   for index in v:
      print('{0} = {1}'.format(v[index], value(v[index])))
```

This approach can be used for other Pyomo components as well.

14.6 Scripting Examples

In this section, we will show a few scripting examples that illustrate the capabilities shown above. These are still relatively simple examples, and more examples can be found in the Pyomo Gallery (see www.pyomo.org).

14.6.1 Warehouse Location Loop and Plotting

The following example formulates the warehouse location problem and solves it repeatedly to find every possible solution. Each time a solution is found, a new cut is added that excludes that solution, and the problem is solved again to find the next solution. This process is repeated until the problem is infeasible, and no more solutions can be found. The `ConstraintList` component is used to contain the list of cuts; each time through the loop a new cut is added to this component.

```
from warehouse_data import *
from pyomo.environ import *
from pyomo.opt import TerminationCondition
import warehouse_function as wf
import matplotlib.pyplot as plt
```

```
# call function to create model
model = wf.create_wl_model(N, M, d, P)
model.integer_cuts = ConstraintList()
objective_values = list()
done = False
while not done:
    # solve the model
    solver = SolverFactory('glpk')
    results = solver.solve(model)
    objective_values.append(value(model.obj))
    term_cond = results.solver.termination_condition
    print('')
    print('--- Solver Status: {0} ---'.format(term_cond))

    if term_cond != TerminationCondition.optimal:
        done = True
    else:
        # look at the solution
        print('Optimal Obj. Value = \
            {0}'.format(value(model.obj)))
        model.y.pprint()

        # create new integer cut to exclude this solution
        N_True = [i for i in N if value(model.y[i]) > 0.5]
        N_False = [i for i in N if value(model.y[i]) < 0.5]
        expr1 = sum(model.y[i] for i in N_True)
        expr2 = sum(model.y[i] for i in N_False)
        model.integer_cuts.add( sum(model.y[i] for i in \
            N_True) \
                              - sum(model.y[i] for i in N_False) \
                              <= len(N_True)-1 )


x = range(len(objective_values))
plt.bar(x, objective_values, align='center')
plt.gca().set_xticks(x)
plt.xlabel('Solution Number')
plt.ylabel('Optimal Obj. Value')
plt.savefig('warehouse_function_cuts.pdf')
#plt.show()
```

This example generates console output that shows each of the solutions encountered. It also generates Figure 14.1 with the package `matplotlib` that shows the value of the optimal objective function for each solution obtained.

14.6.2 A Sudoku Solver

In this section, we further illustrate the power of scripting in Python with Pyomo. Specifically, we will solve a feasibility problem and show how to find all the feasible solutions to the Sudoku puzzle. We will solve the problem once, identify a feasible

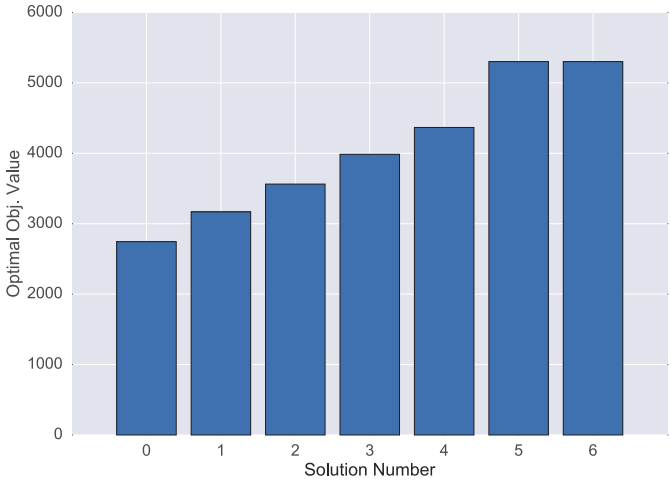


Fig. 14.1: Optimal objective value for a series of solutions obtained from the warehouse location problem.

solution, then add an integer cut to remove this solution from the list of possible solutions, and solve the problem again.

A typical Sudoku puzzle is shown in Figure 14.2. In this puzzle, one must fill

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. 14.2: An example of a Sudoku puzzle prior to solving.

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 occurrence of each number. Finally, each of the nine sub-squares 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, r, v):
   return sum(model.y[r,c,v] for c in model.COLS) == 1
model.RowCon = Constraint(model.ROWS, model.VALUES, \
    rule=_RowCon)

# exactly one nubmer in each column
def _ColCon(model, c, v):
   return sum(model.y[r,c,v] for r in model.ROWS) == 1
model.ColCon = Constraint(model.COLS, model.VALUES, \
    rule=_ColCon)
```

Defining the constraint that restricts the number for the sub-squares is a little more difficult. To make the definition easier, we define a set with an index for each of the sub-squares. Then, we define a list of tuples that describes the map from each of the sub-squares to the list of corresponding indices. This list, along with the corresponding sub-squares constraint, is defined in the complete code listing for this example at the end of this section. The desired constraint for the sub-squares 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 _SqCon(model, s, v):
   return sum(model.y[r,c,v] for (r,c) in \
       subsq_to_row_col[s]) == 1
model.SqCon = Constraint(model.SUBSQUARES, \
    model.VALUES, rule=_SqCon)
```

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, r, c):
    return sum(model.y[r,c,v] for v in model.VALUES) == 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 from a starting puzzle as well as a list of integer cuts. However, such a function would be inefficient for our purposes since we would be creating an entirely new model each time we wanted to add a single new integer cut after each solve. Thus, we will define two separate functions: one that creates the initial model given a Sudoku board, and another that adds a new integer cut to the given model based on the current value of its variables.

We define an integer cut using two sets. The first set S_0 consists of indices for those variables whose current solution is 0, and the second set S_1 consists of indices for those variables whose current solution is 1. Given these two sets, an integer cut constraint that would prevent such a solution from appearing again is defined by,

$$\sum_{(r,c,v) \in S_0} y[r,c,v] + \sum_{(r,c,v) \in S_1} (1 - y[r,c,v]) \geq 1.$$

The following Python code defines three functions. The first, `create_sudoku_model` creates the Pyomo model for the Sudoku problem. The second, `add_integer_cut` creates an integer cut corresponding to the current solution and adds it to the `ConstraintList` called `IntegerCuts`. The third, `print_solution` prints the current solution in the form of a Sudoku board.

```
from pyomo.environ import *

# create a standard python dict for mapping subsquares to
# the list (row,col) entries
subsq_to_row_col = dict()

subsq_to_row_col[1] = [(i,j) for i in range(1,4) for j in range(1,4)]
subsq_to_row_col[2] = [(i,j) for i in range(1,4) for j in range(4,7)]
subsq_to_row_col[3] = [(i,j) for i in range(1,4) for j in range(7,10)]

subsq_to_row_col[4] = [(i,j) for i in range(4,7) for j in range(1,4)]
subsq_to_row_col[5] = [(i,j) for i in range(4,7) for j in range(4,7)]
subsq_to_row_col[6] = [(i,j) for i in range(4,7) for j in range(7,10)]

subsq_to_row_col[7] = [(i,j) for i in range(7,10) for j in range(1,4)]
subsq_to_row_col[8] = [(i,j) for i in range(7,10) for j in range(4,7)]
subsq_to_row_col[9] = [(i,j) for i in range(7,10) for j in range(7,10)]

# creates the sudoku model for a 10x10 board, where the
# input board is a list of fixed numbers specified in
# (row, col, val) tuples.
def create_sudoku_model(board):
```

```
    model = ConcreteModel()

    # store the starting board for the model
    model.board = board

    # 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)

    # create the binary variables to define the values
    model.y = Var(model.ROWS, model.COLS, model.VALUES, within=Binary)

    # fix variables based on the current board
    for (r,c,v) in board:
       model.y[r,c,v].fix(1)

    # create the objective - this is a feasibility problem
    # so we just make it a constant
    model.obj = Objective(expr= 1.0)

    # exactly one number in each row
    def _RowCon(model, r, v):
       return sum(model.y[r,c,v] for c in model.COLS) == 1
    model.RowCon = Constraint(model.ROWS, model.VALUES, rule=_RowCon)

    # exactly one nubmer in each column
    def _ColCon(model, c, v):
       return sum(model.y[r,c,v] for r in model.ROWS) == 1
    model.ColCon = Constraint(model.COLS, model.VALUES, rule=_ColCon)

    # exactly one number in each subsquare
    def _SqCon(model, s, v):
       return sum(model.y[r,c,v] for (r,c) in subsq_to_row_col[s]) == 1
    model.SqCon = Constraint(model.SUBSQUARES, model.VALUES, rule=_SqCon)

    # exactly one number in each cell
    def _ValueCon(model, r, c):
       return sum(model.y[r,c,v] for v in model.VALUES) == 1
    model.ValueCon = Constraint(model.ROWS, model.COLS, rule=_ValueCon)

    return model

# use this function to add a new integer cut to the model.
def add_integer_cut(model):
    # add the ConstraintList to store the IntegerCuts if
    # it does not already exist
    if not hasattr(model, "IntegerCuts"):
       model.IntegerCuts = ConstraintList()

    # add the integer cut corresponding to the current
    # solution in the model
    cut_expr = 0.0
    for r in model.ROWS:
       for c in model.COLS:
          for v in model.VALUES:
             if not model.y[r,c,v].fixed:
                # check if the binary variable is on or off
                # note, it may not be exactly 1
                if value(model.y[r,c,v]) >= 0.5:
                   cut_expr += (1.0 - model.y[r,c,v])
                else:
                   cut_expr += model.y[r,c,v]
    model.IntegerCuts.add(cut_expr >= 1)
```

```
# prints the current solution stored in the model
def print_solution(model):
   for r in model.ROWS:
      print(' '.join(str(v) for c in model.COLS
                     for v in model.VALUES
                     if value(model.y[r,c,v]) >= 0.5))
```

The following code shows a script that drives the optimization process based on these three functions. 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 termination condition is no longer reported as `optimal`.

```
from pyomo.opt import (SolverFactory,
                       TerminationCondition)
from sudoku import (create_sudoku_model,
                    print_solution,
                    add_integer_cut)

# 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)]

model = create_sudoku_model(board)

solution_count = 0
while 1:

   with SolverFactory("glpk") as opt:
      results = opt.solve(model)
      if results.solver.termination_condition != \
        TerminationCondition.optimal:
         print("All board solutions have been found")
         break

   solution_count += 1

   add_integer_cut(model)

   print("Solution #%d" % (solution_count))
   print_solution(model)
```

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 14.3.

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. 14.3: Solved Sudoku puzzle.

Appendix A
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 the 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.

A.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.

A.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 3.5.2 (default, Aug 3 2016, 09:52:55)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-17)] on linux
Type "help", "copyright", "credits" or "license" for more \
    information.
>>> print ("Hello World")
Hello World
>>>
```

On Windows the `python` command can be launched from the DOS shell, and on *nux (which includes Macs) the `python` command can be launched from a bash or csh shell (or terminal). 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" function 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: Proper indentation is required for multi-line statements executed in the interactive shell.

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()
```

A.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.

A.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 in Python versions before 3, which is often not intended. Consider the following program:

```
weight = 200
Weight = 400

print("Division of %d by %d gives %f" % (weight, Weight, \
    weight / Weight))
print("Casting the operands to float gives %f" % \
    (float(weight) / float(Weight)))
print("Casting the result to float gives %f" % \
    (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(value(model.pParm)) / \
       float(value(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.

A.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.

A.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 `str` function. The Python line:

```
NameAge = "SPAM was introduced in " + str(1937)
```

assigns a string to the Python variable called `NameAge`.

A.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: 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]
```

A.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.

A.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])
```

NOTE: Lowercase `set` refers to the built-in *Python* object. Uppercase `Set` refers to the *Pyomo* component.

A.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 = {'Bob':'123-1134',}
>>> D['Alice'] = '331-9987'
>>> print(D)
{'Bob': '123-1134', 'Alice': '331-9987'}
>>> print(D.keys())
['Bob', 'Alice']
>>> print(D['Bob'])
123-1134
```

A.6 Conditionals

Python supports conditional code execution using structures like:

```
if CONDITIONAL1:
   statements
elif CONDITIONAL2:
   statements
else:
   statements
```

The `elif` and `else` statements are optional and any number of `elif` statements can be used. Each conditional code block can contain an arbitrary number of statements. 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.")
# @:all
```

A.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 sorted(D):
   if i == 'Alice':
      continue
   if i == 'John':
      print("Loop ends. Cleese alert!")
      break;
   print(i+" "+str(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 "Cleese is not 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`.

A.8 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 following example:

```
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*trunc(x/2) is not x
    # due to integer divide of x/2
    if 2*int(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*int(x/2) is not x. If x is an odd integer, then int(x/2) will truncate x/2 so two times the result will not be equal to x.

A somewhat advanced programming topic is the writing and use of function wrappers. There are multiple ways to write and use wrappers in Python, but we will now briefly introduce *decorators* because they are sometimes used in Pyomo models and scripts. Although the definition of a decorator can be complicated, the use of one is simple: an at-sign followed by the name of the decorator is placed on the line above the declaration of the function to be decorated.

Below is an example of the definition and use of a silly decorator to change 'c' to 'b' in the return values of a function.

```
# An example of a silly decorator to change 'c' to 'b'
# in the return value of a function.

def ctob_decorate(func):
  def func_wrapper(*args, **kwargs):
     retval = func(*args, **kwargs).replace('c','b')
     return retval.replace('C','B')
  return func_wrapper

@ctob_decorate
def Last_Words():
   return "Flying Circus"

print (Last_Words())
```

In the definition of the decorator, whose name is ctob_decorate, the function wrapper, whose name is func_wrapper uses a fairly standard Python mechanism for allowing arbitrary arguments. The function passed in to the formal argument called func is assumed by the wrapper to return a string (this is not checked by the wrapper). Once defined, the wrapper can then be used to decorate any number of functions. In this example, the function Last_Words is decorated, which has the effect of modifying its return value.

A.9 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 be 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:

```
class IntLocker:
   sint = None
   def __init__(self, i):
      self.set_value(i)
```

```
    def set_value(self, i):
        if type(i) is not int:
            print("Error: %d is not integer." % i)
        else:
            self.sint = i
    def pprint(self):
        print("The Int Locker has "+str(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.

A.10 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 pyomo.environ import *
```

imports the names from the `pyomo` package module called `environ`. Analogous to the `__init__` method in a python class, an `__init__.py` file can be included in a directory and any code therein is executed when that module is imported.

A.11 Python Resources

- *Python Home Page*, `http://www.python.org`.
- *Python Essential Reference*, David M. Beazley, Addison-Wesley, 2009.

Bibliography

[1] AIMMS. Home page. http://www.aimms.com, 2017.

[2] AMPL. Home page. http://www.ampl.com, 2017.

[3] Prasanth Anbalagan and Mladen Vouk. On reliability analysis of open source software - FEDORA. In *19th International Symposium on Software Reliability Engineering*, 2008.

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

[5] S. Bailey, D. Ho, D. Hobson, and SN Busenberg. Population dynamics of deer. *Mathematical Modelling*, 6(6):487–497, 1985.

[6] E. Balas. Disjunctive programming and a hierarchy of relaxations for discrete optimization problems. *SIAM J Alg Disc Math*, 6(3):466–486, 1985.

[7] John F. Bard. *Practical bilevel optimization: Algorithms and applications*. Kluwer Academic Publishers, Dordrecht, 1998.

[8] Colson Benoît, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Ann Oper Res*, 153:235–256, 2007.

[9] B.W. Bequette. *Process control: modeling, design, and simulation*. Prentice Hall, 2003.

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

[11] 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.

[12] BSD. Open Source Initiative (OSI) - the BSD license. http://www.opensource.org/licenses/bsd-license.php, 2009.

[13] C.C. Caroe and R. Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24(1–2):37–45, 1999.

[14] COIN-OR. Home page. http://www.coin-or.org, 2017.

[15] COUENNE. Home page. http://www.coin-or.org/Couenne, 2017.

[16] CPLEX. http://www.cplex.com, July 2010.

[17] Teodor Gabriel Crainic, Mike Hewitt, and Walter Rei. Scenario grouping in a progressive hedging-based meta-heuristic for stochastic network design. *Comput. Oper. Res.*, 43:90–99, March 2014.

[18] 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.

[19] G.B. Dantzig. Linear programming under uncertainty. *Management Science*, 1:197–206, 1955.

[20] Steven P. Dirkse and Michael C. Ferris. MCPLIB: A collection of nonlinear mixed-complementarity problems. *Optimization Methods and Software*, 5(4): 319–345, 1995.

[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] Michael C. Ferris and Todd S. Munson. Complementarity problems in GAMS and the path solver. *Journal of Economic Dynamics and Control*, 24(2):165–188, 2000.

[23] Michael C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. *SIAM Review*, 39(4):669–713, 1997.

[24] Michael C. Ferris, Robert Fourer, and David M. Gay. Expressing complementarity problems in an algebraic modeling language and communicating them to solvers. *SIAM J. Optimization*, 9(4):991–1009, 1999.

[25] Michael C. Ferris, Steven P. Dirkse, and A. Meeraus. Mathematical programs with equilibrium constraints: Automatic reformulation and solution via constrained optimization. In T. J. Kehoe, T. N. Srinivasan, and J. Whalley, editors, *Frontiers in Applied General Equilibrium Modeling*, pages 67–93. Cambridge University Press, 2005.

[26] Michael C. Ferris, Steven P. Dirkse, Jan-H. Jagla, and Alexander Meeraus. An extended mathematical programming framework. *Computers and Chemical Engineering*, 33(12):19731982, 2009.

[27] FLOPC++. Home page. https://projects.coin-or.org/FlopC++, 2017.

[28] José Fortuny-Amat and Bruce McCarl. A representation and economic interpretation of a two-level programming problem. *The Journal of the Operations Research Society*, 32(9):783–792, 1981.

[29] 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.

[30] D. Gade, G. Hackebeil, S.M. Ryan, J.-P. Watson, J-B Wets, and D.L. Woodruff. Obtaining lower bounds from the progressive hedging algorithm for stochastic mixed-integer programs. *Mathematical Programming, Series B*, to appear, 2016.

[31] GAMS. Home page. http://www.gams.com, 2008.

[32] H.I. Gassmann and E. Schweitzer. A comprehensive input format for stochastic linear programs. *Annals of Operations Research*, 104:89–125, 2001.

[33] D.M. Gay. Hooking your solver to ampl. *Numerical Analysis Manuscript*, pages 93–10, 1993.
[34] D.M. Gay. Writing. nl files, 2005.
[35] GLPK. GLPK: GNU linear programming toolkit. http://www.gnu.org/software/glpk, 2009.
[36] Harvey J. Greenberg. A bibliography for the development of an intelligent mathematical programming system. *ITORMS*, 1(1), 1996.
[37] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, 2nd Edition*. Chapman & Hall/CRC, 2006.
[38] Ge Guo, Gabriel Hackebeil, Sarah M Ryan, Jean-Paul Watson, and David L Woodruff. Integration of progressive hedging and dual decomposition in stochastic integer programs. *Operations Research Letters*, 43:311–316, 2015.
[39] GUROBI. Gurobi optimization. http://www.gurobi.com, July 2010.
[40] P. T. Harker and J. S. Pang. Finite-dimensional variational inequality and nonlinear complementarity problems: A survey of theory, algorithms and applications. *Mathematical Programming*, 48:161–220, 1990.
[41] William E. Hart and John D. Siirola. Modeling mathematical programs with equilibrium constraints in Pyomo. Technical Report SAND2015-5584, Sandia National Laboratories, July 2015.
[42] William E. Hart, Jean-Paul Watson, and David L. Woodruff. Pyomo: Modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3:219–260, 2011.
[43] Kjetil K. Haugen, Arne Lkketangen, and David L. Woodruff. Progressive hedging as a meta-heuristic applied to stochastic lot-sizing. *European Journal of Operational Research*, 132(1):116 – 122, 2001.
[44] 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.
[45] 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.
[46] Jing Hu, John E. Mitchell, Jong-Shi Pang, Kristin P. Bennett, and Gautam Kunapuli. On the global solution of linear programs with linear complementarity constraints. *SIAM J. Optimization*, 19(1):445–471, 2008.
[47] 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.
[48] Ipopt. Home page. https://projects.coin-or.org/Ipopt, 2017.
[49] D. Jacobson and M. Lele. A transformation technique for optimal control problems with a state variable inequality constraint. *Automatic Control, IEEE Transactions on*, 14(5):457–464, Oct 1969.
[50] 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.
[51] Joaquim J. Júdice. Algorithms for linear programming with linear complementarity constraints. *TOP*, 20(1):4–25, 2011.

[52] Peter Kall and Janos Mayer. *Stochastic Linear Programming: Models, Theory, and Computation*. Springer, 2005.
[53] Josef Kallrath. *Modeling Languages in Mathematical Optimization*. Kluwer Academic Publishers, 2004.
[54] A.J. King and S.W. Wallace. *Modelling with Stochastic Programming*. Springer, 2010.
[55] S. Lee and I. E. Grossmann. New algorithms for nonlinear generalized disjunctive programming. *Comp.Chem.Engng*, 24(9-10):2125–2141, 2000.
[56] Sven Leyffer. Complementarity constraints as nonlinear equations: Theory and numerical experience. In S. Dempe and V. Kalishnikov, editors, *Optimization with Multivalued Mappings*, pages 169–208. Springer, 2006.
[57] O. Listes and R. Dekker. A scenario aggregation based approach for determining a robust airline fleet composition. *Transportation Science*, 39:367–382, 2005.
[58] Johan Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *2004 IEEE Intl Symp on Computer Aided Control Systems Design*, 2004.
[59] 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.
[60] Z.-Q. Lou, J.-S. Pang, and D. Ralph. *Mathematical Programming with Equilibrium Constraints*. Cambridge University Press, Cambridge, UK, 1996.
[61] MacMPEC. MacMPEC: AMPL collection of MPECs. https://wiki.mcs.anl.gov/leyffer/index.php/MacMPEC, 2000.
[62] MATLAB. *User's Guide*. The MathWorks, Inc., 1992.
[63] Todd S. Munson. *Algorithms and Environments for Complementarity*. PhD thesis, University of Wisconsin, Madison, 2000.
[64] Bethany Nicholson, John D. Siirola, Jean-Paul Watson, Victor M. Zavala, and Lorenz T. Biegler. *Mathematical Programming Computation*, 2016. Manuscript submitted for publication.
[65] J. Nocedal and SJ Wright. Numerical optimization, series in operations research and financial engineering, 2006.
[66] OpenOpt. Home page. https://pypi.python.org/pypi/openopt, 2017.
[67] OptimJ. Wikipedia page. https://en.wikipedia.org/wiki/OptimJ, 2017.
[68] J. Outrata, M. Kocvara, and J. Zowe. *Nonsmooth Approach to Optimization Problems with Equilibrium Constraints*. Kluwer Academic Publishers, Dordrecht, 1998.
[69] PuLP. A python linear programming modeler. https://pythonhosted.org/PuLP/, 2017.
[70] PyGlpk. PyGlpk: A python module which encapsulates GLPK. http://www.tfinley.net/software/pyglpk, 2011.
[71] Pyipopt. Home page. https://github.com/xuy/pyipopt, 2017.

[72] pyomo-model-libraries. Models and examples for pyomo. https://github.com/Pyomo/pyomo-model-libraries, 2015.
[73] Pyomo Software. Github site. https://github.com/Pyomo, 2017.
[74] PYRO4. Python remote objects. https://pythonhosted.org/Pyro4/, 2017.
[75] R. Raman and I. E. Grossmann. Modelling and computational techniques for logic based integer programming. *Comp.Chem.Engng*, 18(7):563–578, 1994.
[76] 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.
[77] N.W. Sawaya and I. E. Grossmann. Computational implementation of nonlinear convex hull reformulation. *Comp.Chem.Engng*, 31(7):856–866, 2007.
[78] Hermann Schichl. Models and the history of modeling. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, Dordrecht, Netherlands, 2004. Kluwer Academic Publishers.
[79] 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.
[80] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczynski. *Lectures on Stochastic Programming: Modeling and Theory*. Society for Industrial and Applied Mathematics, 2009.
[81] 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.
[82] 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.
[83] TOMLAB. TOMLAB optimization environment. http://www.tomopt.com/tomlab, 2008.
[84] Stein W. Wallace and William T. Ziemba, editors. *Applications of Stochastic Programming*. Society for Industrial and Applied Mathematics, 2005.
[85] J.P. Watson and D.L. Woodruff. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science*, 8:355–370, 2010.
[86] H. Paul Williams. *Model Building in Mathematical Programming*. John Wiley & Sons, Ltd., fifth edition, 2013.
[87] D.L. Woodruff and E. Zemel. Hashing vectors for tabu search. *Annals of Operations Research*, 41(2):123–137, 1993.
[88] Guangquan Zhang, Jie Lu, Javier Montero, and Yi Zeng. Model, solution concept and Kth-best algorithm for linear trilevel programming. *Information Sciences*, 180:481–492, 2010.
[89] Ying Zhou and Joseph Davis. Open source software reliability model: An empirical approach. *ACM SIGSOFT Software Engineering Notes*, 30:1–6, 2005.

Index