М. Бинум, Г. Хакебейл, У. Харт, К. Лэрд,
Б. Николсон, Дж. Сиирола, Ж.-П. Уотсон, Д. Вудраф

Pyomo

Моделирование оптимизации на Python

ДМК ПРЕСС
издательство

Майкл Л. Бинум, Габриэль А. Хакебейл, Уильям Э. Харт,
Карл Д. Лэрд, Бетани Л. Николсон, Джон Д. Сиирола,
Жан-Поль Уотсон, Дэвид Л. Вудраф

Pyomo. Моделирование оптимизации на Python

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

Pyomo – Optimization Modeling in Python

Third Edition

Springer

Майкл Л. Бинум, Габриэль А. Хакебейл, Уильям Э. Харт,
Карл Д. Лэрд, Бетани Л. Николсон, Джон Д. Сиирола,
Жан-Поль Уотсон, Дэвид Л. Вудраф

Pyomo. Моделирование оптимизации на Python

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.04
ББК 32.372
Б62

Бинум М. Л., Хакебейл Г. А., Харт У. Э., Лэрд К. Д., Николсон Б. Л., Сиирола Д. Д., Уотсон Ж.-П., Вудраф Д. Л.
Б62 Pyomo. Моделирование оптимизации на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2023. – 230 с.: ил.

ISBN 978-5-93700-230-3

Эта книга представляет собой полное руководство по Python Optimization Modeling Objects — пакету с открытым исходным кодом, предназначенному для формулирования и решения крупномасштабных задач оптимизации. Его можно использовать как из командной строки, так и из интерактивного окружения Python, что сильно упрощает создание моделей Pyomo, применение оптимизаторов и изучение решений. Благодаря многочисленным примерам, иллюстрирующим различные способы формулирования моделей, книга прекрасно раскрывает широту средств моделирования, поддерживаемых Pyomo, и ее подходы к сложным практическим приложениям.

Издание предназначено для начинающих и опытных разработчиков моделей, в том числе студентов старших курсов и аспирантов, научных работников и инженеров-практиков.

УДК 004.04
ББК 32.372

First published in English under the title Pyomo – Optimization Modeling in Python, 3rd ed. by Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, David L. Woodruff.

This edition has been translated and published under licence from Springer Nature Switzerland AG. Springer Nature Switzerland AG takes no responsibility and shall not be made liable for the accuracy of the translation.

ISBN 978-3-030-68927-8 (англ.)
ISBN 978-5-93700-230-3 (рус.)

Посвящается пользователям Ruoto –
прошлым, настоящим и будущим

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Здесь будут фамилии тех, кто помогал изданию этой книги, прислав в издательство найденные ошибки или ссылку на подозрительные материалы.

Предисловие

В этой книге описывается инструмент математического моделирования – программа Python Optimization Modeling Objects (Pyomo). Pyomo поддерживает создание и анализ математических моделей для сложных приложений оптимизации. Обычно эта способность ассоциируется с языками алгебраического моделирования (algebraic modeling languages – AML) высокого уровня. Большинство AML реализованы как самостоятельные языки моделирования, но объекты моделирования Pyomo встроены в Python, полноценный язык программирования высокого уровня, для которого имеется обширный набор библиотек. Pyomo получил награды организации R&D100 и INFORMS Computing Society.

Моделирование – фундаментальный процесс во многих аспектах научных исследований, техники и бизнеса, а благодаря широкому распространению вычислительной техники численный анализ математических моделей стал обыденностью. Ко всему прочему основным принципом языков AML стала робастная формулировка больших моделей для сложных приложений, встречающихся на практике [37]. AML облегчили процесс формулирования моделей, упростив управление разреженными данными и добавив поддержку естественного выражения компонентов модели. Дополнительно AML типа Pyomo поддерживают написание скриптов, включающих объекты модели, что дает возможность проводить нестандартный анализ сложных задач.

Основой Pyomo является объектно ориентированное представление моделей оптимизации. Также Pyomo содержит пакеты для определения расширений и переформулирования модели. Кроме того, в состав Pyomo входят пакеты, определяющие интерфейсы с такими решателями, как CPLEX и Gurobi, и службами решения типа NEOS.

Цели этой книги

В третье издание включено переработанное описание средства моделирования Pyomo. Основная цель книги – дать общее описание Pyomo, которое позволило бы пользователям создавать и оптимизировать модели. Поэтому в книге много примеров, иллюстрирующих различные методы формулирования моделей.

Другая цель книги – проиллюстрировать богатство возможностей Pyomo, в том числе формулирование и анализ типичных моделей оптимизации, включая линейное программирование, смешанно-целочисленное линейное программирование, нелинейное программирование, смешанно-целочисленное нелинейное программирование, математическое программирова-

ние с ограничениями равновесия, ограничения и целевые функции, основанные на дифференциальных уравнениях, обобщенное дизъюнктивное программирование. Кроме того, Pyomo включает интерфейсы к различным распространенным пакетам программ оптимизации, как то CBC, CPLEX, GLPK и Gurobi. Модели Pyomo можно оптимизировать с помощью таких программ, как IPOPT, в которой используется интерфейс к библиотеке AMPL Solver Library.

Наконец, книга призвана облегчить знакомство с Pyomo даже тем пользователям, которые мало что знают о Python. Приложение A содержит краткое введение в Python, но мы были поражены тем, насколько справочники по Python полезны новым пользователям Pyomo. Хотя в Pyomo используются объекты Python, выражение моделей на Pyomo следует ясному и лаконичному синтаксису Python.

Однако в нашем обсуждении продвинутых средств моделирования Pyomo предполагается некоторое знакомство с объектно ориентированным проектированием и возможностями языка Python. Например, мы проводим различие между определением класса и его экземплярами. Мы не пытались объяснить эти средства Python в книге. Поэтому от читателя ожидается желание хотя бы немного изучить Python, если он хочет понимать и эффективно использовать продвинутые средства моделирования.

Для кого предназначена эта книга

Книга представляет собой справочное пособие для студентов, научных работников и инженеров-практиков. Структура Pyomo настолько проста, что программу можно использовать в курсах для студентов и аспирантов. Однако предполагается, что читатель знаком с основами оптимизации и математического моделирования. В книге нет глоссария, но мы рекомендуем использовать для справки глоссарий математического программирования [32].

Pyomo также является ценным инструментом для теоретиков и практиков. При разработке Pyomo много внимания было уделено способности поддерживать описание и анализ реальных приложений. А значит, производительность и надежные интерфейсы с решателями занимали не последнее место.

Кроме того, мы надеемся, что исследователи найдут в Pyomo эффективный каркас для разработки высокоуровневых средств оптимизации и анализа. Например, Pyomo лежит в основе пакета оптимизации в условиях неопределенности `mpi-sppy`, в котором используется тот факт, что объекты моделирования встроены в полнофункциональный высокоуровневый язык программирования. Это позволяет прозрачно распараллеливать подзадачи с помощью соответствующих библиотек Python. Этот механизм разработки обобщенных решателей для сложных моделей очень мощный, и мы надеемся, что его можно будет использовать в сочетании со многими другими методами оптимизации.

Благодарности

Мы ценим усилия многих людей, способствовавших выходу этого и предыдущих изданий книги. Мы благодарны Элизабет Лоев из издательства Springer, которая сопровождала книгу от идеи до последних этапов производства; ее энтузиазм заразителен. Мы также благодарим Маделинн Фарбер из Sandia National Laboratories за помощь в юридических нюансах выпуска программного обеспечения с открытым исходным кодом и публикации книги. Наконец, мы признательны Дугу Проуту за разработку логотипов Pyomo, PySP, Pyomo.DAE и Coopr.

Мы в долгу перед всеми, кто тратил свое время и силы на рецензирование книги. Не будь их, книга содержала бы много опечаток и программных ошибок. Итак, спасибо Джеку Инголлу, Зеву Фридману, Харви Гринбергу, Шону Леггу, Анжелике Вонг, Дэниэлю Уорду, Деанне Гарсиа, Эллис Озакиол и Флориану Мадеру. Отдельное спасибо Эмбер Грей-Фенне и Рэнди Бросту.

Особенно мы благодарны растущему сообществу пользователей Pyomo. Ваш интерес к Pyomo и ваш энтузиазм стали самым важным фактором, повлиявшим на наше решение написать эту книгу. Мы благодарим первых приверженцев Pyomo, которые поделились с нами подробными отзывами о структуре и полезности программы, в том числе Фернандо Бадилла, Стивена Чена, Неда Дмитрова. ЮэЮэ Фана, Эрика Хонга, Аллена Холдера, Андреса Ироуме, Дэррила Меландера, Кэрол Мейерс, Пьера Нансель-Пенарда, Мехула Рангвала, Еву Уормингхаус и Дэвида Алдерсона. Ваши отзывы продолжают оказывать важное влияние на дизайн и возможности Pyomo.

Мы также благодарны нашим друзьям из проекта COIN-OR за поддержку Pyomo. Хотя разработка Pyomo ведется в основном на GitHub, наше партнерство с COIN-OR – ключевая часть нашей стратегии, благодаря которой Pyomo остается жизнеспособным проектом с открытым исходным кодом.

Особая благодарность – нашим коллегам, создававшим пакеты для Pyomo: Франсиско Муньосу, Тимоти Эклу, Кэвину Хантеру, Патрику Стилу и Дэниэлю Уорду. Мы также признательны Тому Браунстейну, Дэйву Гэю и Нику Беневидасу за помощь в разработке модулей Python документации по Pyomo.

Авторы выражают благодарность за поддержку, оказанную при создании этой книги: Национальному научному фонду (гранты CBET#0941313 и CBET#0955205), Управлению передовых научных исследований в области вычислений при отделении по науке Министерства энергетики США, проекту ARPA-E Министерства энергетики США в рамках программы интеграции зеленых электрических сетей, Институту проектирования передовых энергетических систем (IDAES) с финансированием от отдела технических систем на основе имитационного моделирования, междисциплинарную исследовательскую программу отделения ископаемого топлива Министерства энергетики США, программу моделирования передовых сетей электроснабжения (AGM) Министерства энергетики США и Лабораторию целенаправленных исследований и разработок в Sandia National Laboratories.

И наконец, мы благодарны своим семьям и друзьям за их терпеливое отношение к нашей страсти к программам оптимизации.

Замечания и вопросы

В этой книге документирована версия Pyomo 6.0. Дополнительную информацию, включая опечатки, смотрите на сайте Pyomo: http://www.pyomo.org.

Исходный код Pyomo размещен на GitHub, а примеры, приведенные в этой книге, находятся в каталоге `pyomo/examples/doc/pyomobook`: https://github.com/Pyomo/pyomo.

На многие вопросы, касающиеся Pyomo, есть ответы на сайте Stack Overflow: https://stackoverflow.com/.

Мы приветствуем отзывы читателей. Направляйте их непосредственно авторам или на форум Pyomo: pyomo-forum@googlegroups.com.

Удачи!

Альбукерк, Нью-Мексико, США	*Майкл Бинум*
Энн Арбор, Мичиган, США	*Гэйб Хакебейл*
Альбукерк, Нью-Мексико, США	*Уильям Харт*
Альбукерк, Нью-Мексико, США	*Карл Лэрд*
Альбукерк, Нью-Мексико, США	*Бетани Николсон*
Альбукерк, Нью-Мексико, США	*Джон Сиирола*
Ливермор, Калифорния, США	*Жан-Поль Уотсон*
Дэвис, Калифорния, США	*Дэвид Л. Вудраф*

5 января 2021 г.

Глава 1

Введение

В этой главе мы познакомимся с Pyomo, инструментом для моделирования и решения задач оптимизации на основе Python, и расскажем, зачем он создавался. Моделирование – фундаментальный процесс во многих аспектах научных исследований, техники и бизнеса. Языки алгебраического моделирования типа Pyomo – это высокоуровневые языки для формулирования и решения математических задач оптимизации. Pyomo – гибкий и расширяемый каркас моделирования, который вбирает в себя и обобщает главные идеи языков алгебраического моделирования, организуя их в контексте широко распространенного языка программирования.

1.1. Языки моделирования для оптимизации

В этой книге описывается инструмент математического моделирования: программный пакет Python Optimization Modeling Objects (Pyomo). Pyomo поддерживает формулирование и анализ математических моделей для сложных задач оптимизации. Обычно эта возможность ассоциируется с коммерческими языками алгебраического моделирования (AML), такими как AIMMS [1], AMPL [2] и GAMS [22]. Pyomo реализует развитый набор средств моделирования и анализа и предоставляет доступ к этим средствам из полнофункционального высокоуровневого языка программирования Python, для которого написано множество библиотек.

Модели оптимизации определяют целевые функции для рассматриваемой системы. Модели можно использовать для исследования компромиссов между различными целевыми функциями, нахождения экстремальных состояний и худших случаев, а также идентификации основных факторов, от которых зависит поведение системы. Поэтому модели оптимизации применяются для анализа широкого круга научных, деловых и технических задач.

Благодаря широкой доступности вычислительных ресурсов численный анализ моделей оптимизации стал обыденным явлением. Для вычислительного анализа модели оптимизации необходимо описать модель и передать ее программе-решателю. В отсутствие языка спецификации таких моделей процесс написания входных файлов, выполнения решателя и получения ре-

зультатов от него оказывается утомительным и чреват ошибками. На это накладывается тот факт, что реальные приложения велики и сложны, так что отлаживать их трудно. Ко всему прочему для решателей имеется много разных форматов ввода, но лишь малая их часть стандартизирована. Таким образом, применение нескольких решателей для анализа одной модели оптимизации влечет за собой дополнительные сложности. Кроме того, верифицировать модель (т. е. проверить, что переданная решателю модель точно выражает намерения разработчика) исключительно трудно, не имея высокоуровневого языка для выражения модели.

AML – это высокоуровневые языки для описания и решения задач оптимизации [26, 37]. Они сводят к минимуму трудности, связанные с анализом моделей, благодаря высокоуровневой спецификации модели. Кроме того, AML предоставляют интерфейсы к внешним программам-решателям, которые используются для анализа задач, и позволяют пользователю взаимодействовать с результатами решателя в контексте этой спецификации.

Специализированные AML, такие как AIMMS [1], AMPL [2, 21] и GAMS [22], представляют собой языки описания модели оптимизации с интуитивно понятным и лаконичным синтаксисом для определения переменных, ограничений и целевых функций. Кроме того, они поддерживают абстрактные понятия: разреженные множества, индексы и алгебраические выражения, без которых не обойтись при описании крупномасштабных реальных задач с тысячами или миллионами ограничений и переменных. Эти AML могут представлять самые разные модели оптимизации и реализуют интерфейсы с различными решателями. В последнее время AML стали поддерживать новые скриптовые возможности, которые позволяют выражать высокоуровневые алгоритмы анализа вместе со спецификацией модели оптимизации.

Альтернативная стратегия – использовать AML, который расширяет какой-нибудь стандартный высокоуровневый язык программирования (в противоположность специализированному проприетарному языку) для формулирования моделей оптимизации, которые затем анализируются решателями, написанными на низкоуровневых языках. Такой двухъязыковой подход сочетает гибкость высокоуровневого языка для формулирования задач оптимизации и эффективность низкоуровневого для численных расчетов. Этот подход завоевывает все большую популярность в программном обеспечении для научных расчетов. Среда оптимизации TOMLAB, написанная на Matlab [57], – один из наиболее зрелых пакетов, в которых он применяется; в Pyomo также используется этот подход. Есть еще примеры расширения стандартных языков программирования типа Java и C++ путем включения AML-конструкций. Так, библиотеки моделирования FlopC++ [19], OptimJ [47] и JuMP [13] поддерживают описание моделей оптимизации с использованием объектно ориентированного проектирования на C++, Java и Julia соответственно. Эти библиотеки частично приносят в жертву интуитивно понятный математический синтаксис специализированного AML, зато позволяют воспользоваться всей гибкостью современных языков программирования высокого уровня. Их дополнительное преимущество заключается в том, что они непосредственно компонуются с высокопроизводительными библио-

теками оптимизации и решателями, что для некоторых приложений может быть существенно.

1.2. Моделирование на Pyomo

Цель Pyomo – предоставить платформу для описания моделей оптимизации, которая воплощает идеи современных AML в контексте, обеспечивающем гибкость, расширяемость, переносимость, открытость и удобство сопровождения. Pyomo – это AML, расширяющий Python путем включения объектов для моделирования оптимизации [30]. Эти объекты можно использовать для задания моделей оптимизации и их преобразования в различные форматы, понятные внешним решателям.

Теперь мы приведем несколько примеров, иллюстрирующих использование Pyomo для описания моделей оптимизации.

1.2.1. Простые примеры

Рассмотрим следующую линейную программу (ЛП):

$$\begin{aligned} & \min\ x_1 + 2x_2 \\ \text{при условиях}\ & 3x_1 + 4x_2 \geqslant 1 \\ & 2x_1 + 5x_2 \geqslant 2 \\ & x_1, x_2 \geqslant 0 \end{aligned}$$

Эту ЛП легко выразить на Pyomo следующим образом:

```
import pyomo.environ as pyo

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

Первая строка – стандартное предложение импорта в Python, которое инициализирует окружение Pyomo и загружает библиотеку базовых компонентов моделирования. В следующей строке конструируется объект модели и определяются его атрибуты. В этом примере описывается *конкретная модель*. Компоненты модели – объекты, являющиеся атрибутами объекта модели, а объект `ConcreteModel` инициализирует каждый компонент модели при добавлении. Переменные, ограничения и целевая функция модели определяются с помощью *компонентов модели*.

Редко бывает так, что создается единственный экземпляр некоторой модели оптимизации, подлежащей решению. Чаще имеется общая модель оп-

тимизации, а затем создается конкретный экземпляр этой модели с определенными данными. Например, следующие уравнения представляют ЛП со скалярными параметрами n и m, векторными параметрами b и c и матричным параметром a:

$$\begin{aligned} &\min \sum_{i=1}^{n} c_i x_i \\ \text{при условиях } &\sum_{i=1}^{n} a_{ji} x_i \geqslant b_j \qquad \forall j = 1 \ldots m \\ &x_i \geqslant 0 \qquad\qquad\quad \forall i = 1 \ldots n \end{aligned}$$

Эту ЛП можно выразить в виде конкретной модели на Pyomo следующим образом:

```
import pyomo.environ as pyo
import mydata

model = pyo.ConcreteModel()

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

def obj_rule(model):
    return sum(mydata.c[i]*model.x[i] for i in mydata.N)
model.obj = pyo.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 = pyo.Constraint(mydata.M, rule=con_rule)
```

Этот скрипт требует, чтобы при конструировании каждого компонента модели были доступны соответствующие данные. В нашем случае необходимые данные находятся в файле `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}
```

Эту ЛП можно также рассматривать как абстрактную математическую модель, в которой неспецифицированные символические значения параметров определяются позже, на этапе инициализации модели. Например, эту ЛП можно следующим образом выразить в виде абстрактной модели на Pyomo:

```
import pyomo.environ as pyo

model = pyo.AbstractModel()

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

```
model.c = pyo.Param(model.N)
model.a = pyo.Param(model.M, model.N)
model.b = pyo.Param(model.M)
model.x = pyo.Var(model.N, within=pyo.NonNegativeReals)

def obj_rule(model):
    return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = pyo.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 = pyo.Constraint(model.M, rule=con_rule)
```

Этот пример включает компоненты модели, которые предоставляют абстрактные, или символические, определения множества и значений параметров. Объект `AbstractModel` откладывает инициализацию компонентов модели до момента создания *экземпляра модели*, когда передаются заданные пользователем данные множества и параметры. И конкретные, и абстрактные модели можно инициализировать данными из различных источников, включая файлы команд данных, например:

```
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. Пример раскраски графа

Мы продолжим иллюстрировать возможности моделирования Pyomo на примере простой хорошо известной задачи оптимизации: минимальной раскраски графа (известной также как раскраска вершин). В задаче о раскраске графа требуется назначить цвета вершинам графа, так чтобы никакие две смежные вершины не были окрашены в один цвет. У раскраски графов есть много практических применений, включая распределение регистров в компиляторах, планирование ресурсов и сопоставление с образцами, а кроме того, она лежит в основе развлекательных головоломок типа судоку.

Обозначим $G = (V, E)$ граф с множеством вершин V и множеством ребер $E \subseteq V \times V$. Для данного G целью в задаче минимальной раскраски графа является нахождение допустимой раскраски минимальным числом цветов. Для

простоты предположим, что ребра из множества E упорядочены таким образом, что если $(v_1, v_2) \in E$, то $v_1 < v_2$. Обозначим k максимальное число цветов и определим множество возможных цветов $C = \{1, \ldots, k\}$.

Задачу минимальной раскраски графа можно представить в виде следующей целочисленной программы (ЦП):

$$\begin{aligned} & \min y \\ \text{при условиях } & \sum_{c \in C} x_{v,c} = 1 && \forall v \in V \\ & x_{v_1,c} + x_{v_1,c} \leqslant 1 && \forall (v_1, v_2) \in E \\ & y \geqslant c \cdot x_{v,c} && \forall v \in V, c \in C \\ & x_{v,c} \in \{0, 1\} && \forall v \in V, c \in C \end{aligned} \tag{1.1}$$

В этой формулировке переменная $x_{v,c}$ равна 1, если вершина v окрашена цветом c, и 0 в противном случае, а y – число использованных цветов. Первое ограничение требует, чтобы каждая вершина была окрашена ровно одним цветом. Второе ограничение означает, что вершины, соединенные ребром, должны быть окрашены разными цветами. Третье ограничение определяет нижнюю границу y и гарантирует, что y не меньше числа цветов, использованных для раскраски. Четвертое, и последнее, ограничение означает, что $x_{v,c}$ могут принимать только два значения.

На рис. 1.1 приведена формулировка описанной задачи раскраски графа в Pyomo с использованием конкретной модели; пример взят из работы Gross and Yellen [27]. Эта спецификация состоит из предложений Python, определяющих объект `ConcreteModel`, и последующего определения различных атрибутов этого объекта, включая переменные, ограничения и целевую функцию оптимизации. В строках 10–24 определены данные модели. Строка 28 – стандартное предложение импорта Python, которое вносит все символы (например, классы и функции), определенные в `pyomo.environ`, в текущее пространство имен. В строке 31 создается объект модели – экземпляр класса `ConcreteModel`. В строках 34 и 35 определены переменные модели. Отметим, что `y` – скалярная переменная, а `x` – двумерный массив переменных. В остальных строках определяются ограничения и целевая функция модели. Класс `Objective` определяет единственную целевую функцию с помощью именованного параметра `expr`. Класс `ConstraintList` представляет список ограничений, которые добавляются по одному.

По сравнению со специализированными AML, модели Pyomo, очевидно, более многословны (см., например, Hart et al. [30]). Однако этот пример иллюстрирует, что синтаксис Python все же позволяет выразить математические идеи лаконично и интуитивно понятно. Оставляя в стороне классы Pyomo, в этом примере нет ничего, кроме стандартного синтаксиса и методов Python. Например, в строке 4 используются генераторы Python для обхода всех элементов множества `colors` и применения к ним функции `sum`. В Pyomo имеются некоторые служебные функции, позволяющие упростить конструирование выражений, но никаких хитроумных расширений базовой функциональности Python не требуется.

```
#
# Пример раскраски графа, написанный на основе программы из книги
#
#  Jonathan L. Gross and Jay Yellen,
#  "Graph Theory and Its Applications, 2nd Edition",
#  Chapman & Hall/CRC, Boca Raon, FL, 2006.
#

# Определить данные графа
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)


# Предложение импорта Python
import pyomo.environ as pyo

# Создать объект модели Pyomo
model = pyo.ConcreteModel()

# Определить переменные модели
model.x = pyo.Var(vertices, colors, within=pyo.Binary)
model.y=pyo.Var()

# Каждая вершина окрашивается одним цветом
model.nodecoloring = pyo.ConstraintList()
for v in vertices:
    model.node_coloring.add(
                sum(model.x[v,c] for c in colors) == 1)

# Вершины, соединенные ребром, нельзя окрашивать одним цветом
model.edge_coloring = pyo.ConstraintList()
for v,w in edges:
for c in colors:
    model.edge_coloring.add(
                model.x[v,c] + model.x[w,c] <= 1)

# Задать нижнюю границу минимального числа потребных
# цветов
model.min_coloring = pyo.ConstraintList()
for v in vertices:
    for c in colors:
        model.mincoloring.add(
                model.y>=c_model.x[v,c])

# Минимизировать число потребных цветов
model.obj = pyo.Objective(expr=model.y)
```

Рис. 1.1 ❖ Конкретная модель Pyomo для задачи минимальной раскраски графа

1.2.3. Ключевые особенности Pyomo

Python

Понятный синтаксис Python позволяет Pyomo выражать математические идеи интуитивно понятно и лаконично. Кроме того, выразительные средства программирования Python можно использовать для формулирования сложных моделей и определения высокоуровневых решателей, обращающихся к высокопроизводительным библиотекам оптимизации. Python предоставляет развитые возможности написания скриптов и позволяет пользователям анализировать модели и решения Pyomo, привлекая весь потенциал сторонних библиотек (например, numpy, scipy и matplotlib). Наконец, поскольку Pyomo погружена в Python, пользователи могут изучать ее базовый синтаксис, пользуясь обширной документацией по Python.

Настраиваемые возможности

Pyomo спроектирована с учетом модели разработки по принципу «каши из топора», когда каждый разработчик делает то, что ему хочется. Ключевым элементом такого дизайна является каркас на основе плагинов, который Pyomo использует для интегрирования компонентов модели, преобразований модели, решателей и диспетчеров решателей. Каркас управляет регистрацией этих возможностей. Таким образом, пользователи могут настраивать Pyomo модульным образом, не опасаясь дестабилизировать базовую функциональность.

Командные инструменты и скрипты

Модели Pyomo можно анализировать с помощью командных утилит или скриптов на Python. Командная утилита `pyomo` предоставляет обобщенный интерфейс к большинству возможностей моделирования Pyomo и реализует обобщенный процесс оптимизации. Этот процесс можно легко реализовать Python-скриптом и настроить под конкретные потребности пользователя.

Определение конкретных и абстрактных моделей

Примеры в разделе 1.2.1 иллюстрируют поддержку конкретных и абстрактных моделей Pyomo. Разница между этими подходами к моделированию заключается в том, когда именно инициализируются компоненты модели: в конкретных моделях компоненты инициализируются сразу, а в абстрактных инициализация откладывается до момента инициализации модели. Следовательно, оба подхода эквивалентны, а выбор зависит от контекста, в котором используется Pyomo, и от вкусов пользователя. Модели обоих типов легко инициализировать данными из самых разных источников (например, файлов в форматах csv, json, yaml, excel и баз данных).

Объектно ориентированный дизайн

В Pyomo применяется объектно ориентированный подход к проектированию библиотек. Модель – это объект Python, а компоненты модели – атрибуты этого объекта. Такой дизайн позволяет Pyomo автоматически управлять именованием компонентов модели и естественно разделяет компоненты объектов разных моделей. Структуру моделей Pyomo можно уточнять с помощью блоков, т. е. поддерживается иерархическая вложенность компонентов модели. Многие продвинутые средства моделирования Pyomo основаны на таком структурном моделировании.

Выразительные возможности моделирования

Компоненты моделей Pyomo можно использовать для формулирования широкого круга задач оптимизации, в том числе:

- линейные программы;
- квадратичные программы;
- нелинейные программы;
- смешанно-целочисленные линейные программы;
- смешанно-целочисленные квадратичные программы;
- обобщенные дизъюнктивные программы;
- смешанно-целочисленные стохастические программы;
- динамические задачи с дифференциальными алгебраическими уравнениями;
- математические программы с ограничениями равновесия.

Интеграция с решателями

Pyomo поддерживает тесно и слабо связанные интерфейсы с решателями. Тесно связанные инструменты моделирования обращаются к библиотечным решателям задач оптимизации непосредственно (например, путем статической или динамической компоновки), а слабо связанные вызывают внешние исполняемые файлы (например, посредством системных вызовов). Многие решатели задач оптимизации читают данные задач в хорошо известных форматах (например, в формате AMPL `nl` [24]); они слабо связаны с Pyomo. Решатели, имеющие интерфейсы к Python (например, Gurobi и CPLEX), могут быть тесно связаны, что позволяет избежать записи данных во внешние файлы.

Открытый исходный код

Pyomo разрабатывается как проект с открытым исходным кодом, чтобы сделать процесс проектирования и реализации ПО максимально прозрачным. Pyomo распространяется на условиях лицензии BSD [8], которая налагает меньше ограничений на использование в коммерческих или государственных организациях. Управление исходным кодом Pyomo осуществляется с по-

мощью GitHub [53] и проекта COIN-OR [9]. Списки рассылки для разработчиков и пользователей ведутся с помощью Google Groups. Растет количество свидетельств в пользу того, что надежность ПО с открытым исходным кодом не уступает надежности закрытого коммерческого ПО [3, 59], и при разработке Pyomo ведется тщательный контроль для обеспечения стабильности и надежности программы.

1.3. Подготовительные действия

Для выполнения примеров, приведенных в книге, необходимо установить следующее программное обеспечение:

- Python версии 3.6 или старше (хотя почти все примеры работают и с более ранними версиями Python). В настоящее время Pyomo опирается на CPython; для Jython и PyPy поддерживается лишь часть функциональности;
- Pyomo 6.0, именно эта версия используется в книге;
- решатель `GLPK` [25], который используется для генерирования вывода большинства примеров. Можно использовать и другие решатели для задач линейного и смешанно-целочисленного линейного программирования, но `GLPK` легко устанавливается и широко доступен;
- решатель `IPOPT` [34], который используется для генерирования вывода в примерах нелинейных моделей. Можно использовать и другие нелинейные оптимизаторы, если они собраны с библиотекой AMPL Solver Library [23];
- решатель `CPLEX` [11], который используется для генерирования вывода в примерах стохастического программирования. Этот коммерческий решатель предоставляет возможности, которые отсутствуют в решателях задач оптимизации с открытым исходным кодом (например, оптимизацию квадратичных целочисленных программ);
- Python-пакет `matplotlib` для построения графиков.

Инструкции по установке Pyomo имеются на сайте Pyomo по адресу www.pyomo.org. В приложении А приведено краткое пособие по языку программирования Python; в различных онлайновых источниках можно найти более полные пособия и документацию.

1.4. Краткий обзор книги

Оставшийся текст книги разделен на три части. Первая часть – введение в Pyomo. Главу 2 можно рассматривать как очень краткий курс оптимизации и математического моделирования, включающий примеры применения Pyomo к формулированию и решению алгебраических моделей оптимизации. Глава 3 иллюстрирует возможности моделирования Pyomo на примере

простых конкретных и абстрактных моделей, а в главе 4 описываются базовые компоненты моделирования Pyomo. Основы вложения моделей Pyomo в скрипты на Python излагаются в главе 5. И завершается первая часть главой 6, в которой описывается взаимодействие с решателями.

Во второй части книги документированы продвинутые возможности и расширения. В главе 7 описываются средства нелинейного программирования Pyomo, а в главе 8 – построение иерархических моделей. Глава 9 содержит рекомендации по повышению производительности. В главе 10 описан класс `AbstractModel`, синтаксис командных файлов данных и командный интерфейс к Pyomo.

Третья часть книги посвящена расширениям моделирования. В главе 11 приведен обзор обобщенного дизъюнктивного программирования. В главе 12 описаны динамические модели, выражаемые дифференциальными и алгебраическими уравнениями, а в главе 13 – программы с ограничениями равновесия.

> **ПРИМЕЧАНИЕ** Эта книга не является полным справочным руководством по Pyomo. Наша цель – обсудить базовую функциональность, имеющуюся в версии Pyomo 6.0.

1.5. Обсуждение

Многие разработчики пришли к выводу, что понятный синтаксис Python и богатый набор библиотек – отличная почва для моделирования оптимизации [30]. Средства моделирования реализованы в различных пакетах программ на Python, например PuLP [49], APLEpy [4] и OpenOpt [46]. Кроме того, существует немало пакетов, реализующих интерфейсы с решателями, в т. ч. такие пакеты с открытым исходным кодом, как PyGlpk [50] и pyipopt [51], в дополнение к Python-интерфейсам таких коммерческих решателей, как CPLEX [11] и Gurobi [28].

У Pyomo есть несколько отличительных особенностей. Во-первых, Pyomo предоставляет механизмы для расширения базовой функциональности моделирования и оптимизации без внесения изменений в код самой Pyomo. Во-вторых, Pyomo поддерживает определение конкретных и абстрактных моделей. Это дает пользователю большую гибкость при решении вопроса о том, насколько тесно данные должны быть интегрированы с определением модели. Наконец, Pyomo поддерживает широкий класс моделей оптимизации, включая стандартные линейные программы, общие нелинейные модели, обобщенные дизъюнктивные программы, задачи, описываемые дифференциальными уравнениями, и математические программы с условиями равновесия.

Часть I

ВВЕДЕНИЕ В PYOMO

Глава 2

Математическое моделирование и оптимизация

В этой главе объясняются начала оптимизации и математического моделирования. Вы не найдете здесь полного описания этих тем, а лишь информацию, необходимую для чтения книги. Более подробное обсуждение методов оптимизации см. в работе Williams [58]. Приведены реализации простых моделей, чтобы читатель получил первое представление о том, как используется Pyomo.

2.1. Математическое моделирование

2.1.1. Общие сведения

Моделирование – фундаментальный процесс во многих аспектах научных исследований, техники и бизнеса. Под моделированием понимается создание упрощенного представления системы или объекта материального мира. Такое упрощение позволяет структурировать знания об исходной системе, чтобы проанализировать результирующую модель. Шихль (Schichl [56]) отмечает, что модели используются для следующих целей:

- **объяснить явления**, возникающие в системе;
- **предсказать** будущие состояния системы;
- **оценить ключевые факторы**, влияющие на происходящее в системе;
- **идентифицировать экстремальные состояния** системы, возможно, представляющие худшие случаи или планы с минимальной стоимостью;
- **проанализировать компромиссы** и предложить их лицам, принимающим решения.

Кроме того, структура представления модели упрощает передачу ассоциированных с моделью знаний. Например, одним из ключевых аспектов модели является уровень детализации, который отражает знания о системе, необходимые для использования модели в приложении.

Математика всегда играла важнейшую роль в представлении и формулировании наших знаний. Математическое моделирование становилось все более формальным по мере разработки новых каркасов для описания сложных систем. В современном моделировании центральными являются следующие математические понятия:

- **переменные**: представляют *неизвестные* или изменяющиеся части модели (например, принимаемые решения или характеристики выхода системы);
- **параметры**: символические представления реальных данных, могут варьироваться для различных экземпляров задачи или сценариев;
- **отношения**: *равенства*, *неравенства* и другие математические связи, описывающие, как различные части модели соотносятся между собой.

Модели оптимизации – это математические модели, в которых функции представляют цели, ради достижения которых моделируется система. Модели оптимизации можно анализировать, исследуя различные компромиссы, позволяющие находить решения, оптимизирующие целевые функции системы. Следовательно, эти модели можно использовать в разнообразных научных, технических и деловых приложениях.

2.1.2. Пример моделирования

Модель в том смысле, в котором мы употребляем это слово, представляет объекты, абстрагируя некоторые свойства. Все мы знакомы с физическими моделями, например игрушечными железными дорогами или моделями автомобилей. Но нас интересуют математические модели, в которых для представления аспектов системы или объекта материального мира используются символы.

Например, человеку интересно определить, сколько шариков мороженого лучше всего купить. Обозначим символом x число шариков, а символом c – цену одного шарика. Тогда полную стоимость можно смоделировать как произведение c на x, обычно обозначаемое cx.

Может понадобиться более сложная модель, если имеются скидки за объем заказа или наценки за неполные шарики. Скорее всего, эта модель не годится для отрицательных x. Редко удается продать шарик мороженого за ту же цену, за которую купил.

Труднее математически смоделировать счастье от покупки шариков мороженого в вафельном рожке. Один из возможных подходов – измерять счастье пропорционально. Определим единицу счастья, ассоциированную с одним шариком, и обозначим ее h. Тогда можно смоделировать счастье от покупки x шариков как произведение h на x, или hx. Для кого-то такая модель является неплохим приближением для значений x от ½ до 3, но вряд ли кто-то станет

в сто раз счастливее, получив в рожке 100 шариков мороженого, когда ему достаточно одного. Для некоторых людей модель счастья для значений x от 0 до 10 может выглядеть так:

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

Заметим, что в этой модели счастье становится отрицательным, когда в рожке больше 25 шариков, поэтому не для всех она хороша.

Обычно хотят моделировать сразу несколько вещей. Например, к шарикам мороженого могут прилагаться орешки. Поскольку можно купить несколько вкусностей, купленное представляется вектором x (т. е. символ x теперь представляет список). *Элементы* списка обозначаются x_i, где i – индекс элемента. Например, если мы примем, что первый элемент – число шариков мороженого, то на это число можно ссылаться с помощью x_1. При большем числе измерений в качестве индекса используется *кортеж*, обозначаемый i, j или (i, j).

Пусть c – вектор цен с такими же индексами, как у x (т. е. c_1 – цена одного шарика мороженого, а c_2 – цена одной порции орешков). Тогда полная стоимость мороженого и орешков равна

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

И эта модель стоимости, вероятно, годится не для всех возможных значений элементов x, но для каких-то целей она достаточно хороша.

Часто бывает полезно брать индексы из некоторого множества. Для рассмотренного выше примера можно было бы взять множество $\{1, 2\}$ и записать полную стоимость в виде

$$\sum_{i \in \{1,2\}} c_i x_i,$$

но обычно используется более абстрактное выражение вида

$$\sum_{i \in \mathcal{A}} c_i x_i,$$

где под $\mathcal{A}$ понимается множество индексов элементов векторов c и x (в нашем примере $\mathcal{A}$ будет равно $\{1, 2\}$).

Помимо суммирования по множеству индексов, нам иногда нужны условия, выполняющиеся для всех индексов. В этом случае используется запятая. Например, если мы хотим, чтобы никакой элемент x не был отрицательным числом, то можем написать:

$$x_i \geqslant 0, i \in \mathcal{A}$$

– это читается как «x i-е больше либо равно нулю для всех i, принадлежащих $\mathcal{A}$».

Ни в математике, ни в математическом моделировании нет такого закона, чтобы использовать только однобуквенные символы, например x, c или i. Никто не запрещает в качестве элементов $\mathcal{A}$ использовать изображения рожка мороженого и чашки орешков, но работать с такими обозначениями было бы трудно. Множество может быть и таким: {*Scoops*, *Cups*}, но в книгах

так обычно не делается, потому что занимает слишком много места и пришлось бы переносить строки. Точно так же *x* можно было бы заменить словом *Quantity* или чем-то подобным. Длинные имена поддерживаются таким языками моделирования, как Pyomo, и, вообще говоря, использовать в моделях осмысленные имена – здравая мысль. Пробелы или знаки минуса в именах часто приводят к проблемам и путанице, поэтому обычно вместо них используются знаки подчеркивания.

2.2. Оптимизация

Символ *x* часто применяется как *переменная* при моделировании оптимизации. Иногда он называется *переменной решения*, потому что модели оптимизации и строятся для того, чтобы принимать решения. Это может запутать людей, знакомых со статистическим моделированием. Там символ *x* часто используется для обозначения данных, и компьютеру передаются значения *x*, чтобы он вычислил статистики, тогда как в моделях оптимизации компьютеру передаются другие данные, а требуется вычислить хорошие значения *x*. Конечно, никто не заставляет использовать именно букву *x*, но в учебниках и вводных курсах обычно выбирают ее.

Такие значения, как стоимость (мы обозначали ее символом *c*), называются *данными*, или *параметрами*. Модель оптимизации можно описать, не определяя значений параметров, но в конкретном оптимизируемом экземпляре значения данных должны быть определены, иногда мы называем их *данными экземпляра*.

Чтобы можно было произвести оптимизацию, в модели должна быть определена *целевая функция*. При оптимальных значениях переменных решения получается наилучшее возможное значение целевой функции. Важно отметить, что мы говорим «оптимальные значения» (во множественном числе), потому что часто бывает так, что наилучшее значение целевой функции достигается при нескольких наборах значений переменных. Эта функция обычно записывается в очень абстрактной форме, например $f(x)$. Что понимать под «наилучшим» значением – наибольшее или наименьшее, определяется типом оптимизации: *максимизация* или *минимизация*.

Предположим, к примеру, что *x* – не вектор, а скаляр, равный числу шариков мороженого. Если воспользоваться описанной выше моделью счастья, то

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

где *h* – данные. (На самом деле в этом конкретном примере значение *h* совершенно не важно для нахождения *x*, максимизирующего счастье.) Смоделированная нами задача оптимизации записывается в виде

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

но склонные к строгости авторы пишут так:

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

чтобы было понятно, что x – переменная решения. В данном случае существует всего одно наилучшее значение x, которое можно найти методами численной оптимизации. Это значение оказывается дробным, т. е. придется купить нецелое число шариков. Такую модель не назовешь полезной для типичной лавки мороженого, где купить можно только целое неотрицательное число шариков. Чтобы включить это требование, мы добавим в модель оптимизации *ограничение*:

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

при условии

$x \in$ множеству целых неотрицательных чисел.

Предположим, что эта модель используется не в лавке, а дома, где мороженое подает родитель пользователя модели. Если родитель готов делить шарики на части, но не подает больше двух шариков, то ограничение

$x \in$ множеству целых неотрицательных чисел

следует заменить таким:

$0 \leqslant x \leqslant 2$.

Эта модель неидеальна, потому что не все дробные значения x осмыслены.

Чтобы проиллюстрировать рассмотренные до сих пор аспекты модели, вернемся к нескольким продуктам, которые описываются множеством индексов $\mathcal{A}$, так что x – вектор. Воспользуемся следующей моделью счастья для продукта с индексом i:

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

где h и d – векторы данных с таким же множеством индексов, как у x. Пусть далее c – вектор цен, а u – вектор максимальных количеств продуктов, которые можно приобрести. Предположим временно, что для любого продукта купленное количество может быть дробным. И наконец, предположим, что бюджет ограничен величиной b. Тогда задачу оптимизации можно записать в виде:

$$\max_x \sum_{i \in \mathcal{A}} h_i \cdot (x_i - (x_i/d_i)^2 \quad (H)$$

при условии $\sum_{i \in \mathcal{A}} c_i x_i \leqslant b$

$0 \leqslant x_i \leqslant u_i,\ i \in \mathcal{A}$

Иногда последнее ограничение записывают в виде двух:

$$\max_x \sum_{i \in \mathcal{A}} h_i \cdot (x_i - (x_i/d_i)^2 \quad (H)$$

при условии $\sum_{i \in \mathcal{A}} c_i x_i \leqslant b$

$x_i \leqslant u_i,\ i \in \mathcal{A}$

$x_i \geqslant 0,\ i \in \mathcal{A}$

Принято помещать сокращенное имя модели в скобках в одной строке с целевой функцией. Имя (P) очень распространенное, но мы использовали (H) как сокращение от «happiness» (счастье). По имени (H) мы будем ссылаться на эту модель далее в данной главе, когда покажем, как реализовать ее в Pyomo и решить.

2.3. Моделирование в Pyomo

Теперь рассмотрим различные стратегии формулирования и оптимизации алгебраических моделей в Pyomo. Детальное объяснение моделей Pyomo мы отложим до главы 3, а здесь приведем только примеры использования Pyomo для модели (H).

2.3.1. Конкретная формулировка

Конкретная модель Pyomo инициализирует компоненты по мере их конструирования. Это позволяет использовать встроенные в Python структуры данных при определении экземпляра модели. Существует много способов реализовать нашу модель в виде конкретной модели Pyomo, и начнем мы с использования списков и словарей Python.

> **ПРИМЕЧАНИЕ** Понимая, что нам часто придется создавать экземпляры моделей с разными данными, мы решили написать функцию, которая принимает требуемые данные в качестве аргументов и возвращает модель Pyomo. При таком подходе мы сможем повторно использовать общую модель Pyomo с разными данными.

```
import pyomo.environ as pyo

def IC_model(A, h, d, c, b, u):

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

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

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

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

return model
```

В объявлении `budgetconstr` мы определяем ограничение непосредственно с помощью именованного аргумента `expr`, хотя можно было бы использо-

вать и правило конструирования. Пример правила конструирования показан в объявлении целевой функции. Но при желании мы могли бы и там использовать именованный аргумент `expr`.

> **ПРИМЕЧАНИЕ** В Python символ обратной косой черты в конце строки означает, что предложение продолжается на следующей строке; мы пользуемся этим приемом, принимая во внимание ограничения на длину печатной строки в книге. В данном случае без символа продолжения можно было бы обойтись, потому что строка разрывается после круглой скобки.

Существуют и более элегантные способы написать функцию `IC_Model`, которые мы рассмотрим ниже. Имея конкретные данные, мы можем написать Python-программу, которая передает эти данные функции и получает полностью инициализированную модель Pyomo. Если на компьютере установлен решатель, то эта программа может отправить ему модель и в случае успеха запросить решение. Но прежде чем переходить к этим шагам, поговорим о других способах написать функцию `IC_model`.

Заметим, что `IC_model` – обычная функция Python. Модель Pyomo можно было бы реализовать просто как Python-программу, вызывающую решатель, или в виде функции `IC_model`, которая принимает в качестве единственного аргумента словарь, а не явный список аргументов. Программист, наверное, смог бы придумать и другие, более удачные способы написать такой код на Python.

Помимо рассмотренных выше компонентов моделирования, Pyomo предлагает класс множества (`Set`) и класс параметров (`Param`), которые мы обсудим в последующих главах. Ниже определена функция `IC_model_dict`, которая принимает словарь Python и определяет ту же самую модель, используя объекты типа `Set` и `Param`.

```
import pyomo.environ as pyo

def IC_model_dict(ICD):
    # ICD - словарь, содержащий данные для задачи

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

    model.A = pyo.Set(initialize=ICD["A"])
    model.h = pyo.Param(model.A, initialize=ICD["h"])
    model.d = pyo.Param(model.A, initialize=ICD["d"])
    model.c = pyo.Param(model.A, initialize=ICD["c"])
    model.b = pyo.Param(initialize=ICD["b"])
    model.u = pyo.Param(model.A, initialize=ICD["u"])

    def xbounds_rule(model, i):
        return (0, model.u[i])
    model.x = pyo.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 = pyo.Objective(rule=obj_rule,sense=pyo.maximize)

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

    return model
```

2.4. Линейные и нелинейные модели оптимизации

2.4.1. Определение

Говорят, что выражение в модели оптимизации линейно, если оно включает только суммы переменных решения и произведения переменных решения на данные. Таким образом, линейное выражение – это непостоянная линейная функция от переменных решения. Пусть x – векторная переменная, c – вектор данных, и оба они индексированы множеством $\mathcal{A}$. И пусть 2 и 3 – элементы $\mathcal{A}$. Следующие выражения линейны:

$\sum_{i \in \mathcal{A}} c_i x_i$
$\sum_{i \in \mathcal{A}} x_i$
x_2
$c_3 x_2 + c_2 x_3$
$c_3 x_2 + c_2 x_3 + 4$

С другой стороны, следующие выражения нелинейны: x_i^2, $x_2 x_3$ и $\cos(x_2)$.

Линейные выражения часто приводят к задачам, для решения которых нужно значительно меньше вычислительных ресурсов, чем в случае нелинейных выражений. Поэтому многие авторы стремятся по возможности использовать линейные выражения, а некоторые из кожи вон лезут, чтобы не было ничего, кроме линейных выражений. Кроме того, часто ищут линейные аппроксимации нелинейных моделей, чтобы получить «достаточно хорошие» решения для исходной нелинейной модели.

Для иллюстрации рассмотрим следующую линейную аппроксимацию модели (Н), в которой целевая функция заменена такой:

$$\max_x \sum_{i \in \mathcal{A}} h_i \cdot (1 - u_i / d_i^2) x_i, \tag{2.1}$$

где u_i – новый параметр модели. Это выражение действительно линейно, потому что переменные решения только умножаются на данные и складываются. Правда, параметр d возводится в квадрат, но это не переменная решения. Числовое значение всего выражения

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

вычисляется Pyomo до передачи экземпляра задачи решателю, а решатель должен найти оптимальные значения переменных решения.

2.4.2. Линейная версия

Если мы хотим модифицировать конкретную модель, приведенную на стр. 34, используя в ней выражение (2.1), то должны изменить правило выражения целевой функции следующим образом:

```
def obj_rule(model):
    return sum(h[i]*(1 - u[i]/d[i]**2) * model.x[i] \
            for i in A)
```

2.5. Решение модели Pyomo

Pyomo предлагает методы для (1) объединения модели и данных, (2) передачи результирующего экземпляра модели решателю и (3) получения результатов для отображения и дальнейшего использования. Но Pyomo не занимается решением задач оптимизации самостоятельно. Это ответственность решателей.

2.5.1. Решатели

Pyomo можно установить вообще без решателей. Например, Pyomo может просто записывать экземпляры задач в файлы, которые затем будут переданы какому-то решателю. Так поступают, если решатель работает на другом компьютере. Но обычно решатель устанавливается вместе с Pyomo, и в большинстве примеров это предполагается.

Напомним, что целевая функция в модели (H) не является линейной функцией переменной x, а ограничение на бюджет линейно. Многие решатели умеют решать задачу с квадратичной целевой функцией и линейными ограничениями, но не все. Если единственный установленный на компьютере решатель ограничен только линейными задачами, то (H) следует аппроксимировать линейной моделью.

2.5.2. Python-скрипты

Python-скрипт запускается из командной строки или из среды разработки. Как и в случае модели, есть много способов написать скрипт и подать ему данные, мы опишем их в следующих разделах. Например, скрипт, определяющий конкретную модель из раздела 2.3.1, может выглядеть так:

```
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}

model = IC_model_linear(A, h, d, c, b, u)

opt = pyo.SolverFactory('glpk')
results = opt.solve(model) # решает и обновляет модель
pyo.assert_optimal_termination(results)

model.display()
```

Если назвать этот файл `ConcHLinScript.py`, то из командной строки он запускается следующим образом:

```
python ConcHLinScript.py
```

Первые несколько строк, в которых переменным Python присваиваются значения, выглядят немного странно, но удивляться тут нечему. Обычно данные для задач оптимизации читаются из файлов или базы данных, но здесь мы присваиваем литеральные значения, чтобы не вводить дополнительных зависимостей. В последних строках создается решатель, находится решение задачи и отображается модель с найденными значениями. Функция `assert_optimal_termination` останавливает скрипт и выводит сообщение, если решатель не смог найти оптимальное решение. Родственная ей функция `check_optimal_termination` возвращает `True`, если решатель нашел оптимальное решение, и `False` в противном случае.

Глава 3

Обзор Pyomo

В этой главе приводится обзор стратегий моделирования и возможностей Pyomo. Кратко обсуждаются базовые компоненты моделирования и некоторые особенности Pyomo (например, дискретные переменные и нелинейные модели).

3.1. Введение

Pyomo поддерживает объектно ориентированный подход к определению моделей оптимизации. Объект *модели* Pyomo содержит коллекцию компонентов модели, определяющих задачу оптимизации. В пакете Pyomo имеются компоненты, необходимые для формулирования задачи оптимизации: переменные, целевые функции и ограничения, а также другие компоненты, поддерживаемые большинством современных AML, в т. ч. множества индексов и параметры. Эти базовые компоненты моделирования определяются с помощью следующих классов Python:

`Var`	переменные оптимизации;
`Objective`	выражения, подлежащие минимизации или максимизации;
`Constraint`	выражения ограничений;
`Set`	множество данных, используемых в определении экземпляра модели;
`Param`	параметры, используемые в определении экземпляра модели.

В этой главе в общих чертах описываются эти компоненты и процедура определения и решения моделей Pyomo. Перечислим основные шаги простой процедуры моделирования.

1. Создать экземпляр модели, используя компоненты моделирования Pyomo.
2. Передать экземпляр решателю для нахождения решения.
3. Получить результаты от решателя и проанализировать их.

Pyomo поддерживает Python-скрипты общего вида, т. е. пользователь может по своему усмотрению управлять процессом решения, создавая собст-

венный технологический процесс, например решать последовательности слегка отличающихся задач или применять более сложные метаалгоритмы.

В этой главе мы на примере покажем процесс формулирования реальной модели и продемонстрируем использование компонентов моделирования, компонентов индексирования и правил конструирования. Обсуждается также написание скриптов для более сложных технологических процессов.

3.2. Задача о расположении складов

На протяжении всей этой главы мы будем использовать задачу о выборе местоположений складов. Требуется найти такие местоположения множества складов, чтобы оптимизировать транспортные затраты. Пусть N – множество потенциальных местоположений, а M – множество местоположений потребителей. Для каждого склада n стоимость доставки изделия потребителю m равна $d_{n,m}$. Цель заключается в том, чтобы определить местоположения складов, доставляющие минимум общей стоимости доставки изделий. Двоичные переменные y_n определяют, следует ли строить склад в данном месте; $y_n = 1$, если следует, и 0 в противном случае. Переменная $x_{n,m}$ определяет, какая доля спроса потребителя m удовлетворяется складом n.

Переменные x и y должны быть найдены решателем, а остальные величины – известные входные данные или параметры задачи. Эта задача – частный случай задачи о p-медиане, она обладает интересным свойством: оптимальные значения x принадлежат множеству {0, 1}, хотя они и не определены как двоичные переменные.

Полная формулировка задачи выглядит так:

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

при условии

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

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

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

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

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

Здесь целевая функция (формула WL.1) – минимизировать полную стоимость доставки изделий всем потребителям. Формула WL.2 означает, что спрос каждого потребителя полностью удовлетворен, а формула WL.3 – что склад может доставлять изделия потребителям, только если он будет построен. Неравенство WL.4 означает, что может быть построено не более P складов.

В нашем примере мы предположим, что $P = 2$, а данные о складах и потребителях следующие:

Местоположения потребителей

M = {‘Нью-Йорк’, ‘Лос-Анджелес’, ‘Чикаго’, ‘Хьюстон’}

Местоположения потенциальных складов

N = {‘Харлинген’, ‘Мемфис’, ‘Эшланд’}

И пусть стоимости $d_{n,m}$ определяются следующей таблицей:

	Нью-Йорк	**Лос-Анджелес**	**Чикаго**	**Хьюстон**
Харлинген	1956	1606	1410	330
Мемфис	1096	1792	531	567
Эшланд	485	2322	324	1236

3.3. Модели Pyomo

Pyomo поддерживает объектно ориентированный подход, при котором компоненты моделирования добавляются в модель для определения задачи оптимизации. В этом разделе мы дадим обзор наиболее распространенных компонентов моделирования и приведем полные примеры решения задачи о местоположениях складов в Pyomo.

3.3.1. Переменные, целевые функции и ограничения

В задаче оптимизации должна быть по крайней мере одна переменная и целевая функция. В большинстве задач имеются также ограничения. Классы Pyomo, реализующие эти компоненты моделирования, – `Var`, `Objective` и `Constraint`. В следующем примере показано, как они определяются:

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

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

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

Здесь мы видим три переменные (x, y и z), одну целевую функцию и два ограничения. Для каждой переменной создается экземпляр класса `Var`, и этот экземпляр добавляется в объект модели в качестве атрибута. Код `model.x=pyo.Var()` создает экземпляр класса `Var` и присваивает его атрибуту `model.x`. Объект модели знает, что в него добавляется компонент, и выполняет специальную обработку, в частности назначает экземпляру `Var` имя «x» и записывает в него ссылку на объемлющую модель.

В этом примере x объявлена просто как непрерывная переменная, но с помощью именованных аргументов можно задать другие свойства объекта `Var`.

Например, аргумент `bounds` задает верхнюю и нижнюю границы, аргумент `initialize` – начальные значения, а аргумент `within` – область определения. В нашем примере для `model.y` задана нижняя граница –2 и верхняя граница 4, для `model.z` нижняя граница равна 0, а верхней границы нет, потому что аргумент `within` говорит, что областью определения является множество неотрицательных вещественных чисел.

> **ПРИМЕЧАНИЕ** Именованные аргументы часто используются в конструкторах компонентов Pyomo для задания свойств компонентов. Более подробно о поддерживаемых именованных аргументах рассказано в главе 4.

В этом примере целевая функция определена с помощью компонента `Objective`. Выражение этой функции задается именованным аргументом `expr`. По умолчанию считается, что целевая функция должна быть минимизирована, но если решается задача максимизации, то аргумент `sense` следует задать равным `maximize`. В этом примере ограничения типа равенства и неравенства объявлены с помощью компонентов `Constraint`. Чтобы определить математические выражения для ограничения (в т. ч. логический оператор между левой и правой частями), снова используется именованный аргумент `expr`. Ограничения могут включать логические операторы `==` (равно), `<=` (меньше или равно) и `>=` (больше или равно). Детальные описания компонентов `Objective` и `Constraint` и соответствующих именованных аргументов см. в главе 4.

> **ПРИМЕЧАНИЕ** В примере выше целевая функция и ограничения определены с помощью именованного аргумента `expr`. В коротких иллюстративных примерах это удобно, но чаще эти компоненты определяются с помощью правил конструирования, как обсуждается в разделах 3.3.3 и 4.2.1.

3.3.2. Индексированные компоненты

В примере выше все компоненты моделирования *скалярные*. Это значит, что подлежащие оптимизации переменные *x*, *y* и *z* принимают только одно значение, а не являются векторами или массивами. Ограничения также скалярные, т. е. в каждом объявлении создается только одно математическое ограничение. При моделировании больших сложных приложений чаще применяются векторы переменных и ограничения, размерность и порядок индексирования которых определяются моделью данных. В Pyomo для этого предусмотрены *индексированные* компоненты.

Для иллюстрации этого понятия рассмотрим задачу о расположении складов (WL), в определении которой использовались только скалярные компоненты. Далее будет показан улучшенный подход на основе индексированных компонентов. Например, можно создать отдельные переменные *x* для каждой пары (склад, потребитель):

```
model.x_Harlingen_NYC = pyo.Var(bounds=(0,1))
model.x_Harlingen_LA = pyo.Var(bounds=(0,1))
```

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

а ограничение в формуле (WL.4) вручную расписать следующим образом:

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

Тогда все ограничения в формуле (WL.2) можно было бы записать явно:

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

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

Однако для больших наборов данных такой подход был бы слишком громоздким, а выразить то же самое можно гораздо проще с помощью *индексированных* компонентов. Сначала определим список допустимых индексов для местоположений складов и потребителей:

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

Затем, пользуясь этими данными, можно определить переменные:

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

Мы обозначаем `N` и `M` множества индексов для индексированных переменных `model.x` и `model.y`. Точнее, переменная *y* индексируется значениями из *N*, а переменная *x* – двумерный массив, индексируемый значениями из *N* и *M*. При таком объявлении к элементу *x* можно обратиться `model.x[i,j]`, где `i` и `j` – элементы множеств *N* и *M* соответственно.

> **ПРИМЕЧАНИЕ** Компоненты моделирования Pyomo могут включать сколько угодно множеств индексов в виде позиционных аргументов в объявлении, но они должны предшествовать всем именованным аргументам. Эти множества индексов определяют допустимые индексы для индивидуальных элементов компонента.

При таких объявлениях ограничение (WL.4) можно переписать в виде

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

Здесь используется имеющийся в Python синтаксис итерирования для вычисления суммы по множеству индексированных переменных. Синтаксис

спискового включения позволяет коротко записать суммирование – он означает, что члены `model.y[n]` порождаются в результате обхода множества `N`. После порождения каждого члена функция `sum` прибавляет его к сумме. Аналогично целевую функцию можно определить следующим образом:

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

где члены `d[n,m]*model.x[n,m]` порождаются в результате обхода одновременно N и M.

3.3.3. Правила конструирования

Для конструирования многих индексированных ограничений применяются правила конструирования. Рассмотрим ограничение

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

Эта математическая нотация означает, что для каждого m, принадлежащего множеству M, определено одно ограничение. Компонент `Constraint` можно объявить как ограничение, индексированное элементами этого множества. Однако необходим механизм, который передает Pyomo явные выражения для каждого элемента M. Pyomo позволяет инициализировать компоненты модели с помощью определенных пользователем функций, называемых *правилами*.

Следующий пример иллюстрирует использование правила конструирования для определения ограничения (WL.2):

```
def demand_rule(mdl, m):
    return sum(mdl.x[n,m] for n in N) == 1
model.demand = pyo.Constraint(M, rule=demand_rule)
```

В первых двух строках определяется функция Python, которая будет вызываться для порождения правильного выражения ограничения для каждого элемента `M`. А в последней строке ограничение объявляется путем создания компонента `Constraint`, который индексируется элементами множества `M`. Именованный аргумент `rule` означает, что для конструирования каждого ограничения будет вызываться функция `demand_rule`.

Первый аргумент функции `demand_rule` автоматически делается равным конструируемому объекту модели. За ним следуют аргументы, задающие индексы конкретного конструируемого ограничения. Когда Pyomo конструирует объект `Constraint`, правило конструирования вызывается по одному разу для каждого значения указанных множеств индексов.

> **ПРИМЕЧАНИЕ** Pyomo ожидает, что правило конструирования возвращает выражение для каждого значения индекса. Если для некоторой комбинации индексов ограничение не требуется, то можно вернуть специальное значение `Constraint.Skip`.

Правила конструирования можно использовать для большинства компонентов моделирования, применяя именованный аргумент `rule`, – даже если компонент не индексирован. Хотя аргументы функций, порождающих правила, одинаковы для компонентов всех типов, типы возвращаемых значений различны, как показано в таблице ниже.

Компонент	Типы значений, возвращаемых функцией, порождающей правило конструирования
`Set`	Объект множества или списка Python
`Param`	Значение, целое или с плавающей точкой
`Objective`	Выражение
`Constraint`	Выражение ограничения

3.3.4. Конкретная модель для задачи о расположении складов

Конкретную модель для задачи о расположении складов можно определить следующим образом.

```
# wl_concrete.py
# Версия ConcreteModel для задачи о расположении складов
import pyomo.environ as pyo

def create_warehouse_model(N, M, d, P):
    model = pyo.ConcreteModel(name="(WL)")

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

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

def demand_rule(mdl, m):
    return sum(mdl.x[n,m] for n in N) == 1
model.demand = pyo.Constraint(M, rule=demand_rule)

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

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

return model
```

Этот файл начинается с импорта окружения Pyomo, в котором определены классы Python, используемые при построении модели. В строке 5 определяется функция, которая будет вызываться для создания и возврата модели. Это необязательно, модель можно создать и непосредственно в скрипте Python, однако такая стратегия зачастую предпочтительнее, поскольку код построения модели проще повторно использовать с другими данными. В строке 6 создается объект `ConcreteModel` и модели присваивается имя.

В строках 8 и 9 объявляются и конструируются переменные для задачи. Объект модели имеет тип `ConcreteModel`, и после выполнения этих строк переменные *x* и *y* полностью сконструированы и их индексы известны. В строках 11 и 12 определяется правило конструирования целевой функции, а в строке 13 целевая функция объявляется и присваивается атрибуту `model.obj`. Сразу после выполнения строки 13 вызывается правило, объявленное в строках 11 и 12, и конструируется выражение для целевой функции. В последующих строках точно так же сначала объявляются правила конструирования, а затем сами объекты ограничений. Так как это конкретная модель, правила ограничений вызываются, когда Python исполняет строки 17, 21 и 25. В строке 27 функция возвращает сконструированную модель.

Определив модель, мы можем написать короткий Python-скрипт, который решает конкретный экземпляр модели и показывает решение.

```
 1 # wl_concrete_script.py
 2 # Решить экземпляр задачи о расположении складов
 3
 4 # Импортировать окружение Pyomo и модель
 5 import pyomo.environ as pyo
 6 from wl_concrete import create_warehouse_model
 7
 8 # Задать данные для этой модели (их можно было бы также
 9 # импортировать из других Python-пакетов)
10
11 N = ['Harlingen', 'Memphis', 'Ashland']
12 M = ['NYC', 'LA', 'Chicago', 'Houston']
13
14 d = {('Harlingen', 'NYC'): 1956, \
15      ('Harlingen', 'LA'): 1606, \
16      ('Harlingen', 'Chicago'): 1410, \
17      ('Harlingen', 'Houston'): 330, \
18      ('Memphis', 'NYC'): 1096, \
19      ('Memphis', 'LA'): 1792, \
20      ('Memphis', 'Chicago'): 531, \
21      ('Memphis', 'Houston'): 567, \
22      ('Ashland', 'NYC'): 485, \
23      ('Ashland', 'LA'): 2322, \
24      ('Ashland', 'Chicago'): 324, \
```

```
25      ('Ashland', 'Houston'): 1236 }
26 P = 2
27
28 # Создать модель Pyomo
29 model = create_warehouse_model(N, M, d, P)
30
31 # Создать интерфейс с решателем и решить модель
32 solver = pyo.SolverFactory('glpk')
33 res = solver.solve(model)
34 pyo.assert_optimal_termination(res)
35
36 model.y.pprint() # Напечатать оптимальные местоположения складов
```

В строке 5 импортируется окружение Pyomo, а в строке 6 – функция создания модели по данным, определенная в файле `wl_concrete.py`. В строках 11–16 определяются данные для этой задачи. В списках `N` и `M` задаются местоположения складов и потребителей соответственно. В словаре `d` определяются стоимости доставки изделий каждому потребителю с каждого склада, а в строке 26 задается величина *P*, определяющая количество складов.

В строке 29 эти структуры данных Python передаются нашей функции `create_warehouse_location`, где они используются для объявления и конструирования компонентов моделирования Pyomo: объектов `Var`, `Objective` и `Constraint`. Сконструированная модель, возвращенная функцией, присваивается переменной `model`. В строке 32 создается интерфейс с решателем `"glpk"`, который решает задачу оптимизации. В строке 33 вызывается метод `solve` для выполнения решателя, и в переменной `res` возвращается объект результатов, который затем передается функции `assert_optimal_termination` (строка 34). Если решатель сообщает, что не смог найти оптимальное решение (например, потому что он сам установлен неправильно или потому что оптимального решения не существует), то эта функция напечатает сообщение и завершит скрипт.

> **ПРИМЕЧАНИЕ** Сконструированную модель Pyomo можно распечатать методом pprint – `model.pprint()`. Он выводит сводную информацию о модели, в т. ч. выражения ограничений и целевой функции. Это может оказаться очень полезно для отладки, если модель порождает результаты, отличные от ожидаемых.

В нашем примере данные для задачи (`N`, `M`, `d`, `P`) явно определены в скрипте. В коротком примере это удобно, но на практике часто требуется гораздо больше данных, поэтому обычно они загружаются из какого-то источника (например, из файла в формате Excel или JSON).

На рис. 3.1 показаны данные для задачи о расположении складов, заданные в Microsoft Excel. Следующий скрипт загружает эти данные из электронной таблицы Excel с помощью Python-пакета Pandas, а затем выполняет тот же код, что и раньше для конструирования и решения модели и распечатки результатов.

```
# wl_excel.py: загрузка Excel-данных с помощью Pandas
import pandas
import pyomo.environ as pyo
from wl_concrete import create_warehouse_model

# читать данные из Excel с помощью Pandas
df = pandas.read_excel('wl_data.xlsx', '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 = 2

# создать модель Pyomo
model = create_warehouse_model(N, M, d, P)

# создать интерфейс с решателем и решить модель
solver = pyo.SolverFactory('glpk')
solver.solve(model)
model.y.pprint() # напечатать оптимальные местоположения складов
```

Рис. 3.1 ❖ На этом рисунке показаны данные в формате Microsoft Excel для задачи о расположении складов

3.3.5. Компоненты моделирования для множеств и параметров

Хотя данные можно задавать с помощью встроенных в Python типов, Pyomo включает также компоненты моделирования `Set` и `Param` для определения множеств индексов и параметров соответственно.

Компонент `Set` служит для объявления допустимых индексов для любого индексированного компонента. Например, в контексте задачи о расположении складов имеются два множества: в N хранятся допустимые местоположения складов, а в M – местоположения потребителей. Их легко объявить следующим образом:

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

Эти объекты `Set` можно использовать для определения индексированных переменных или ограничений:

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

Здесь объекты `Set` передаются конструктору `Var`, а не спискам Python, как в предыдущих примерах. Компонент `Set` можно инициализировать множест-

вом, списком или кортежем Python, воспользовавшись именованным аргументом `initialize`.

Объекты Pyomo `Set` можно также индексировать другими множествами. Рассмотрим следующий пример:

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

Множество `model.Toppings` является индексированным. Если в `model.PremierSundaes` передать значения `{'PBC-Banana', 'Very Berry'}`, то можно будет определить украшения (topping) для каждого из этих индексов. Например, `model.Toppings['PBC-Banana']` может содержать множество `{'Peanut Butter', 'Chocolate Fudge', 'Banana'}`, а `model.Toppings['Very Berry']` – множество `{'Strawberries', 'Raspberries', 'Blueberries', 'Crunch-berries'}`.

Компонент Pyomo `Param` можно использовать, чтобы определить значения данных для задачи. В контексте задачи о расположении складов нужно определить два элемента данных: P и $d_{n,m}$. Это делается следующим образом:

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

Здесь объявляется скалярный параметр P и индексированный параметр d. Параметр d индексирован определенными ранее множествами Pyomo, описывающими местоположения складов и потребителей. Как и в случае объекта `Set`, значения этих параметров можно задать с помощью именованного аргумента `initialize`, используя словарь Python или определив правило конструирования.

По умолчанию параметры неизменяемы, т. е. после начального задания их нельзя модифицировать. Такое поведение позволяет повысить эффективность работы с выражениями внутри Pyomo. Однако можно определить и изменяемый параметр, добавив именованный аргумент `mutable=True`. Это может оказаться полезным, если модель требуется решить несколько раз с различными значениями некоторых параметров.

Например, снова рассмотрим задачу о расположении складов. Предположим, что требуется гораздо больше данных (много потенциальных местоположений складов и потребителей). Сделав параметр P изменяемым, мы сможем легко увидеть, как изменяется оптимальная величина транспортных затрат при изменении максимального количества складов.

В коде ниже показано применение объекта `Param` для определения модели с изменяемым параметром P.

```
# wl_mutable.py: задача о расположении складов с изменяемым параметром
import pyomo.environ as pyo

def create_warehouse_model(N, M, d, P):
    model = pyo.ConcreteModel(name="(WL)")

    model.x = pyo.Var(N, M, bounds=(0,1))
    model.y = pyo.Var(N, within=pyo.Binary)
    model.P = pyo.Param(initialize=P, mutable=True)
```

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

    def demand_rule(mdl, m):
        return sum(mdl.x[n,m] for n in N) == 1
    model.demand = pyo.Constraint(M, rule=demand_rule)

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

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

    return model
```

Основное отличие – объявление объекта `model.P` типа `Param` и использование `model.P` в ограничении `num_warehouses`. Скрипт можно модифицировать, так что он будет загружать данные о расстояниях из файла Excel и в цикле решать задачу оптимизации с разными значениями изменяемого параметра `model.P`. Эта версия показана ниже.

```
# wl_mutable_excel.py: решение задачи с разными значениями P
import pandas
import pyomo.environ as pyo
from wl_mutable import create_warehouse_model

# читать данные из файла Excel с помощью Pandas
df = pandas.read_excel('wl_data.xlsx', '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 = 2

# создать модель Pyomo
model = create_warehouse_model(N, M, d, P)

# создать интерфейс с решателем
solver = pyo.SolverFactory('glpk')

# в цикле перебрать значения изменяемого параметра P
for n in range(1,10):
    model.P = n
    res = solver.solve(model)
    pyo.assert_optimal_termination(res)
    print('# warehouses:', n, 'delivery cost:', pyo.value(model.obj))
```

Пользователи Pyomo могут использовать мощные возможности Python для создания специальных технологических процессов манипуляции и оптимизации моделей. В этом разделе мы лишь слегка коснулись того, на что способны такие скрипты. Мы продолжим эту тему в главе 5.

Глава 4

Модели Pyomo и их компоненты: введение

В этой главе описываются базовые классы, используемые при определении моделей оптимизации в Pyomo. В основном обсуждаются компоненты моделирования, необходимые для объявления частей модели. Рассматриваются параметры, задаваемые при объявлении компонентов, и дается информация об основных атрибутах и методах компонентов.

4.1. Объектно ориентированный AML

Pyomo поддерживает объектно ориентированный подход к представлению математических моделей оптимизации. Сначала создается объект модели, а затем в него добавляются компоненты моделирования, описывающие различные части модели. Pyomo включает компоненты моделирования, поддерживаемые большинством современных AML: переменные, ограничения, целевые функции, множества индексов и символические параметры. В этой главе описываются компоненты моделирования Pyomo. В последующих главах мы расскажем о дополнительных компонентах, расширяющих функциональность моделей оптимизации.

В Pyomo можно создавать модели двух типов: конкретные и абстрактные. *Конкретная* модель строится «на лету» по мере объявления компонентов. Поэтому данные, ассоциированные с конкретной моделью, необходимо задавать до того, как будут объявлены ее компоненты. Для определения компонентов конкретной модели можно использовать встроенные в Python структуры данных. Для представления конкретной модели служит класс `ConcreteModel`.

Напротив, *абстрактная* модель поддерживает объявление модели целиком. Экземпляр задачи не конструируется, пока не будут сконструированы

все компоненты и переданы данные. Для создания абстрактной модели служит класс `AbstractModel`. Поскольку абстрактные модели позволяют компонентам ссылаться на данные до того, как они будут определены, они часто полагаются на такие компоненты Pyomo, как `Set` и `Param`, чтобы предоставить абстрактное определение данных, используемых для конструирования модели (хотя эти компоненты можно использовать и в конкретных моделях).

Ниже перечислены основные компоненты моделирования в Pyomo.

`Var` — Компонент `Var` используется для представления переменных, подлежащих оптимизации. Pyomo поддерживает непрерывные и дискретные переменные и включает несколько предопределенных областей определения.

`Objective` — Компонент `Objective` определяет одну или несколько функций, подлежащих оптимизации решателем. Он содержит выражение, определяющее целевую функцию, и флаг направления оптимизации (максимизировать или минимизировать).

`Constraint` — Дополнительные ограничения на переменные, подлежащие оптимизации. Компонент `Constraint` содержит выражения и оператор сравнения. Pyomo поддерживает ограничения типа равенства (==) и нестрогого неравенства (<= или >=).

`Set` — Компонент `Set` представляет коллекцию данных, которая может включать числовые (например, целые) или символические (например, строковые) элементы. Чаще всего применяется для определения допустимых индексов для других компонентов. Поддерживаются также некоторые теоретико-множественные операции.

`Param` — Компонент `Param` представляет числовые или символические значения данных для задачи оптимизации. В отличие от простых типов данных Python (например, чисел с плавающей точкой), объекты `Param` позволяют изменять значения (т. е. являются изменяемыми) и поддерживают такие возможности, как разреженное представление и значения по умолчанию.

`Expression` — Компонент `Expression` служит для создания выражений Pyomo, которые можно повторно использовать в разных частях модели. Это полезно для представления общих подвыражений в целях более эффективного использования памяти. Как и значения изменяемых параметров, выражение можно изменять между вызовами решателя.

`Suffix` — Часто возникает потребность предоставить или получить метаданные модели или компонента (например, двойственную информацию от ограничения). Для этой цели и служит компонент Pyomo `Suffix`.

В этой главе все эти компоненты описываются подробно. В Pyomo имеются и другие компоненты моделирования, некоторые из них кратко обсуждаются в конце данной главы и более детально – в следующих главах.

ПРИМЕЧАНИЕ Если явно не оговорено противное, все фрагменты кода и примеры в этой главе относятся к конкретным моделям.

4.2. Общие парадигмы компонентов

У большинства компонентов моделирования Pyomo, перечисленных выше, есть общие виды поведения. Кроме того, существуют парадигмы, присущие многим компонентам. В этом разделе мы опишем такие общие поведения.

4.2.1. Индексированные компоненты

Как показано в предыдущей главе, компоненты Pyomo можно объявлять как индивидуальные, атомарные сущности или как индексированные коллекции. Индексированные компоненты встретятся в нескольких примерах в этой главе. Рассмотрим следующую модель:

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

Компонент `c` определяет единственное ограничение в этой модели, а компонент `d` – коллекцию ограничений, индексированных элементами множества `A`. Компонент `Constraint` можно использовать для объявления как простых, так и индексированных ограничений. В общем случае компоненты можно индексировать также несколькими множествами индексов. Например, компонент `model.y` индексирован множествами `A` и `B`, на него можно ссылаться `model.y[i,j]`, где `i` – произвольный элемент из `model.A`, а `j` – произвольный элемент из `model.B` (например, `model.y[2,'Q']`).

> **ПРИМЕЧАНИЕ** Предполагается, что все позиционные аргументы конструктора компонента – множества индексов для этого компонента.

Объявление аргументов для индексированных компонентов часто бывает более сложным. Например, при объявлении одной переменной можно использовать именованный аргумент `initialize`:

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

Задавать значение именованных аргументов такого типа просто, если компонент неиндексированный. Но если компонент индексированный, то иногда требуется задавать разные значения для каждого индекса. Это можно сделать тремя способами.

- Если аргументу `initialize` передано скалярное значение, то оно присваивается всем элементам компонента независимо от индекса.

- Можно также передать словарь Python (пары индекс–значения), в котором ключами являются допустимые индексы.
- Или можно передать функцию Python, которая возвращает значение для каждого индекса. Часто такие функции называются правилами.

Все три способа продемонстрированы ниже.

```
model.A = pyo.Set(initialize=[1,2,3])
model.x = pyo.Var(model.A, initialize=3.14)
model.y = pyo.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 = pyo.Var(model.A, initialize=z_init_rule)
```

4.3. Переменные

Переменные Pyomo создаются с помощью класса `Var`, который может представлять одно значение или индексированную коллекцию значений. Переменные могут иметь начальные значения, и значение переменной может получить или задать пользователь либо решатель в процессе решения.

4.3.1. Объявления Var

В следующей строке создается неиндексированный объект `Var`:

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

Поддерживаются именованные и позиционные аргументы, в табл. 4.1 приведен перечень аргументов, которые можно передать при объявлении компонента `Var`.

Таблица 4.1. Аргументы в объявлении компонента Var

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`within` или `domain`	задает допустимые области определения или значения переменной	объект Pyomo `Set`, список Python или функция-правило
`bounds`	задает нижнюю и верхнюю границы переменной	2-кортеж или функция-правило
`initialize`	задает начальное значение переменной	скалярное значение, словарь Python или функция-правило

Область определения переменной (т. е. множество ее допустимых значений) задается с помощью ключевого именованного параметра `domain` или `within` конструктора `Var`:

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

В этом примере y может принимать только целые значения 1, 2, 3. Переменная r может принимать любое вещественное значение, а w – только два значения (т. е. 0/1 или True/False). Если область определения не задана, то по умолчанию подразумевается виртуальное множество Reals. Другие виртуальные множества, поддерживаемые Pyomo, перечислены в табл. 2.4. Отметим, что их можно использовать и в иных контекстах (например, при конструировании объектов Param).

Таблица 4.2. Предопределенные виртуальные множества в Pyomo

Any	Множество всех возможных значений, кроме None
AnyWithNone	Множество всех возможных значений
EmptySet	Множество, не содержащее ни одного значения
Reals	Множество чисел с плавающей точкой
PositiveReals	Множество положительных чисел с плавающей точкой
NonPositiveReals	Множество неположительных чисел с плавающей точкой
NegativeReals	Множество отрицательных чисел с плавающей точкой
NonNegativeReals	Множество неотрицательных чисел с плавающей точкой
PercentFraction	Множество чисел с плавающей точкой в диапазоне [0,1]
UnitInterval	То же, что PercentFraction
Integers	Множество целых чисел
PositiveIntegers	Множество положительных целых чисел
NonPositiveIntegers	Множество неположительных целых чисел
NegativeIntegers	Множество отрицательных целых чисел
NonNegativeIntegers	Множество неотрицательных целых чисел
Boolean	Множество булевых значений, которые можно представить в виде False/True, 0/1, 'False'/'True' и 'F'/'T'
Binary	То же, что Boolean

Аргумент domain или within может также принимать в качестве значения функцию, которая определяет область определения отдельных элементов индексированной переменной. Например:

```
model.A = pyo.Set(initialize=[1,2,3])
def s_domain(model, i):
    return pyo.RangeSet(i, i+1, 1) # (начало, конец, шаг)
model.s = pyo.Var(model.A, domain=s_domain)
```

Здесь s – индексированная переменная, для элементов которой областями определения являются последовательные интервалы целых чисел.

> **ПРИМЕЧАНИЕ** Pyomo поддерживает общее представление для ограничения области определения переменной, но не обо всех решателях можно сказать то же самое. Бывает, что для работы с конкретным решателем приходится ограничивать себя.

Границы переменных можно явно задать с помощью именованного параметра `bounds`:

```
model.A = pyo.Set(initialize=[1,2,3])
model.a = pyo.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 = pyo.Var(model.A, bounds=f)
```

Значением параметра `bounds` может быть 2-кортеж, содержащий нижние и верхние границы. Или же функция, которая возвращает 2-кортеж для каждого индекса переменной. Отметим, что любая граница может быть равна `None`, если соответствующее ограничение отсутствует. В примере выше для `model.a` задана нижняя граница 0, а верхняя граница не задана, тогда как для `model.b` заданы различные границы по каждому индексу. Например, для `model.b[3]` нижняя граница равна 6,5, а верхняя – 7,5.

Начальное значение переменной можно задать с помощью именованного параметра `initialize`:

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

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

Для неиндексированных переменных именованному аргументу `initialize` присваивается скалярное значение. Для индексированного компонента задавать скалярное значение тоже можно, в этом случае все элементы индексированной переменной будут инициализированы одним и тем же значением. Можно также передать словарь, ключами которого являются допустимые индексы переменной. Наконец, значением этого аргумента может быть правило (функция Python), которое принимает модель и индекс в качестве аргументов и возвращает начальное значение элемента переменной с этим индексом:

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

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

4.3.2. Работа с объектами Var

При форматировании результатов или разработке нетривиальных технологических процессов часто используются атрибуты и методы объекта `Var`. Рассмотрим следующие объявления:

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

Текущее значение переменной можно получить с помощью функции `value()`, а атрибуты `lb` и `ub` содержат значения нижней и верхней границ переменной соответственно. Эти значения можно вывести из области определения переменной.

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

Методы `setlb` и `setub` используются для задания нижней и верхней границ переменной.

Значение переменной можно задать с помощью оператора присваивания Python:

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

Можно также вызвать метод `set_values`, чтобы задать значения всех элементов переменной из словаря.

Значения компонентов `Var` можно зафиксировать. Если атрибут `fixed` равен `True`, то значение переменной не может быть изменено оптимизатором. Для фиксирования элементов `Var` служит метод `fix`, а для отмены фиксирования – метод `unfix`.

```
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. Целевые функции

Целевая функция подлежит минимизации или максимизации решателем. Решатель ищет значения переменных, доставляющие наилучшее возможное значение функции. В следующих разделах описаны синтаксис объявления целевых функций и работа с ними.

4.4.1. Объявление Objective

Большинство решателей умеют работать только с моделями оптимизации, содержащими единственную целевую функцию. В следующей строке создается объект `Objective`:

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

Поддерживаются именованные и позиционные аргументы, в табл. 4.3 перечислены аргументы конструктора `Objective`.

Таблица 4.3. Аргументы в объявлении компонента `Objective`

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`expr`	выражение, определяющее целевую функцию	любое допустимое выражение Pyomo
`rule`	функция-правило, которая дает доступ к выражению, определяющему целевую функцию	функция, возвращающая выражение Pyomo или значение `Objective.Skip`
`sense`	направление оптимизации: минимизация или максимизация (по умолчанию – минимизация)	`minimize` или `maximize`

Для задания явного выражения целевой функции применяется именованный аргумент `expr`. Можно также использовать именованный аргумент `rule` для задания функции-правила, которая возвращает выражение и контролирует построение целевой функции. Ниже показаны обе возможности:

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

model.b = pyo.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 = pyo.Objective(rule=m_rule)
```

Некоторые решатели умеют выполнять многоцелевую оптимизацию, когда имеется две или более целевых функций. Их можно объявлять по отдельности или проиндексировать и определить с помощью правила, как показано ниже:

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

Если объект Objective объявлен как индексированный компонент, то Pyomo обходит все элементы множества индексов на этапе конструирования объекта, передавая каждый элемент функции в виде значения именованного аргумента rule. Если в объявлении Objective указано несколько множеств, то Pyomo обходит их декартово произведение, передавая правилу по одному элементу из каждого множества.

Иногда бывает удобно не определять целевые функции для некоторых индексов. Если правило конструирования возвращает значение Objective.Skip, то соответствующая целевая функция игнорируется.

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

По умолчанию в объявлении объекта Objective предполагается, что требуется минимизировать целевую функцию. Если на самом деле ее нужно максимизировать, то следует задать sense=pyo.maximize.

4.4.2. Работа с объектами Objective

У целевой функции есть несколько атрибутов, полезных для написания скриптов и отладки. В атрибуте expr хранится выражение функции. Атрибут sense показывает, что нужно сделать: минимизировать или максимизировать функцию. Атрибут value можно использовать для вычисления значения функции. Все они иллюстрируются в примере ниже:

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

4.5. Ограничения

Ограничение определяет одно или несколько выражений, ограничивающих возможные значения переменных. Объявление ограничения похоже на объявление целевой функции. Отличие в том, что выражение ограничения может включать операторы сравнения (на равенство или неравенство). Хотя целевые функции можно индексировать, используется эта возможность нечасто. Напротив, ограничения обычно индексируются, что позволяет конструировать коллекцию связанных ограничений и хранить ее в одном объекте.

4.5.1. Объявление Constraint

Следующий код создает один неиндексированный объект `Constraint`:

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

Поддерживаемые именованные аргументы конструктора `Constraint` перечислены в табл. 4.4.

Выражение можно как задать в именованном аргументе `expr`, так и сгенерировать функцией-правилом. Например, ограничение `diff` можно объявить следующим образом:

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

Таблица 4.4. Аргументы в объявлении компонента `Constraint`

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`expr`	выражение, определяющее ограничение	любое допустимое выражение Pyomo с оператором сравнения, 2-кортеж или 3-кортеж
`rule`	функция-правило, которая дает доступ к выражению, определяющему ограничение	функция, возвращающая выражение Pyomo с оператором сравнения, 2-кортеж или 3-кортеж или значение `Constraint.Skip`

Ограничения можно индексировать, а индексы использовать для ссылки на элементы индексированных параметров и переменных при конструировании выражений. В следующем фрагменте приведен пример:

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

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

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

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

Индексированные ограничения задаются так же, как индексированные целевые функции. Pyomo обходит декартово произведение множеств индексов, передавая функции-правилу по одному индексу из каждого множества. Ограничение `CoverConstr` в примере выше реализует следующую математическую модель:

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

При тех данных, которые заданы для a и b, экземпляр модели, переданный решателю, будет включать следующие явные ограничения:

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

В Pyomo допускается три типа выражений ограничений:

- ограничения типа неравенства имеют вид

  $expr_1 \leqslant expr_2$ или $expr_1 \geqslant expr_2$,

  где $expr_1$ и $expr_2$ могут быть неконстантными выражениями (отметим, что ограничения типа < и > не поддерживаются);
- ограничения типа равенства имеют вид

  $expr_1 = expr_2$,

  где $expr_1$ и $expr_2$ могут быть неконстантными выражениями;
- диапазонные ограничения имеют вид

  $lower \leqslant expr_1 \leqslant upper$,

  где *lower* и *upper* – константные выражения, а $expr_1$ – неконстантное.

В некоторых моделях оптимизации ограничение может быть определено не для всех индексов. Например, некоторые индексы могут быть нереализуемы физически. Функция-правило может возвращать значение `Constraint.Skip` (или `Constraint.NoConstraint`), означающее, что с данным индексом не ассоциировано никакое ограничение. Например, рассмотрим объявление воображаемого ограничения на планирование задач:

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

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

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

model.Pred = pyo.Constraint(TimePeriods, rule=Pred_rule)
```

Значение `Constraint.Skip` означает, что ограничение не генерируется и соответствующий индекс пропускается. Альтернатива этому подходу – сконструировать разреженное множество индексов, в котором заданы только индексы, действительно участвующие в ограничении. Однако в сложных моделях это не всегда практично (разреженные множества обсуждаются в разделе 9.4).

Значение `Constraint.Feasible` указывает, что ограничение, генерируемое для данного индекса, всегда выполняется. Поэтому его не нужно генерировать, и индекс пропускается. Аналогично значение `Constraint.Infeasible` означает, что генерируемое ограничение никогда не выполняется. Его можно использовать, например, в случае, когда некоторая комбинация значений параметров порождает некорректное ограничение. В таком случае Pyomo возбуждает исключение, поскольку обычно это означает, что в модели или в данных имеется ошибка.

4.5.2. Работа с объектами Constraint

После того как ограничение объявлено, выражение ограничения обрабатывается с целью определить элементы логического кортежа (*lower*, *body*, *upper*): все неконстантные выражения перемещаются в тело *body*. Таким образом, атрибуты `lower` и `upper` – константные выражения или `None`, а атрибут `body` содержит выражение Pyomo. Если `Constraint` содержит выражение равенства, то атрибут `equality` равен `True`, а атрибуты `lower` и `upper` имеют одинаковые значения.

Значение тела ограничения можно вычислить с помощью функции `value`. Аналогично методы `lslack` и `uslack` можно использовать для вычисления резервов (разностей между текущим значением выражения и нижней или верхней границей):

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

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

print(pyo.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. Множества

Множество – это коллекция данных, которая может содержать как числовые данные (например, целые или вещественные), так и символические (например, строки). Для определения множеств в моделях Pyomo предусмотрено несколько классов:

`Set`	обобщенный компонент для определения множеств;
`RangeSet`	компонент для определения числовых диапазонов;
`SetOf`	компонент, который создает множество из внешних данных без их копирования.

4.6.1. Объявление Set

В следующей строке создается объект `Set`:

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

Поддерживаются именованные и позиционные аргументы, в табл. 4.5 перечислены аргументы конструктора `Set`.

Таблица 4.5. Аргументы в объявлении компонента Set

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`initialize`	задает начальное значение, сохраняемое в множестве	список Python, словарь Python или функция-правило
`within` или `domain`	задает допустимые значения, которые могут храниться в множестве	объект Pyomo `Set` или список Python
`ordered`	следует ли сохранять порядок элементов множества	`True`, `False`, `Set.InsertionOrder` или `Set.SortedOrder`
`bounds`	задает нижнюю и верхнюю границы элементов множества	2-кортеж, словарь Python или функция-правило
`filter`	задает правило для определения того, является ли значение элементом множества	функция-правило

Можно также определить индексированное множество, указав другие множества или списки Python в качестве позиционных аргументов конструктора:

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

Аналогично можно использовать стандартные типы Python для определения множества индексов:

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

При объявлении множеств можно использовать теоретико-множественные операции:

```
model.A = pyo.Set()
model.B = pyo.Set()
model.G = model.A | model.B # объединение множеств
model.H = model.B & model.A # пересечение множеств
model.I = model.A - model.B # разность множеств
model.J = model.A ^ model.B # симметрическая разность множеств
```

Для определения декартова произведения используется оператор умножения:

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

Для задания элементов множества можно использовать именованный аргумент `initialize`:

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

Значением параметра `initialize` может быть также словарь Python, в котором задаются значения элементов множества для каждого индекса:

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

Можно также передать в параметре `initialize` функцию-правило, которая принимает модель и индекс и возвращает элементы множества для этого индекса:

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

В примерах выше показано, как можно задавать или динамически генерировать данные для инициализации множества. Но в некоторых случаях проще задать те элементы, которые не должны принадлежать множеству. Именованный параметр `filter` задает функцию, которая возвращает `True`, если элемент принадлежит множеству, и `False` в противном случае. Например:

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

Здесь множество `P` содержит простые числа, а множество `Q` – все числа, кроме входящих в `P`.

После того как в конкретной модели сконструировано индексированное множество, можно добавлять множества для конкретных индексов, применяя оператор присваивания Python:

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

Проверку данных множества можно организовать двумя способами. Во-первых, можно определить надмножество с помощью именованного параметра `within` или `domain`:

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

При добавлении элемента в множество `B` проверяется, что он принадлежит `A`. Тем самым гарантируется, что `B` – подмножество `A`.

Можно также определить функцию-правило с помощью именованного параметра `validate`. Она должна возвращать `True`, если переданный элемент принадлежит множеству, и `False` в противном случае (тогда Pyomo возбудит исключение). Например, следующая функция `C_validate` имитирует поведение именованного аргумента `within`:

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

Наконец, отметим, что если заданы оба аргумента `within` и `validate`, то для проверки добавляемых в множество элементов применяются оба механизма.

По умолчанию элементы множества хранятся в том порядке, в каком вставлялись. Но иногда требуется хранить их в отсортированном порядке. Это можно сделать, задав значение `Set.SortedOrder` аргумента `ordered`:

```
model.A = pyo.Set(ordered=pyo.Set.SortedOrder)
```

Элементы множества могут быть одиночными значениями или k-кортежами. Для задания ожидаемой размерности данных служит именованный параметр `dimen`. Значение по умолчанию равно 1 и означает, что элементы – одиночные значения. В некоторых случаях размерность можно определить по другим параметрам, но, вообще говоря, пользователь должен сделать это сам, когда элементами множества являются кортежи.

В упорядоченных множествах может быть первое и последнее значения. Аргумент `bounds` задает 2-кортеж, определяющий верхнюю и нижнюю границы множества. Если множество упорядочено, то границы можно вывести из аргумента `within`.

Компонент `RangeSet` определяет упорядоченное виртуальное множество, представляющее последовательность чисел – целых или с плавающей точкой. Эта последовательность задается начальным значением, конечным значением и шагом. Если при создании объекта `RangeSet` задан только один аргумент, то считается, что это конечное значение. При этом начальное значение и шаг по умолчанию равны 1. Например, следующий код определяет последовательность целых чисел от 1 до 10:

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

Если при определении RangeSet задано два аргумента, то первый считается начальным значением, а второй – конечным. Например, следующий код определяет последовательность целых чисел от 5 до 10:

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

Наконец, если при определении RangeSet задано три аргумента, то они интерпретируются как начальное значение, конечное значение и шаг соответственно. Например, следующий код определяет последовательность целых чисел с плавающей точкой от 2.5 до 10.0 с шагом 1.5:

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

4.6.2. Работа с объектами Set

Функция len() возвращает количество элементов множества:

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

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

К элементам множества можно обращаться с помощью метода data(), который возвращает данные в виде кортежа Python (или словаря Python для индексированных множеств):

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

print(type(model.A.data()) is tuple) # True
print(type(model.B.data()) is tuple) # 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]
```

Для сравнения множеств и проверки принадлежности объекта множеству применяются операторы Python:

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

# Проверить, принадлежит ли элемент множеству
print(1 in model.A) # True

# Верно ли, что множества равны
print([1, 2] == model.A) # False

# Верно ли, что множества не равны
print([1, 2] != model.A) # True
```

```
# Верно ли, что множество является подмножеством или совпадает с другим множеством
print([1, 2] <= model.A) # True

# Верно ли, что множество является подмножеством другого множества
print([1, 2] < model.A) # True

# Верно ли, что множество является надмножеством другого множества
print([1, 2, 3] > model.A) # False

# Верно ли, что множество является надмножеством или совпадает с другим множеством
print([1, 2, 3] >= model.A) # True
```

Множество можно обойти, обращаясь к каждому его элементу:

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

У упорядоченных множеств есть ряд методов, учитывающих порядок элементов:

```
model.A = pyo.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
```

Методы `first()` и `last()` возвращают первый и последний элементы множества соответственно. Метод `next()` принимает элемент множества и возвращает следующий за ним. Метод `prev()` возвращает предыдущий элемент. Методы `nextw()` и `prevw()` работают аналогично, но при достижении конца или начала множества переходят в начало или в конец соответственно. В примере выше `nextw(1)` возвращает 3, потому что 1 – последний элемент множества, а 3 следует за ним, если множество перебирается циклически. Метод `ord()` возвращает индекс позиции элемента упорядоченного множества, а оператор `[]` можно использовать для доступа к элементу в позиции с заданным индексом:

```
model.A = pyo.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
```

ПРИМЕЧАНИЕ Позиции нумеруются с единицы, а не с нуля. Порядок множества определяется последовательностью данных, заданной при создании экземпляра, и значением именованного аргумента `ordered`.

4.7. Параметры

Параметр – это числовое или символическое значение, используемое для задания ограничений и целевых функций модели. Для создания параметров в Pyomo существует класс `Param`, который может содержать одиночное значение, массив значений или многомерный массив значений. Неиндексированный компонент `Param` очень похож на скалярное значение, а индексированный – на словарь Python. Компонент `Param` поддерживает изменяемость и разреженные представления со значениями по умолчанию.

4.7.1. Объявление Param

В следующей строке создается объект `Param`:

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

Поддерживаются именованные и позиционные аргументы, в табл. 4.6 перечислены аргументы конструктора `Param`.

Таблица 4.6. Аргументы в объявлении компонента `Param`

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`initialize`	задает начальное значение, сохраняемое в множестве	скалярное значение, словарь Python или функция-правило
`default`	задает значение по умолчанию, которое будет использоваться, если не задано никакое другое значение параметра	скалярное значение, словарь Python или функция-правило
`validate`	задает функцию, которая вызывается, чтобы определить, допустимо ли данное значение параметра	функция, принимающая значение параметра и возвращающая `True` или `False`
`mutable`	может ли параметр изменяться между вызовами решателя	`True` или `False`

Можно также определить индексированный параметр, указав множества в качестве позиционных аргументов конструктора:

```
model.A = pyo.Set(initialize=[1,2,3])
model.B = pyo.Set(initialize=['A','B'])
model.U = pyo.Param(model.A, initialize={1:10, 2:20, 3:30})
```

```
model.T = pyo.Param(model.A, model.B,
                    initialize={(1,'A'):10, (2,'B'):20, (3,'A'):30})
```

Аргумент `initialize` позволяет задать значение параметра, как было показано в двух предыдущих фрагментах кода. Для той же цели аргументу `initialize` можно передать функцию-правило:

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

Если для определения индекса индексированного параметра используются упорядоченные множества, то функция инициализации может ссылаться на ранее определенные значения параметра:

```
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 = pyo.Param(model.A, model.A, initialize=XX_init)
```

Именованный аргумент `default` можно использовать, чтобы задать не инициализированные явно значения параметра для всех допустимых индексов. Например, можно определить индексированный параметр, представляющий диагональную матрицу 3×3:

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

Как и для компонента `Set`, существует два способа проверки значений параметра. Во-первых, именованный аргумент `within` позволяет задать область определения значений параметра:

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

Во-вторых, проверить корректность значений параметра можно с помощью аргумента `validate`, который задает функцию, возвращающую `True`, если значение допустимо, и `False` в противном случае (тогда Pyomo возбудит исключение). В примере ниже аргумент `validate` имитирует поведение аргумента `within`:

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

Индексированные параметры проверяются аналогично. Аргумент `validate` задает функцию, которая принимает модель, значение параметра и индекс:

```
model.A = pyo.Set(initialize=[1,2,3])
def X_validate(model, value, i):
```

```
    return value > i
model.X = pyo.Param(model.A, validate=X_validate)
```

Если заданы оба аргумента `within` и `validate`, то для проверки значений параметра применяются оба механизма.

Компонент `Param` можно использовать для представления константных значений в моделях Pyomo, однако изменяемость также поддерживается. В следующем примере Pyomo порождает выражение для целевой функции модели вида

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

А именно Pyomo рассматривает значения параметров как фиксированные константы, а их выражения просто содержат числовые постоянные.

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

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

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

Отметим, что это «преобразование» происходит сразу после создания выражения. Тот факт, что источником значений является компонент `Param`, забыт, остались только числовые значения. Это сделано ради эффективности. Таким образом, эти значения *нельзя изменять* после того, как выражение создано.

Однако это поведение меняется, если при конструировании модели был задан аргумент `mutable`. Если он равен `True`, то значения параметров уже не считаются константами. Рассмотрим тот же пример, но сделаем параметр `p` изменяемым:

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

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

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

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

Генерируя выражение целевой функции в этой модели, Pyomo сохраняет информацию о компоненте `Param`, так что выражение теперь имеет вид:

$$p_1x_1 + p_2x_2 + p_3x_3,$$

где значения p_i – объекты `Param`, сохраняющие ссылки на значения параметров. Здесь Pyomo рассматривает значения параметров как изменяемые, до-

пуская, что впоследствии они могут быть модифицированы пользователем. В данном примере значения параметров изменяются после того, как выражение целевой функции определено, так что результирующая целевая функция равна

$x_1 + 4.2x_2 + 3.14x_3$.

Параметры заменяются своими числовыми значениями только при вызове решателя. Стало быть, их значения можно изменять между последовательными вызовами решателя.

С изменяемыми параметрами сопряжен дополнительный расход памяти, и требуется дополнительная обработка на этапе преобразования выражений Pyomo в форму, понятную решателю. Поэтому по умолчанию параметры неизменяемы.

4.7.2. Работа с объектами Param

Pyomo предполагает, что представление значений параметров разреженное. Например, в объявлении объекта `T` типа `Param` заданы два множества индексов, `A` и `B`:

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

Однако не все эти значения необходимо определять в модели, например:

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

Параметр `W` определен для индексов 1 и 3, но множество индексов `B` содержит элементы 1, 2 и 3. При попытке доступа к `W[2]` произойдет ошибка, и Python возбудит исключение.

Как отмечалось выше, с помощью именованного аргумента `default` можно задать значение по умолчанию. Если оно задано и модель пытается обратиться к неинициализированному значению, то вместо него используется значение по умолчанию (и исключение не возбуждается). Отметим, что представление параметров разреженное, даже если значение по умолчанию задано. Это сделано ради эффективного использования памяти. Таким образом, автор модели может обращаться к разреженным значениям, не изобретая специализированную структуру данных.

В силу разреженности представления несколько методов, учитывающих допустимость индексированных параметров, нуждаются в специальном поведении. Пусть *допустимое множество индексов* ссылается на полный список всех допустимых индексов, как инициализированных, так и неинициализированных, и пусть *эффективное множество индексов* – это его часть, соответствующая инициализированным значениям ключей в индексированной

компоненте. Если значение по умолчанию не задано, то функция `len` возвращает размер эффективного множества индексов, а оператор `in` проверяет присутствие указанного значения в эффективном множестве. Итерирование осуществляется по эффективному множеству индексов, и оператор Python `[]` можно использовать для доступа к отдельным элементам (в данном случае значениям параметра).

Если значение по умолчанию задано, то все индексы в модели равно допустимы, не важно, индексируют они какой-то элемент или нет. Поэтому функция `len()` возвращает размер полного множества индексов, а итерирование и оператор `in` тоже применяются к полному множеству. Таким образом, если значение по умолчанию задано, то множество значений параметра кажется плотным, хотя в действительности структура данных по-прежнему разрежена. Это иллюстрируется примером ниже:

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

# Демонстрация функции len()
print(len(model.p)) # 2
print(len(model.q)) # 3

# Демонстрация оператора in (проверка ключей компонента)
print(2 in model.p) # False
print(2 in model.q) # True

# Демонстрация итерирования по ключам компонента
print([key for key in model.p]) # [1, 3]
print([key for key in model.q]) # [1, 2, 3]
```

Методы `sparse_keys()`, `sparse_values()`, `sparse_items()`, `sparse_iterkeys()`, `sparse_itervalues()` и `sparse_iteritems()` – разреженные версии соответствующих методов класса `IndexedComponent`. Они возвращают значения только определенных параметров вне зависимости от того, задано значение по умолчанию или нет.

4.8. Именованные выражения

Выражения Pyomo могут содержать числа, параметры и переменные, соединенные знаками операций, например `+`, `-`, `*`, `/`. Эти выражения лежат в основе алгебраического представления модели и хранятся в компонентах ограничений и целевой функции.

Компонент `Expression` предоставляет механизм для хранения выражения Pyomo в модели, благодаря которому выражение можно повторно использовать в нескольких контекстах (например, как общее подвыражение одного или нескольких ограничений), не неся каждый раз накладных расходов на его генерирование. Кроме того, выражение Pyomo, хранящееся в компоненте `Expression`, можно изменить впоследствии и таким образом обновить ссы-

лающееся на него ограничение или целевую функцию. Это эффективный подход к модификации модели между вызовами решателя.

В следующих разделах описываются синтаксис объявления именованных выражений и работа с ними.

4.8.1. Объявление Expression

В следующей строке создается одиночный, неиндексированный объект `Expression`:

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

Поддерживаются именованные и позиционные аргументы, в табл. 4.7 перечислены аргументы конструктора `Expression`.

Именованные параметры `expr` и `rule` можно использовать для инициализации именованного выражения в момент объявления:

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

Таблица 4.7. Аргументы в объявлении компонента `Expression`

Имя	Описание	Допустимые значения
позиционный	зарезервированы для задания множеств индексов	произвольное число объектов Pyomo `Set` или списков Python
`expr`	хранимое выражение	любое допустимое выражение Pyomo
`rule`	функция-правило, которая вызывается для получения хранимого выражения	функция, возвращающая выражение Pyomo или значение `Expression.Skip`

Как и все прочие компоненты, `Expression` можно индексировать, включив в объявление один или более позиционных аргументов, представляющих множества индексов. В примере ниже объявляется индексированный компонент `Expression`, в котором определены элементы для всех индексов, кроме первого. Если некоторый индекс должен быть опущен, то функция-правило возвращает для него значение `Expression.Skip`.

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

4.8.2. Работа с объектами Expression

Проще всего объявить одиночное выражение и использовать его в объявлении целевой функции или ограничения:

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

Значение именованного выражения можно вычислить с помощью функции `value`. Кроме того, выражение, хранящееся в компоненте `Expression`, можно обновлять. Как видно из следующего примера, после обновления выражения изменяются также целевая функция и ограничение, в котором оно используется:

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

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

С компонентом `Expression` необязательно связывать выражение в момент его объявления в модели, но оно должно быть присвоено до передачи модели решателю, если компонент используется в каких-то активных целевых функциях или ограничениях. Кроме того, в именованных выражениях, используемых в целевых функциях или ограничениях, нельзя хранить реляционные выражения Pyomo, т. е. выражения, содержащие операторы `<=`, `<`, `>=`, `>` и `==`.

4.9. Суффиксы

Суффиксы – это механизм аннотирования модели дополнительными данными, не имеющими прямого отношения к объявлению и структуре модели. Обычно суффиксы используются плагинами решателей для хранения дополнительной информации о решении. Вообще, суффиксы можно использовать для следующих целей:

- импорт информации о решении из решателя в математическую программу (например, двойственных ограничений, переменных приведенных затрат, базисной информации);
- экспорт информации в решатель или алгоритм для настройки процесса решения (например, информации о теплом старте, приоритеты ветвления по переменным);

- пометка компонентов модели локальными данными для последующего использования в продвинутых скриптах.

Эта функциональность доступна автору модели в виде класса `Suffix`, предоставляющего интерфейс для аннотирования компонентов модели Pyomo дополнительными данными.

4.9.1. Объявления Suffix

В следующей строке создается суффикс `foo`:

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

Поддерживаются именованные аргументы, в табл. 4.8 перечислены аргументы конструктора `Suffix`.

Таблица 4.8. Аргументы в объявлении компонента `Suffix`

Имя	Описание	Допустимые значения
`direction`	определяет, является ли суффикс входом или выходом решателя	`Suffix.LOCAL`, `Suffix.IMPORT`, `Suffix.EXPORT`, `Suffix.IMPORT_EXPORT` (детали см. ниже)
`datatype`	определяет тип данных, хранящихся в суффиксе	`Suffix.FLOAT`, `Suffix.INT`, `None` (детали см. ниже)
`initialize`	задает начальные значения суффикса	функция-правило

Компонент `Suffix` не является индексируемым, поэтому в его объявлении не может быть безымянных позиционных аргументов. Именованный аргумент `direction` определяет направление потока информации при взаимодействии с решателем и может принимать одно из четырех значений:

- `Suffix.LOCAL`: суффиксные данные локальны относительно модели. Они не импортируются и не экспортируются плагинами решателя. Это значение по умолчанию;
- `Suffix.IMPORT`: суффиксные данные импортируются из решателя в модель плагинами решателя;
- `Suffix.EXPORT`: суффиксные данные экспортируются из модели в решатель плагинами;
- `Suffix.IMPORT EXPORT`: суффиксные данные импортируются и экспортируются плагинами решателя.

Не все плагины решателей умеют обрабатывать суффиксную информацию, но пользователь может управлять ее потоком, настраивая суффиксные компоненты.

Именованный аргумент `datatype` определяет тип данных, хранящихся в суффиксе, и может принимать одно из трех значений:

- `Suffix.FLOAT`: данные с плавающей точкой (по умолчанию);
- `Suffix.INT`: целочисленные данные;
- `None`: данные любого типа.

Для некоторых интерфейсов с решателями этот аргумент необязателен, но если данные экспортируются в решатель через файловый интерфейс `nl`, то во всех активных экспортируемых суффиксах тип данных должен быть определен строго (т. е. значение `None` аргумента `datatype` не допускается).

В примере ниже иллюстрируются различные объявления суффиксов:

```
# Экспорт целочисленных данных
model.priority = pyo.Suffix(direction=pyo.Suffix.EXPORT,
                  datatype=pyo.Suffix.INT)

# Экспорт и импорт данных с плавающей точкой
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT_EXPORT)
```

Не гарантируется, что суффиксы совместимы со всеми плагинами решателей в Pyomo. Допустим суффикс или нет, зависит как от решателя, так и от используемого интерфейса с решателем. В некоторых случаях плагин возбуждает исключение, встретив суффикс, который не умеет обрабатывать, но так бывает не всегда. Например, файловый интерфейс `nl` является общим для всех AMPL-совместимых решателей, поэтому Pyomo никак не может проверить, что суффикс с данным именем, направлением и типом данных подходит для решателя. При переходе на другой решатель или интерфейс следует внимательно проверять, что объявления суффикса обрабатываются, как ожидалось.

Именованный аргумент `initialize` можно использовать для определения значений суффикса. Он задает функцию, которая вызывается при конструировании модели и возвращает список или иной итерируемый объект, состоящий из кортежей (компонент, значение).

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

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

4.9.2. Работа с суффиксами

Рассмотрим следующий пример:

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

Здесь мы видим два компонента `Var`, индексированный и неиндексированный, и суффикс. Концептуально объявление суффикса `foo` позволяет связать `foo` с каждым компонентом модели. Например:

```
# Присвоить значение 1.0 суффиксу 'foo' для model.x
model.x.set_suffix_value('foo', 1.0)

# Присвоить значение 2.0 суффиксу model.foo для model.x
model.x.set_suffix_value(model.foo, 2.0)

# Получить значение суффикса 'foo' дл model.x
print(model.x.get_suffix_value('foo')) # 2.0
```

Для присваивания значения суффиксу служит метод `set_suffix_value`, а для получения значения – метод `get_suffix_value`. Этот пример иллюстрирует два способа задания одного и того же суффикса: по имени и по объекту компонента.

Суффиксам индексированных компонентов также можно присваивать значения методом `set_suffix_value`:

```
# Присвоить значение 3.0 суффиксу model.foo для model.y
model.y.set_suffix_value(model.foo, 3.0)

# Присвоить значение 4.0 суффиксу model.foo для model.y[2]
model.y[2].set_suffix_value(model.foo, 4.0)

# Получить значение суффикса 'foo' для 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
```

Этот пример показывает использование метода `set_suffix_value` для задания значения индексированного и одиночного компонента. Когда этот метод вызывается для индексированного компонента, по умолчанию значение суффикса задается для всех элементов, или индексов компонента, а не для самого компонента. Поэтому, попытавшись получить значение суффикса для компонента `model.y`, мы обнаружили, что оно равно `None`.

Значения суффиксов можно также очищать, что эквивалентно заданию значения `None`:

```
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. Другие компоненты моделирования

В этой главе подробно описаны некоторые из наиболее употребительных компонентов моделирования в Pyomo. Но есть и другие, а именно:

- `Block`: компонент `Block` позволяет объявлять модели с повторяющейся или вложенной структурой (например, в модели могут существовать отдельные объекты `Block` для представления различных моментов времени в многопериодной оптимизации). Блок состоит из коллекции компонентов моделирования. Более подробно блоки обсуждаются в главе 8;
- `Model`: компонент `Model` – это контейнер для группировки компонентов моделирования. Pyomo поддерживает абстрактные и конкретные представления модели. Хотя объекты «моделей» использовались выше повсеместно (они необходимы для формулирования и решения задач оптимизации), мы не отметили тот факт, что они сами являются компонентами. На самом деле они наследуют компоненту `Block`;
- `Complementarity`: этот компонент используется для определения условий дополнительности в математической программе с ограничениями равновесия. Поддерживается несколько форм такой дополнительности. Компонент документирован в главе 13;
- `ContinuousSet`: используется для представления ограниченных непрерывных областей определения в контексте моделирования дифференциальных уравнений. Документирован в главе 12;
- `DerivativeVar`: используется для представления производных компонентов `Var` в контексте моделирования дифференциальных уравнений. Документирован в главе 12;
- `Disjunct`: этот компонент поддерживает обобщенное дизъюнктивное программирование (Generalized Disjunctive Programming – GDP) в Pyomo. Это контейнер для индикаторной переменной и множества ограничений, которые должны быть активны, когда эта индикаторная переменная равна `True`. Этот компонент документирован в главе 11;
- `Disjunction`: поддерживает обобщенное дизъюнктивное программирование в Pyomo. Содержит множество объектов `Disjunct`, соединенных логическим оператором «OR». Документирован в главе 11;
- `Piecewise`: этот компонент поддерживает кусочное моделирование функций общего вида и в том числе различные преобразования для порождения смешанно-целочисленных представлений кусочных функций. Дополнительную документацию можно найти на сайте Pyomo;
- `SOSConstraint`: специальные упорядоченные множества (special ordered set – SOS) определяются в Pyomo с помощью компонента `SOSConstraint`. Поддерживаются такие множества типа 1 и 2 (SOS1 и SOS2). Дополнительную документацию можно найти на сайте Pyomo;
- `BuildAction` и `BuildCheck`: эти компоненты применяются главным образом в абстрактных моделях и описаны в разделе 10.4.

Глава 5

Программирование нестандартных технологических процессов

В этой главе иллюстрируется использование Python совместно с Pyomo для анализа решения и разработки нестандартных технологических процессов или высокоуровневых метаалгоритмов. Например, показано, как получить доступ к значениям переменной и целевой функции, добавлять и удалять ограничения и обходить компоненты модели. Также приведены более полные примеры, показывающие, как пользователь Pyomo может разработать собственные стратегии решения и анализа, не ограничиваясь тем, что предлагается «из коробки».

5.1. Введение

В предыдущих главах мы описали, как обобщенный процесс оптимизации можно реализовать с помощью Pyomo: построить модель, решить ее и отобразить результаты. Благодаря использованию Python и Pyomo API резко повышается гибкость разработки продвинутых технологических процессов. Для других AML изобретается новый скриптовый язык, и разработчики пакета создают для него грамматический анализатор. Это отделяет пользователя от кода самого каркаса. В случае Pyomo и сам каркас, и среда моделирования написаны на Python. Это дает пользователю полный контроль над всем процессом решения и выливается в два важных высокоуровневых преимущества:

- пользователи Pyomo могут применять существующие библиотеки Python для анализа данных до и после решения задачи оптимизации;

- Pyomo поддерживает разработку алгоритмов, требующих преобразования задачи и нескольких решателей с различными структурами и данными. Вкупе с возможностями программирования на Python это открывает возможность для построения высокоуровневых алгоритмов (например, разложение Бендерса, MINLP-решатели и стратегии многоступенчатой инициализации).

В этой главе обсуждаются основы написания скриптов с использованием Pyomo. Эта функциональность демонстрируется на различных примерах, включая решатель головоломок судоку.

> **ПРИМЕЧАНИЕ** В данной главе показано, чего можно достичь в Pyomo благодаря языку Python, а в примерах используются методы компонентов, образующих часть базовой инфраструктуры Pyomo. Разработчики Pyomo стремятся сохранять обратную совместимость всюду, где возможно. Однако отметим, что встречающиеся в этой главе методы могут измениться с большей вероятностью, чем прочие возможности, обсуждаемые в книге.

В главе 3 мы поставили задачу о расположении складов. В ней требовалось найти оптимальные местоположения складов, удовлетворяющие спрос потребителей с наименьшими транспортными затратами. Если вы забыли формулировку, обратитесь к разделу 3.2.

В следующем примере, основанном на задаче о расположении складов, иллюстрируются основные элементы, встречающиеся практически в любом скрипте решения задачи с применением Pyomo: (1) загрузка данных, (2) создание модели, (3) выполнение оптимизации модели путем обращения к интерфейсу с решателем и (4) извлечение и отображение решения.

```
import json
import pyomo.environ as pyo
from warehouse_model import create_wl_model

# загрузить данные из json-файла
with open('warehouse_data.json', 'r') as fd:
    data = json.load(fd)

# вызвать функцию создания модели
model = create_wl_model(data, P=2)

# решить модель
solver = pyo.SolverFactory('glpk')
solver.solve(model)

# посмотреть на решение
model.y.pprint()
```

Для этого скрипта нужно два дополнительных файла. В стандартный дистрибутив Python включены классы для чтения и записи файлов в формате JSON. Показанный ниже файл содержит необходимые данные в этом формате.

```
{
  "WH": [
    "Harlingen",
```

```
    "Memphis",
    "Ashland"
  ],
  "CUST": [
    "NYC",
    "LA",
    "Chicago",
    "Houston"
  ],
  "dist": {
    "Harlingen": {
      "NYC": 1956,
      "LA": 1606,
      "Chicago": 1410,
      "Houston": 330
    },
    "Memphis": {
      "NYC": 1096,
      "LA": 1792,
      "Chicago": 531,
      "Houston": 567
    },
    "Ashland": {
      "NYC": 485,
      "LA": 2322,
      "Chicago": 324,
      "Houston": 1236
    }
  }
}
```

Модель складов определена в коде ниже.

```
import pyomo.environ as pyo
def create_wl_model(data, P):
    # создать модель
    model = pyo.ConcreteModel(name="(WL)")
    model.WH = data['WH']
    model.CUST = data['CUST']
    model.dist = data['dist']
    model.P = P
    model.x = pyo.Var(model.WH, model.CUST, bounds=(0,1))
    model.y = pyo.Var(model.WH, within=pyo.Binary)

    def obj_rule(m):
        return sum(m.dist[w][c]*m.x[w,c] for w in m.WH for c in m.CUST)
    model.obj = pyo.Objective(rule=obj_rule)

    def one_per_cust_rule(m, c):
        return sum(m.x[w,c] for w in m.WH) == 1
    model.one_per_cust = pyo.Constraint(model.CUST, rule=one_per_cust_rule)

    def warehouse_active_rule(m, w, c):
```

```
        return m.x[w,c] <= m.y[w]
    model.warehouse_active = pyo.Constraint(model.WH, \
        model.CUST, rule=warehouse_active_rule)

    def num_warehouses_rule(m):
        return sum(m.y[w] for w in m.WH) <= m.P
    model.num_warehouses = pyo.Constraint(rule=num_warehouses_rule)

    return model
```

Python позволяет строить гораздо более сложные технологические процессы, чем показано в примере выше. В этой главе мы опишем, как обращаться к компонентам модели и модифицировать модель из программы.

> **ПРИМЕЧАНИЕ** Программировать можно и абстрактные модели, хотя чаще всего из программы взаимодействуют с конкретными моделями. Примеры в этой главе работают только с конкретными моделями, но конкретную модель можно получить из абстрактной, вызвав метод `create_instance()`. Подробности см. в главе 10.

5.2. Опрос модели

У объектов моделирования Pyomo (например, `Var`, `Param`) есть атрибуты, к которым можно обращаться из программы. Их можно опрашивать, чтобы получить информацию о состоянии модели. В этом разделе мы покажем, как обойти компоненты модели и обратиться к основным атрибутам этих компонентов. А в главе 4 приведена дополнительная информация об атрибутах компонентов моделирования.

Например, можно получить выражение целевой функции. Рассмотрим задачу о расположении складов. Показанный ниже код печатает выражение целевой функции, ее значение, полученное в результате решения, и значение одной из переменных.

```
import json
import pyomo.environ as pyo
from warehouse_model import create_wl_model

# загрузить данные из json-файла
with open('warehouse_data.json', 'r') as fd:
    data = json.load(fd)

# вызвать функцию создания модели
model = create_wl_model(data, P=2)

# решить модель
solver = pyo.SolverFactory('glpk')
solver.solve(model)

# напечатать выражение целевой функции
print(model.obj.expr)
```

```
# напечатать значение целевой функции для решения
print(pyo.value(model.obj))

# напечатать значение конкретной переменной
print(pyo.value(model.y['Harlingen']))
```

5.2.1. Функция value

Некоторые атрибуты компонентов Pyomo могут содержать объекты Pyomo, а не значения встроенных в Python числовых типов. Например, нижняя граница `Var` может быть как простым числом с плавающей точкой (например, 3.2), так и объектом `Param`, с которым ассоциировано какое-то значение. Поэтому для получения атрибутов с ожидаемыми числовыми значениями важно использовать функцию `value`.

Как функция `value` преобразует выражения Pyomo в числовые значения, показано в примере ниже.

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.u = pyo.Var(initialize=2.0)

# неожиданное выражение вместо значения
a = model.u - 1
print(a) # "u - 1"
print(type(a)) # <class 'pyomo.core.expr.numeric_expr.SumExpression'>

# правильный способ доступа к значению
b = pyo.value(model.u) - 1
print(b) # 1.0
print(type(b)) # <class 'float'>
```

В этом примере `a` содержит объект выражения Pyomo (а не числовое значение, как можно было бы ожидать). Это становится понятно, когда мы его печатаем и проверяем тип. Предложение `print` правильно показывает тип выражения. А чтобы получить значение переменной (или других компонентов Pyomo), нужно вызвать функцию `pyo.value`. Здесь `b` содержит числовое значение, как и следовало ожидать.

Этот пример иллюстрирует создание неявных выражений и возможные нежелательные последствия. Мы настоятельно рекомендуем избегать создания выражений всюду, кроме процесса конструирования модели.

> **ПРИМЕЧАНИЕ** Часто забывают вызвать функцию `value()`, когда требуется получить значение компонента Pyomo `Var`. Например, в коде ниже Python напечатает представление объекта переменной, а не его значение:
>
> ```
> print(model.y)
> ```
>
> В данном случае будет напечатано имя переменной «y», а не значение.

5.2.2. Доступ к атрибутам индексированных компонентов

В предыдущем примере было показано, как получать значения скалярных переменных. Чтобы получить значение элемента индексированной переменной с конкретным индексом, нужно указать этот индекс:

```
print(pyo.value(model.y['Ashland']))
```

Также можно получить значения всех элементов, перебрав их в цикле:

```
for i in model.y:
    print('{0} = {1}'.format(model.y[i], pyo.value(model.y[i])))
```

Здесь в цикле перебираются ключи индексированной переменной `model.y`. Этот пример можно переписать, обойдя множество индексов непосредственно:

```
for i in model.WH:
    print('{0} = {1}'.format(model.y[i], pyo.value(model.y[i])))
```

Тот же подход годится для доступа к другим атрибутам переменных (например, нижней границы, `model.y[i].lb`) и прочих компонентов модели.

5.2.2.1. Срезы индексов компонентов

Pyomo поддерживает нотацию срезов для более точного контроля над циклом обхода элементов компонента модели. Например, мы можем найти всех потребителей, обслуживаемых данным складом:

```
for v in model.x['Ashland',:]:
    print('{0} = {1}'.format(v, pyo.value(v)))
```

> **ПРИМЕЧАНИЕ** Нотация срезов возвращает сами объекты компонентов, а не их индексы. Дополнительную информацию см. в разделе 8.5.

5.2.2.2. Обход всех объектов Var в модели

Pyomo также предоставляет общие методы для обхода компонентов заданной модели или блока. В примере ниже показано, как перебрать все объекты `Var` в модели.

```
# цикл по всем объектам Var в модели
for v in model.component_data_objects(ctype=pyo.Var):
    print('{0} <= {1}'.format(v, pyo.value(v.ub)))

# можно также обойти в цикле индексы элементов каждого объекта Var
for v in model.component_objects(ctype=pyo.Var):
    for index in v:
        print('{0} <= {1}'.format(v[index], pyo.value(v[index].ub)))
```

ПРИМЕЧАНИЕ Описанные методы можно использовать для нахождения и обхода всех компонентов в модели Pyomo. Этот подход обобщается и на иерархические модели с компонентами `Block`. Более полное их рассмотрение смотрите в онлайновой документации.

5.3. Модификация структуры модели Pyomo

Часто возникает необходимость многократно решать модель с разными значениями параметров или небольшими изменениями ограничений. Это можно эффективно сделать с помощью изменяемых параметров. Пример использования изменяемого объекта `P` типа `Param` был приведен в разделе 3.3.5.

В Pyomo разрешается модифицировать структуру модели между вызовами решателя. Например:

- целевые функции и ограничения можно активировать и деактивировать, не изменяя хранящиеся в модели данные. Деактивированные компоненты будут исключены из модели, передаваемой решателю;
- переменные можно трактовать как фиксированные или нефиксированные (по умолчанию);
- Pyomo также позволяет добавлять и удалять компоненты моделирования, например ограничения.

В следующем примере показано, как модифицировать структуру модели.

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var(bounds=(0,5))
model.y = pyo.Var(bounds=(0,1))
model.con = pyo.Constraint(expr=model.x + model.y == 1.0)
model.obj = pyo.Objective(expr=model.y-model.x)

# решить задачу
solver = pyo.SolverFactory('glpk')
solver.solve(model)
print(pyo.value(model.x)) # 1.0
print(pyo.value(model.y)) # 0.0

# добавить ограничение
model.con2 = pyo.Constraint(expr=4.0*model.x + model.y == 2.0)
solver.solve(model)
print(pyo.value(model.x)) # 0.33
print(pyo.value(model.y)) # 0.66

# деактивировать ограничение
model.con.deactivate()
solver.solve(model)
print(pyo.value(model.x)) # 0.5
print(pyo.value(model.y)) # 0.0
```

```
# вновь активировать ограничение
model.con.activate()
solver.solve(model)
print(pyo.value(model.x)) # 0.33
print(pyo.value(model.y)) # 0.66

# удалить ограничение
del model.con2
solver.solve(model)
print(pyo.value(model.x)) # 1.0
print(pyo.value(model.y)) # 0.0

# зафиксировать переменную
model.x.fix(0.5)
solver.solve(model)
print(pyo.value(model.x)) # 0.5
print(pyo.value(model.y)) # 0.5

# отменить фиксирование переменной
model.x.unfix()
solver.solve(model)
print(pyo.value(model.x)) # 1.0
print(pyo.value(model.y)) # 0.0
```

5.4. Типичные примеры программирования

В этом разделе мы приведем несколько примеров скриптов, иллюстрирующих описанные выше возможности. Дополнительные примеры можно найти в галерее Pyomo на сайте `www.pyomo.org`.

5.4.1. Цикл по местоположениям складов и построение диаграммы

В следующем примере задача о расположении складов формулируется и многократно решается для нахождения всех возможных решений. После нахождения очередного решения добавляется новый разрез, который исключает это решение, и задача решается снова для нахождения следующего решения. Процесс повторяется, пока не получится задача, не имеющая решений. Список разрезов хранится в компоненте `ConstraintList`; на каждой итерации цикла в него добавляется новый разрез.

```
import json
import pyomo.environ as pyo
```

```
from warehouse_model import create_wl_model
import matplotlib.pyplot as plt

# загрузить данные из json-файла
with open('warehouse_data.json', 'r') as fd:
    data = json.load(fd)

# вызвать функцию создания модели
model = create_wl_model(data, P=2)
model.integer_cuts = pyo.ConstraintList()
objective_values = list()
done = False
while not done:
    # решить модель
    solver = pyo.SolverFactory('glpk')
    results = solver.solve(model)

    term_cond = results.solver.termination_condition
    print('')
    print('--- Solver Status: {0} ---'.format(term_cond))

    if pyo.check_optimal_termination(results):
        # посмотреть на решение
        print('Optimal Obj. Value = {0}'.format(pyo.value(model.obj)))
        objective_values.append(pyo.value(model.obj))
        model.y.pprint()

        # создать новый целочисленный разрез, чтобы исключить это решение
        WH_True = [i for i in model.WH if pyo.value(model.y[i]) > 0.5]
        WH_False = [i for i in model.WH if pyo.value(model.y[i]) < 0.5]
        expr1 = sum(model.y[i] for i in WH_True)
        expr2 = sum(model.y[i] for i in WH_False)
        model.integer_cuts.add(
          sum(model.y[i] for i in WH_True) - sum(model.y[i] for i in WH_False) \
          <= len(WH_True)-1)
    else:
        done = True

x = range(1, len(objective_values)+1)
plt.bar(x, objective_values, align='center')
plt.gca().set_xticks(x)
plt.xlabel('Solution Number')
plt.ylabel('Optimal Obj. Value')
plt.savefig('WarehouseCuts.pdf')
```

Этот пример выводит все найденные решения на консоль. Кроме того, с помощью пакета `matplotlib` он строит диаграмму на рис. 5.1, показывающую оптимум целевой функции для каждого решения.

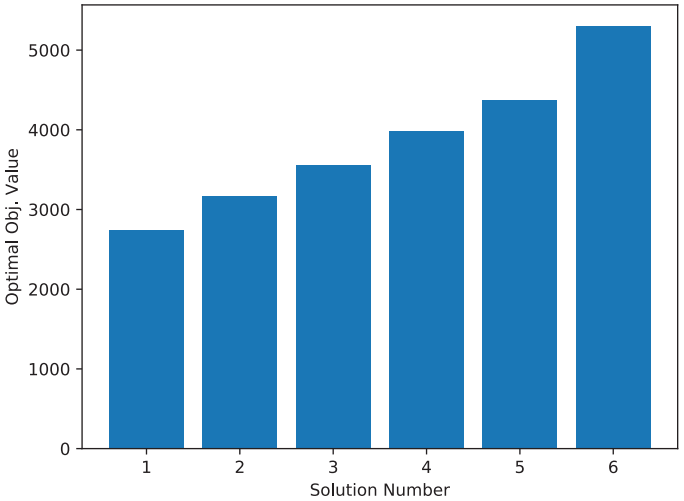

Рис. 5.1 ❖ Оптимальные значения целевой функции для последовательности решений задачи о расположении складов

5.4.2. Решатель судоку

В этом разделе мы продолжим демонстрировать возможности программирования Pyomo на Python. А именно мы выясним, разрешима ли заданная головоломка судоку, и если да, то найдем все ее решения. Мы решим задачу один раз, затем добавим целочисленный разрез, исключающий найденное решение из списка возможных, и решим задачу снова.

Типичная головоломка судоку показана на рис. 5.2. Требуется заполнить пустые клетки числами от 1 до 9 с соблюдением следующих ограничений. В каждой строке и в каждом столбце каждое число должно встречаться ровно один раз. Кроме того, в каждом из девяти квадратов, на которые разбивается большой квадрат, числа от 1 до 9 тоже должны встречаться по одному разу. Определим множества *ROWS*, *COLS* и *VALUES* (все они содержат целые числа от 1 до 9). Определим двоичную переменную $y[r, c, v]$, показывающую, какое число находится в каждой клетке. Если $y[r, c, v] = 1$, то в клетке на пересечении строки r и столбца c находится число v.

В этой нотации сравнительно просто определить ограничения на числа, допустимые в каждой строке и в каждом столбце:

$$\sum_{c \in COLS} y[r, c, v] = 1, \forall r \in ROWS, v \in VALUES;$$

$$\sum_{r \in ROWS} y[r, c, v] = 1, \forall c \in COLS, v \in VALUES.$$

В Pyomo код этих ограничений выглядит следующим образом:

```
# каждое число встречается ровно один раз в каждой строке
def _RowCon(model, r, v):
```

```
    return sum(model.y[r,c,v] for c in model.COLS) == 1
model.RowCon = pyo.Constraint(model.ROWS, model.VALUES, rule=_RowCon)

# каждое число встречается ровно один раз в каждом столбце
def _ColCon(model, c, v):
    return sum(model.y[r,c,v] for r in model.ROWS) == 1
model.ColCon = pyo.Constraint(model.COLS, model.VALUES, rule=_ColCon)
```

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

Рис. 5.2 ❖ Пример
еще не решенной головоломки судоку

Определить ограничения на числа в малых квадратах немного труднее. Чтобы упростить себе жизнь, определим множество, содержащее индексы этих квадратов. Затем определим список кортежей, который описывает отображение квадратов на список соответствующих индексов. Этот список вместе с соответствующим ограничением на квадраты определен в полном листинге в конце раздела. Нужное нам ограничение для малых квадратов выглядит так:

$$\sum_{(r,c)\in ss_{map}[i]} y[r, c, v] = 1, \forall i \in SUBSQUARES.$$

А так выглядит код Pyomo для этого ограничения:

```
# ровно одно число в каждом малом квадрате
def _SqCon(model, s, v):
    return sum(model.y[r,c,v] for (r,c) in n
        subsq_to_row_col[s]) == 1
model.SqCon = pyo.Constraint(model.SUBSQUARES, model.VALUES, rule=_SqCon)
```

И последнее ограничение в судоку – в каждой клетке может находиться только одно число. Вот как оно записывается:

$$\sum_{v\in VALUES} y[r, c, v] = 1, \forall r \in ROWS, c \in COLS.$$

А вот его код:

```
# ровно одно число в каждой клетке
def _ValueCon(model, r, c):
    return sum(model.y[r,c,v] for v in model.VALUES) == 1
model.ValueCon = pyo.Constraint(model.ROWS, model.COLS, rule=_ValueCon)
```

При создании головоломок судоку часто изменяются два свойства: начальная конфигурация доски и число целочисленных разрезов для удаления ранее встретившихся решений. Один из способов разобраться с этим разнообразием потенциальных входных данных заключается в том, чтобы определить функцию, которая создает модель по начальной конфигурации и списку целочисленных разрезов. Но такой функции было бы недостаточно для наших целей, потому что пришлось бы создавать новую модель всякий раз, как добавляется новый целочисленный разрез после очередного решения. Поэтому мы определим две разные функции: одна создает начальную модель по доске судоку, а другая добавляет целочисленный разрез в данную модель на основе текущих значений ее переменных.

Мы определим целочисленный разрез с помощью двух множеств. Первое множество S_0 состоит из индексов переменных, которые в текущем решении равны 0, а второе множество S_1 – из индексов переменных, которые в текущем решении равны 1. Зная эти два множества, можно следующим образом записать ограничение целочисленного разреза, которое помешает повторному нахождению этого решения:

$$\sum_{(r,c,v)\in S_0} y[r, c, v] + \sum_{(r,c,v)\in S_1} (1 - y[r, c, v]) \geqslant 1.$$

В приведенном ниже коде на Python определены три функции. Первая, `create_sudoku_model`, создает модель Pyomo для задачи судоку. Вторая, `add_integer_cut`, создает целочисленный разрез, соответствующий текущему решению, и добавляет его в компонент `ConstraintList`, который мы назвали `IntegerCuts`. Третья, `print_solution`, печатает текущее решение в форме доски судоку.

```
import pyomo.environ as pyo

# создать стандартный словарь python для отображения малых квадратов
# в список записей (row,col)
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)]
```

```
# создает модель судоку для доски 9×9, которая представлена
# списком кортежей фиксированных чисел (row, col, val).
def create_sudoku_model(board):

    model = pyo.ConcreteModel()

    # сохранить начальную доску в модели
    model.board = board

    # создать множества для индексирования строк, столбцов и квадратов
    model.ROWS = pyo.RangeSet(1,9)
    model.COLS = pyo.RangeSet(1,9)
    model.SUBSQUARES = pyo.RangeSet(1,9)
    model.VALUES = pyo.RangeSet(1,9)

    # создать двоичные переменные для определения значений
    model.y = pyo.Var(model.ROWS, model.COLS, model.VALUES, within=pyo.Binary)

    # зафиксировать переменные на основе текущей доски
    for (r,c,v) in board:
        model.y[r,c,v].fix(1)

    # создать целевую функцию; это задача о разрешимости,
    # поэтому сделаем ее постоянной
    model.obj = pyo.Objective(expr= 1.0)

    # в каждой строке должно быть по одному числу от 1 до 9
    def _RowCon(model, r, v):
        return sum(model.y[r,c,v] for c in model.COLS) == 1
    model.RowCon = pyo.Constraint(model.ROWS, model.VALUES, rule=_RowCon)

    # в каждом столбце должно быть по одному числу от 1 до 9
    def _ColCon(model, c, v):
        return sum(model.y[r,c,v] for r in model.ROWS) == 1
    model.ColCon = pyo.Constraint(model.COLS, model.VALUES, rule=_ColCon)

    # в каждом малом квадрате должно быть по одному числу от 1 до 9
    def _SqCon(model, s, v):
        return sum(model.y[r,c,v] for (r,c) in subsq_to_row_col[s]) == 1
    model.SqCon = pyo.Constraint(model.SUBSQUARES, model.VALUES, rule=_SqCon)

    # в каждой клетке должно быть ровно одно число
    def _ValueCon(model, r, c):
        return sum(model.y[r,c,v] for v in model.VALUES) == 1
    model.ValueCon = pyo.Constraint(model.ROWS, model.COLS, rule=_ValueCon)

    return model

# эта функция служит для добавления в модель нового целочисленного разреза
def add_integer_cut(model):
    # добавить компонент ConstraintList для хранения разрезов IntegerCut,
    # если его еще не существует
    if not hasattr(model, "IntegerCuts"):
        model.IntegerCuts = pyo.ConstraintList()
```

```
    # добавить целочисленный разрез в текущее решение модели
    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:
                    # проверить, включена или выключена двоичная переменная
                    # отметим, что она может быть равна не в точности 1
                    if pyo.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)

# печатает текущее решение, хранящееся в модели
def print_solution(model):
    for r in model.ROWS:
        print(' '.join(str(v) for c in model.COLS
              for v in model.VALUES
              if pyo.value(model.y[r,c,v]) >= 0.5))
```

Ниже показан скрипт, который управляет процессом оптимизации, вызывая три представленные выше функции. Он определяет начальную доску и итеративно решает задачи судоку, добавляя целочисленные разрезы, пока не получится задача, не имеющая решения. Задача считается неразрешимой, если решатель завершился не по условию `optimal`.

```
from pyomo.opt import (SolverFactory, TerminationCondition)
from sudoku import (create_sudoku_model,
                    print_solution,
                    add_integer_cut)

# определить доску
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)
```

На выходе этого скрипта печатаются все возможные решения. В данном случае найдено только одно решение, показанное на рис. 5.3.

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

Рис. 5.3 ❖ Решенная головоломка судоку

Глава 6

Взаимодействие с решателями

В этой главе описывается интерфейс Pyomo с решателями. Базовая функциональность, поддерживаемая всеми интерфейсами, включает преобразование экземпляра модели Pyomo в формат, ожидаемый решателем, задание параметров решателя, вызов решателя, проверку состояния и загрузку решения.

6.1. Введение

На рис. 6.1 показано верхнеуровневое представление связи между Pyomo и программой оптимизации, которую мы будем называть решателем. Pyomo – это средство моделирования, и решатели в него не входят. Вместо этого Pyomo поддерживает несколько интерфейсов с решателями. Как показано на рисунке, Pyomo преобразует модель в формат, понятный решателю. Результаты решателя используются для присваивания значений объектам `Var` в модели Pyomo, а кроме того, возвращается объект `Results`, от которого можно получить информацию о работе решателя. Решателю можно передать дополнительную информацию (например, параметры) и получить от него дополнительную информацию (например, двойственные значения), но на рисунке это не показано.

Модели Pyomo можно анализировать с помощью различных решателей, а в Pyomo имеется несколько типов интерфейсов с ними.

- *Внешний решатель* запускается как отдельный подпроцесс путем выполнения исполняемого файла, путь к которому прописан в переменной окружения `PATH`. Для взаимодействия с такими решателями Pyomo использует файлы. Pyomo генерирует файл, содержащий описание задачи, запускает решатель и загружает результаты из журналов и стандартного выхода. Это самая распространенная форма решателя.
- *Прямой решатель* выполняется как подпрограмма. Интерфейс с такими решателями реализуется с помощью установленных библиотек и име-

ет вид Python-пакетов. Такие решатели встречаются реже, потому что необходимо иметь интерфейс между библиотеками решателя и Python.

- *Хранимый решатель* тесно связан с прямым, но включает дополнительные возможности, позволяющие инкрементно модифицировать и повторно решать модель. Хранимые решатели обычно используются для повышения производительности в ситуациях, когда конструирование модели занимает много времени, или для сокращения времени работы можно воспользоваться известными решениями для родственного семейства моделей. Хранимые решатели обсуждаются более подробно в главе 9.

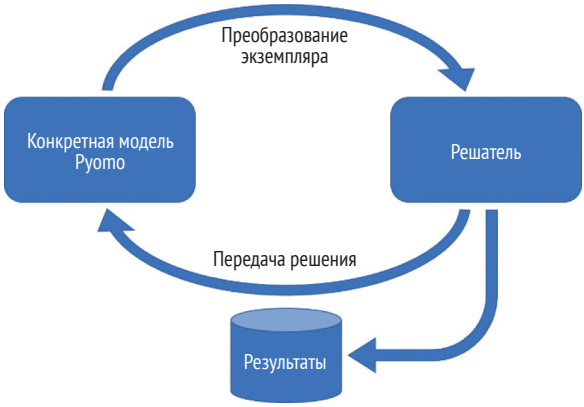

Рис. 6.1 ❖ Стилизованная картина типичных взаимодействий между Pyomo, обрабатывающей модель, и решателем, вычисляющим решения

Далее в этой главе обсуждается, как вызывать решатель, передавать ему параметры и получать информацию о работе решателя и результаты.

6.2. Использование решателей

Как показано в разделе 5.1, для конструирования объекта интерфейса с решателем используется функция `SolverFactory`, которой передается имя решателя. В большинстве случаев это имя исполняемого файла, который будет использован для решения задачи, однако для некоторых решателей Pyomo поддерживает и короткие имена. Например, решатель GLPK можно указать так:

```
solver = pyo.SolverFactory('glpk')
```

После того как объект решателя сконструирован, решатель можно вызвать методом `solve()`. Этот метод принимает ряд именованных аргументов, некоторые из которых описаны ниже, более-менее в порядке важности.

- `options`: словарь параметров, передаваемых решателю;
- `tee`: если этот аргумент равен `True`, то все, что печатает решатель, направляется как на стандартный вывод, так и в файл журнала. Если же

он равен False (значение по умолчанию), то вывод сохраняется только в файле журнала, если решатель создает таковой;

- load_solutions: если этот аргумент равен True (значение по умолчанию), то решение автоматически переносится в объекты Var модели. Если же он равен False, то в объекте результатов хранится *исходное* представление решений, и в модель оно не передается. Его можно передать в модель, вызвав метод model.solutions.load from();
- logfile: имя файла, в котором хранится вывод внешнего решателя;
- solnfile: имя файла, в котором хранится решение внешнего решателя;
- timelimit: сколько секунд разрешено работать внешнему решателю, прежде чем он будет завершен (по умолчанию None);
- report_timing: если этот аргумент равен True, то решатель сообщает информацию о хронометраже (по умолчанию False);
- solver_io: задает альтернативный интерфейс с решателем, например solver_io='nl';
- suffixes: список суффиксов, экспортируемых решателю.

Атрибут options можно использовать для передачи решателю параметров. В примере ниже мы передаем именованный аргумент tee=True, чтобы Pyomo печатала трассу выполнения на экране. И кроме того, мы передаем два параметра, специфичных для GLPK (направлять вывод в файл журнала warehouse.log и выключить масштабирование). Отметим, что у некоторых параметров решателей нет значения (например, noscale), это просто флаги, включающие или выключающие определенное поведение. Для таких параметров значение в словаре должно быть равно None. Отметим, что параметры можно передать и непосредственно функции solve в виде словаря, но такие параметры не сохраняются между вызовами решателя.

```
import json
import pyomo.environ as pyo
from warehouse_model import create_wl_model

# загрузить данные из json-файла
with open('warehouse_data.json', 'r') as fd:
    data = json.load(fd)

# вызвать функцию создания модели
model = create_wl_model(data, P=2)

# создать решатель
solver = pyo.SolverFactory('glpk')

# параметры можно передать непосредственно решателю
solver.options['noscale'] = None
solver.options['log'] = 'warehouse.log'
solver.solve(model, tee=True)
model.y.pprint()

# параметры можно также передать при вызове solve
myoptions = dict()
myoptions['noscale'] = None
```

```
myoptions['log'] = 'warehouse.log'
solver.solve(model, options=myoptions, tee=True)
model.y.pprint()
```

6.3. Исследование решения

После того как модель решена, необходимо исследовать два аспекта решения. Первый – состояние решения, возвращенное решателем, второй – значения переменных и целевой функции. Состояние решения, в принципе, можно посмотреть на консоли, передав команде `solve` параметр `tee=True`, но часто требуется получить его в программе. В этом разделе мы обсудим объект `results` и покажем, как получить значения переменных после решения.

6.3.1. Результаты решателя

Метод `solve()` возвращает объект `results`, содержащий информацию решателя. Если задача была решена успешно, то значения решения загружаются непосредственно в модель. Этот процесс состоит из трех шагов: (1) сохранение решений в атрибуте `solutions` модели, (2) загрузка значений переменных из выбранного решения и (3) удаление решений из объекта `results`. После этого объект `results` содержит только метаданные о модели и процессе оптимизации. Чтобы уменьшить потребление памяти, он по умолчанию уже не содержит самого решения.

> **ПРИМЕЧАНИЕ** По умолчанию, когда решатель завершает работу, решение автоматически загружается в объект модели и удаляется из объекта `results`. Поэтому объект `results` будет показывать, что содержит 0 решений.

Как правило, решатель возвращает только одно решение, но бывают случаи, когда решений несколько (пул решений). Поэтому объект `results` поддерживает интерфейс, похожий на словарь списков, содержащий более одного решения. Однако в самом распространенном случае, когда решение одно, объект `results` поддерживает простой интерфейс, напоминающий атрибуты. Объект `results`, возвращенный методом `solve()`, имеет атрибуты `problem` и `solver`, которые содержат информацию о статистике задачи и состоянии решения.

Атрибут `results.solver` содержит объект `SolverInformation`, основные атрибуты которого показаны в табл. 6.1.

Как отмечалось в разделе 2.5, самое простое, что можно сделать с объектом `results`, – передать его функции `assert_optimal_termination`, которая завершает скрипт и выводит сообщение, если решатель не нашел оптимального решения. Если скрипт не должен останавливаться в таком случае, но проверка все равно необходима, объект `results` можно передать функции `check_optimal_termination`, которая возвращает `True`, если решатель нашел оптимальное решение, и `False` в противном случае.

Таблица 6.1. Основные атрибуты объекта `SolverInformation`

Атрибут	Пояснение
`status`	Возвращает состояние решателя в виде элемента перечисления `SolverStatus`: `ok`, `warning`, `error`, `aborted` или `unknown`
`termination_condition`	Возвращает причину завершения, сообщенную решателем, в виде элемента перечисления `TerminationCondition`, который может принимать, в частности, значения `optimal`, `infeasible` и `unbounded`. Возможно много других причин завершения, и их состав зависит от решателя
`termination_message`	Возвращенная решателем строка, содержащая краткое описание состояния завершения

В некоторых скриптах имеет смысл отложить перенос решения из объекта `results` в модель до момента проверки оптимальности. Иногда это делается ради эффективности или чтобы избежать загрузки неоптимальных значений переменных в модель. Чтобы предотвратить автоматическую загрузку решения в модель, передайте методу `solve()` аргумент `load_solutions=False`. Чтобы переместить значения переменных решения из объекта `results` в объекты `Var` модели, вызовите метод `model.solutions.load_from(results)`, который пользуется объектом `solutions`, автоматически присоединяемым к объекту `model` при передаче методу `solve`. Ниже показан пример этих шагов.

```
from pyomo.opt import SolverStatus, TerminationCondition
# Отложить загрузку решения в модель до проверки
# состояния решателя
results = solver.solve(model, load_solutions=False)
if (results.solver.status == SolverStatus.ok) and n
    (results.solver.termination_condition == n
    TerminationCondition.optimal):
    # Вручную загрузить решение в модель
    model.solutions.load_from(results)
else:
    print("Solve failed.")
```

Часть II

ДОПОЛНИТЕЛЬНЫЕ ТЕМЫ

Глава 7

Нелинейное программирование в Pyomo

В этой главе описываются средства нелинейного программирования в Pyomo. Рассматриваются поддерживаемые нелинейные выражения и функции, даются рекомендации по формулированию и решению задач нелинейного программирования. Также приводятся иллюстрирующие примеры, взятые из практики. Наконец, кратко обсуждаются поддерживаемые решатели нелинейных задач.

7.1. Введение

Многие задачи невозможно адекватно представить без моделирования нелинейных связей. К счастью, в Pyomo имеется возможность естественно представить задачи нелинейного программирования (НЛП) общего вида. Однако при решении таких задач возникают проблемы, которые для линейных задач отсутствуют. Например, в большинстве современных эффективных НЛП-решателей необходимо вычислять производные ограничений и целевой функции. Поскольку функции нелинейны, требуется производить точные численные вычисления производных. Кроме того, для невыпуклых задач может существовать несколько локальных минимумов из-за формы целевой функции или ограничений, поэтому очень важно задать подходящую начальную точку.

В разделе 7.2 мы опишем нелинейные выражения, поддерживаемые Pyomo, а затем покажем, как формулируется нелинейная задача. В разделе 7.3 мы кратко обсудим, какие решатели поддерживает Pyomo, и дадим несколько советов, как эффективно формулировать задачи нелинейного программи-

рования. И завершим главу несколькими небольшими, но реальными примерами нелинейного программирования.

7.2. Задачи нелинейного программирования в Pyomo

Pyomo поддерживает следующую общую постановку задачи нелинейного программирования:

$$\begin{aligned} & \min_x f(x) \\ \text{при условиях } & c(x) = 0 \\ & d^L \leqslant d(x) \leqslant d^U \\ & x^L \leqslant x \leqslant x^U \end{aligned}$$

Допустимая форма целевой функции $f(x)$, вектора ограничений в виде равенства $c(x)$ и вектора ограничений в виде неравенства $d(x)$ зависит от выбранного решателя. Однако Pyomo тщательно протестирована совместно с локальными и глобальными решателями, которые, как правило, предполагают, что эти функции непрерывные и гладкие, т. е. имеют непрерывные первые (и, возможно, вторые) производные. Разработка нелинейных расширений для Pyomo сосредоточена на этом широком классе задач.

7.2.1. Нелинейные выражения

Формулирование задач нелинейной оптимизации в Pyomo не отличается от формулирования линейных или смешанно-целочисленных задач. Все рассмотренные выше компоненты моделирования (например, `Objective`, `Constraint`) используются точно так же, только могут включать нелинейные выражения.

В табл. 7.1 перечислены операторы, поддерживаемые в выражениях (в примерах `x` и `y` – объекты Pyomo типа `Var`). Помимо этих операторов, Pyomo поддерживает ряд нелинейных функций, перечисленных в табл. 7.2. Отметим, что это функции, написанные специально для Pyomo, а нелинейные функции из других библиотек в выражениях не поддерживаются.

> **ПРИМЕЧАНИЕ** Передача компонентов Pyomo (например, `Var`, `Param`) нелинейным функциям из других Python-пакетов (например, `math` или `numpy`) при создании выражения Pyomo приведет к исключению, отлаживать которое трудно. Этой ошибки можно избежать, если всегда передавать именованные предложения импорта и явно указывать, откуда берется нелинейная функция.

```
import pyomo.environ as pyo # использовать в выражениях Pyomo
import math as mt           # pyo.sin а не mt.sin
```

Таблица 7.1. Операторы Python, переопределенные для выражений Pyomo

Операция	Оператор	Пример
умножение	*	`expr = model.x * model.y`
деление	/	`expr = model.x / model.y`
возведение в степень	**	`expr = (model.x+2.0) ** model.y`
умножение на месте[1]	*=	`expr *= model.x`
деление на месте[2]	/=	`expr /= model.x`
возведение в степень на месте[3]	**=	`expr **= model.x`

[1] Этот пример умножения на месте эквивалентен `expr = expr * model.x`.
[2] Этот пример деления на месте эквивалентен `expr = expr / model.x`.
[3] Этот пример возведения в степень на месте эквивалентен `expr = expr ** model.x`.

Таблица 7.2. Функции, поддерживаемые Pyomo в определениях нелинейных выражений. Предполагается, что Pyomo импортирована предложением `import pyomo.environ as pyo`

Операция	Функция	Пример
арккосинус	`acos`	`expr = pyo.acos(model.x)`
гиперболический арккосинус	`acosh`	`expr = pyo.acosh(model.x)`
арксинус	`asin`	`expr = pyo.asin(model.x)`
гиперболический арксинус	`asinh`	`expr = pyo.asinh(model.x)`
арктангенс	`atan`	`expr = pyo.atan(model.x)`
гиперболический арктангенс	`atanh`	`expr = pyo.atanh(model.x)`
косинус	`cos`	`expr = pyo.cos(model.x)`
гиперболический косинус	`cosh`	`expr = pyo.cosh(model.x)`
экспонента	`exp`	`expr = pyo.exp(model.x)`
натуральный логарифм	`log`	`expr = pyo.log(model.x)`
десятичный логарифм	`log10`	`expr = pyo.log10(model.x)`
синус	`sin`	`expr = pyo.sin(model.x)`
квадратный корень	`sqrt`	`expr = pyo.sqrt(model.x)`
гиперболический синус	`sinh`	`expr = pyo.sinh(model.x)`
тангенс	`tan`	`expr = pyo.tan(model.x)`
гиперболический тангенс	`tanh`	`expr = pyo.tanh(model.x)`

7.2.2. Задача Розенброка

В этом разделе мы представим небольшой пример для иллюстрации формулирования и решения нелинейной модели Pyomo. Мы рассматриваем безусловную минимизацию функции Розенброка от двух переменных – классическую задачу, которую часто приводят как пример алгоритма безусловной нелинейной оптимизации (см., например, [45]). Задача ставится следующим образом:

$$\min_{x,y} f(x, y) = (1 - x)^2 + 100(y - x^2)^2,$$

а ее решение – точка в нижней части банановидной долины с координатами $x = 1$ и $y = 1$ (см. рис. 7.1).

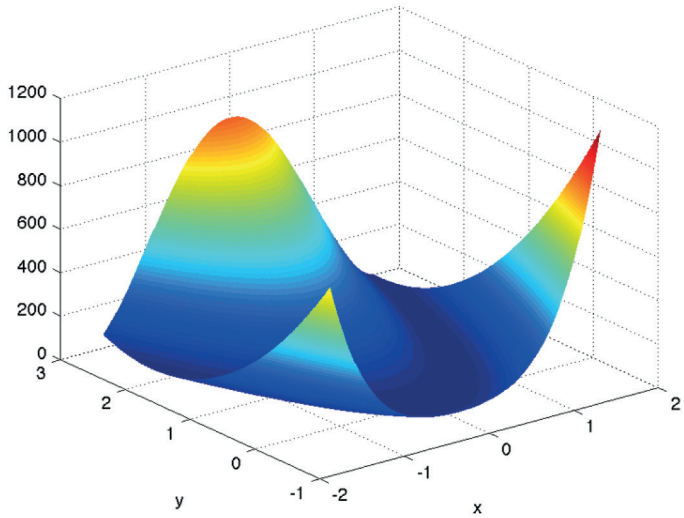

Рис. 7.1 ❖ График поверхности функции Розенброка $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$. Минимум достигается в нижней точке банановидной долины с координатами $x = 1$ и $y = 1$

Рассмотрим следующую модель Pyomo для этой задачи:

```
# rosenbrock.py
# Модель Pyomo для задачи Розенброка
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(model):
    return (1.0 - model.x)**2 + 100.0*(model.y - model.x**2)**2
model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

status = pyo.SolverFactory('ipopt').solve(model)
pyo.assert_optimal_termination(status)
model.pprint()
```

Этот пример показывает, что определение нелинейной модели ничем не отличается от определения линейной. Модель создает две переменные x и y и инициализирует каждую значением 1.5. Отметим, что не нужно как-то указывать, что переменная впоследствии встретится в нелинейном выражении; Pyomo сама установит это перед решением задачи. Правило конструирования целевой функции просто возвращает нелинейное выражение. Затем

для решения задачи используется нелинейный решатель IPOPT, после чего проверяется состояние решателя и печатается результат.

> **ПРИМЕЧАНИЕ** *Pyomo должна работать с любым решателем на основе AMPL или GAMS.* Поэтому для решения задач Pyomo можно использовать многочисленные коммерческие и открытые пакеты. В этой главе используется IPOPT [34], пакет нелинейной оптимизации с открытым исходным кодом.

Решение задачи Розенброка можно запустить следующей командой:

```
python rosenbrock.py
```

В результате печатается:

```
2 Var Declarations
    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

1 Objective Declarations
    obj : Size=1, Index=None, Active=True
        Key : Active : Sense :   Expression
        None : True : minimize : (1.0 - x)**2 + 100.0*(y - x**2)**2

3 Declarations: x y obj
```

Здесь мы видим, что задача решена и найдено оптимальное значение $x = y = 1.0$, при котором целевая функция равна нулю. В этом примере есть только одна нелинейная целевая функция и две скалярные переменные, но, вообще говоря, можно использовать также компоненты, рассмотренные в предыдущих главах.

7.3. Решение задач нелинейного программирования

В большинстве случаев, перед тем как использовать Pyomo для решения задачи нелинейной оптимизации, необходимо установить подходящий нелинейный решатель. Средства Pyomo ориентированы на моделирование задач оптимизации, и существует не так уж много решателей, непосредственно интегрированных с Pyomo.

7.3.1. Нелинейные решатели

Для решателей задач нелинейного программирования требуется, чтобы каркас моделирования вычислял целевую функцию и ограничения в точках-кандидатах *x*. Кроме того, многие нелинейные решатели требуют также вычисления первой, а зачастую и второй производной в точках-кандидатах. Однако пользователю Pyomo не нужно выполнять эти вычисления. Есть средства автоматического дифференцирования (АД), которые точно и эффективно вычисляют первую и вторую производные без вмешательства пользователя.

Pyomo предоставляет интерфейсы к нелинейным решателям, откомпилированные для AMPL или GAMS, и пользуется их средствами вычисления модели и автоматического дифференцирования для эффективного вычисления производных. Благодаря этим интерфейсам Pyomo непосредственно доступен широкий круг решателей и не требуется разрабатывать отдельные интерфейсы для каждого решателя.

В случае AMPL-решателей код самого решателя собран с интерфейсом к библиотеке AMPL Solver Library (ASL) [23], которая находится в открытом доступе. Поэтому есть немало решателей с открытым исходным кодом, доступных Pyomo через этот интерфейс. Интерфейс Pyomo-GAMS выполняет преобразование модели Pyomo в модель GAMS, и, следовательно, пользователь должен будет установить GAMS для работы с GAMS-решателями.

7.3.2. Дополнительные советы по нелинейному программированию

Эффективная постановка и решение задач нелинейного программирования могут оказаться гораздо труднее, чем в случае линейных задач. В этом разделе мы дадим несколько простых советов на эту тему.

Инициализация переменных

Решатели задач нелинейного программирования часто требуют инициализировать переменные задачи. Если начальные значения не заданы, то Pyomo предполагает, что они равны нулю. Однако во многих приложениях на такие значения по умолчанию полагаться нельзя.

В общем невыпуклом случае задачи нелинейного программирования могут иметь локальные решения, и часто так и происходит. Хотя в теории глобальной оптимизации (т. е. методов, дающих строгие гарантии глобальной оптимальности) достигнуты значительные успехи, крупномасштабные задачи общего вида зачастую остаются неразрешимыми, даже если применяются самые современные глобальные решатели. Следовательно, мы часто вынуждены довольствоваться решателем, дающим только гарантии локальной оптимальности. И тогда на первое место выходит инициализация задачи, гарантирующая сходимость к желательному локальному решению.

Иногда нежелательные локальные решения физически бессмысленны и разумной инициализации с разумными границами переменных достаточно, чтобы гарантировать надежную сходимость к желательному решению. А иногда бывает несколько физически осмысленных локальных решений. Для хорошей постановки нелинейной задачи часто необходимо приложить значительные усилия к разработке разумной стратегии инициализации.

Неопределенные вычисления

Некоторые нелинейные функции определены не всюду (например, $log(x)$ определен только при $x > 0$). Поэтому автор модели должен позаботиться о том, чтобы в формулировке задачи переменные были ограничены допустимой областью определения. Обычно для этого достаточно задать разумные границы и начальные значения переменных.

Также важно отметить, что многие нелинейные решатели используют информацию о первой (а иногда и о второй) производной целевой функции и ограничений. Поэтому важно наложить на переменные такие ограничения, чтобы были определены также производные нелинейных выражений. Например, если в выражение входит `sqrt(x)`, то границы $x \geqslant 0$ может быть недостаточно. Хотя функция $\sqrt{x}$ определена при $x = 0$, ее производная, $1/\sqrt{x}$, там не определена. Это следует учитывать при задании границ переменных.

Наконец, отметим, что некоторые нелинейные решатели, применяющие метод внутренней точки (например, IPOPT), могут немного ослаблять границы переменных перед началом решения задачи. Доказано, что во многих приложениях эта стратегия эффективна, но может приводить к выходу за границы области определения, даже если в модели заданы правильные границы переменных. В таком случае нужно либо запретить это поведение решателя, либо задать более консервативные границы.

Сингулярности модели и масштабирование задачи

Многие нелинейные решатели задают для ограничений условия (называемые квалификацией ограничений), которые должны удовлетворяться, чтобы можно было гарантировать сходимость. В частности, нередко бывает полезно делать ограничения независимыми друг от друга всюду в области определения решения (т. е. множество градиентов активных ограничений должно быть линейно независимо). В работе Nocedal and Wright [45] этот вопрос обсуждается более подробно (см. главу 12).

К сожалению, даже если эти условия чисто математически удовлетворяются для модели, все равно могут возникать проблемы при численном решении. Если модель плохо обусловлена, то многие решения не сходятся к решению или ищут его слишком долго. Важно масштабировать модель, так чтобы якобиан и гессиан были хорошо обусловлены. Иногда для этого достаточно линейного масштабирования переменных и ограничений. А в трудных случаях модель иногда приходится переформулировать.

7.4. Примеры нелинейного программирования

В этом разделе мы приведем несколько примеров, иллюстрирующих возможности Pyomo по решению нелинейных задач.

7.4.1. Инициализация переменных для мультимодальной функции

В следующем примере иллюстрируется важность эффективной инициализации переменных. Рассмотрим задачу о минимизации мультимодальной функции

$$f(x) = (2 - \cos(\pi x) - \cos(\pi y))x^2y^2,$$

имеющей несколько локальных минимумов. Показанная ниже модель Pyomo для этой задачи инициализирует переменные при $x = y = 0.25$.

```
# multimodal_init1.py
import pyomo.environ as pyo
from math import pi

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize = 0.25, bounds=(0,4))
model.y = pyo.Var(initialize = 0.25, bounds=(0,4))

def multimodal(m):
    return (2-pyo.cos(pi*m.x)-pyo.cos(pi*m.y)) * (m.x**2) * (m.y**2)
model.obj = pyo.Objective(rule=multimodal, sense=pyo.minimize)

status = pyo.SolverFactory('ipopt').solve(model)
pyo.assert_optimal_termination(status)
print(pyo.value(model.x), pyo.value(model.y))
```

Для решения этой задачи выполним команду

```
python multimodel_init1.py
```

IPOPT находит решение, близкое к нашей начальной точке $x = y = 0.0178$. Однако если изменить задачу и инициализировать переменные при $x = y = 2.1$,

```
model.x = pyo.Var(initialize = 2.1, bounds=(0,4))
model.y = pyo.Var(initialize = 2.1, bounds=(0,4))
```

то IPOPT найдет другое локальное решение в точке $x = y = 2.0$.

7.4.2. Оптимальные квоты для неистощительной добычи оленей

Для поддержания здоровой популяции оленей необходимо уделять внимание как естественной среде обитания, так и неистощительной стратегии добычи. У охотников наибольшим спросом пользуются лицензии на добычу взрослых самцов. Однако изъятие из популяции слишком большого числа самцов может ограничить рост популяции. Мы рассмотрим задачу нелинейного программирования, определяющую оптимальную политику добычи оленей, в которой целью является максимизация добычи при условии сохранения сильной и устойчивой популяции.

Рассмотрим адаптированную для Pyomo модель Bailey [5], описывающую динамику популяции оленей. Популяцию в данной местности можно разделить на три части: самцы, самки и детеныши. Кроме того, каждый год разбивается на четыре периода: зима, сезон размножения, лето и сезон охоты. Модель, описывающая динамику популяции, основана на следующих допущениях:

- предполагается, что каждую часть популяции можно представить непрерывными переменными (т. е. популяция достаточно велика, чтобы это приближение можно было считать хорошим);
- в каждом сезоне имеет место сокращение числа самцов, самок и детенышей. Это сокращение обусловлено естественными причинами и пропорционально размеру части популяции. Его можно учесть, задав дробный коэффициент выживания, зависящий от периода (зима, сезон размножения, лето, сезон охоты) и части популяции (самцы, самки, детеныши);
- новые детеныши рождаются каждый год во время периода размножения. Детеныши рождаются в результате спаривания самок и взрослых самцов, а коэффициент рождаемости зависит от количества доступного корма. Предполагается, что самцов и самок рождается поровну. Половина проживших год – самцы, половина – самки;
- полный годовой запас корма постоянный, это ограничение на управление средой обитания;
- вся добыча обусловлена охотой. Квоты могут быть установлены для каждой части популяции, и предполагается, что они выбираются полностью (т. е. все охотники успешны).

Полный вывод модели части популяции приведен в работе [5] и приводит к следующей системе разностных уравнений:

$$f_{y+1} = p_1 b r_y \left(\frac{p_2}{10} f_y + p_3 d_y \right) - h_y^f; \tag{7.1}$$

$$d_{y+1} = p_4 d_y + \frac{p_5}{2} f_y - h_y^d; \tag{7.2}$$

$$b_{y+1} = p_6 b_y + \frac{p_5}{2} f_y - h_y^b; \tag{7.3}$$

$$br_y = 1.1 + 0.8 \frac{p_s - c_y}{p_s}; \tag{7.4}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y, \tag{7.5}$$

где значения параметров p_1, ..., p_9 вычисляются, исходя из различных коэффициентов выживания и потребления корма. Эти значения приведены в табл. 7.3. Переменные f_y, d_y и b_y представляют число детенышей, самок и самцов в году y соответственно. Аналогично h_y^f, h_y^d и h_y^b – неизвестные числа добытых детенышей, самок и самцов в году y. Коэффициент рождаемости br_y для самок описывается нелинейным соотношением, в котором c_y – количество корма, потребленного оленем (в фунтах), а p_s – общий доступный запас корма (тоже в фунтах).

Таблица 7.3. Значения параметров для задачи о добыче оленей

Параметр	Значение	Параметр	Значение
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		

В оригинальной статье эта система разностных уравнений оптимизировалась на данных за 20 лет с целью вывести неистощительную устойчивую стратегию добычи на последующие годы. Здесь же мы включим только один год и добавим ограничение: числа детенышей, самок и самцов в год $y + 1$ равны соответствующим числам за год y. Это дает то же самое устойчивое решение в постановке с гораздо меньшим количеством данных.

Цель состоит в максимизации числа добытых оленей, что дает следующую постановку задачи нелинейного программирования:

$$\max w_b h_y^b + w_f h_y^f + w_d h_y^d; \tag{7.6}$$

$$f_y = p_1 br_y \left(\frac{p_2}{10} f_y + p_3 d_y \right) - h_y^f; \tag{7.7}$$

$$d_y = p_4 d_y + \frac{p_5}{2} f_y - h_y^d; \tag{7.8}$$

$$b_y = p_6 b_y + \frac{p_5}{2} f_y - h_y^b; \tag{7.9}$$

$$br_y = 1.1 + 0.8\frac{p_s - c_y}{p_s}; \tag{7.10}$$

$$c_y = p_7 b_y + p_8 d_y + p_9 f_y; \tag{7.11}$$

$$c_y \leqslant p_s; \tag{7.12}$$

$$b_y \geqslant \frac{1}{5}(0.4 f_y + d_y), \tag{7.13}$$

где w_f, w_d и w_b – количества добытых детенышей, самок и самцов соответственно. Как видно из табл. 7.3, количество лицензий на отстрел самцов в 10 раз больше, чем на отстрел самок и детенышей. Неравенство (7.12) гарантирует, что количество потребленного корма не может быть больше доступного запаса, что ограничивает общий размер популяции. Неравенство (7.13) гарантирует, что количество самцов достаточно для неистощительного размножения.

Следующая модель представляет задачу об оптимальной добыче оленей.

```
# DeerProblem.py
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.p1 = pyo.Param();
model.p2 = pyo.Param();
model.p3 = pyo.Param();
model.p4 = pyo.Param();
model.p5 = pyo.Param();
model.p6 = pyo.Param();
model.p7 = pyo.Param();
model.p8 = pyo.Param();
model.p9 = pyo.Param();
model.ps = pyo.Param();

model.f = pyo.Var(initialize = 20, within=pyo.PositiveReals)
model.d = pyo.Var(initialize = 20, within=pyo.PositiveReals)
model.b = pyo.Var(initialize = 20, within=pyo.PositiveReals)

model.hf = pyo.Var(initialize = 20, within=pyo.PositiveReals)
model.hd = pyo.Var(initialize = 20, within=pyo.PositiveReals)
model.hb = pyo.Var(initialize = 20, within=pyo.PositiveReals)

model.br = pyo.Var(initialize=1.5, within=pyo.PositiveReals)

model.c = pyo.Var(initialize=500000, within=pyo.PositiveReals)

def obj_rule(m):
    return 10*m.hb + m.hd + m.hf
model.obj = pyo.Objective(rule=obj_rule, sense=pyo.maximize)

def f_bal_rule(m):
    return m.f == m.p1*m.br*(m.p2/10.0*m.f + m.p3*m.d) - m.hf
```

```
model.f_bal = pyo.Constraint(rule=f_bal_rule)

def d_bal_rule(m):
    return m.d == m.p4*m.d + m.p5/2.0*m.f - m.hd
model.d_bal = pyo.Constraint(rule=d_bal_rule)

def b_bal_rule(m):
    return m.b == m.p6*m.b + m.p5/2.0*m.f - m.hb
model.b_bal = pyo.Constraint(rule=b_bal_rule)

def food_cons_rule(m):
    return m.c == m.p7*m.b + m.p8*m.d + m.p9*m.f
model.food_cons = pyo.Constraint(rule=food_cons_rule)

def supply_rule(m):
    return m.c <= m.ps
model.supply = pyo.Constraint(rule=supply_rule)

def birth_rule(m):
    return m.br == 1.1 + 0.8*(m.ps - m.c)/m.ps
model.birth = pyo.Constraint(rule=birth_rule)

def minbuck_rule(m):
    return m.b >= 1.0/5.0*(0.4*m.f + m.d)
model.minbuck = pyo.Constraint(rule=minbuck_rule)

# создать экземпляр ConcreteModel
instance = model.create_instance('DeerProblem.dat')
status = pyo.SolverFactory('ipopt').solve(instance)
pyo.assert_optimal_termination(status)

instance.pprint()
```

В следующем файле данных представлены параметры из табл. 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;
```

Для оптимизации модели следует выполнить команду

```
python DeerProblem.py
```

Оптимальное решение – добыть 62 самца, 37 самок и ни одного детеныша. Это решение отдает предпочтение добыче самцов, но если добыть их слишком много, то это отразится на размере популяции. Невязка для ограничения

`minbuck` равна нулю, т. е. это неравенство обращается в равенство для решения и, следовательно, ограничивает количество разрешенных к добыче самцов.

Очевидно, что это решение является функцией значений параметров, которые определяют количества детенышей, самок и самцов в целевой функции, а также параметров модели популяционной динамики. Поскольку Pyomo основана на Python, нетрудно написать скрипт, который будет определять оптимальное решение в виде функции от значений других параметров, что позволит более глубоко проанализировать систему. В главе 5 эта функциональность обсуждается более подробно.

7.4.3. Оценка моделей инфекционных заболеваний

Широкое распространение программ вакцинации значительно уменьшило детскую заболеваемость. Однако инфекционные детские болезни попрежнему остаются серьезной проблемой в развивающихся странах, а появление новых штаммов бактерий и вирусов ставит вопросы перед лицами, ответственными за здравоохранение. В этом примере мы смоделируем вспышку инфекционного заболевания в небольшом сообществе из 300 человек (например, в небольшой школе). Мы построим базовую модель, описывающую распространение инфекции в популяции, и используем нелинейное программирование, чтобы оценить ключевые параметры модели на имитированных данных.

Для представления системы будем использовать стандартную камерную модель с дискретным временем. Люди распределены по трем камерам в зависимости от их отношения к заболеванию: предрасположенные (S), инфицированные (I) или выздоровевшие (R). Мы предполагаем, что человек, который заразился и выздоровел, получает иммунитет к заболеванию (т. е. не возвращается в пул предрасположенных). Модель системы с дискретным временем описывается следующими уравнениями:

$$I_i = \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N};$$

$$S_i = S_{i-1} - I_i.$$

Эти два разностных уравнения описывают распространение заболевания в популяции. Поскольку модель поколенческая, предполагается, что все лица, инфицированные в момент i, выздоравливают к моменту $i + 1$. I_i и S_i – число заразившихся и предрасположенных лиц в момент i соответственно. Размер популяции задается числом N, а β и α – параметры модели.

> **ПРИМЕЧАНИЕ** Обычно *параметры* задачи оптимизации считаются фиксированными данными. Но в этом примере параметры модели инфекционного заболевания неизвестны, и мы хотим оценить их по имеющимся данным. В результате параметры модели β и α становятся в модели переменными Pyomo (поскольку они подлежат оценке в процессе оптимизации).

В этом примере мы воспользуемся методом наименьших квадратов, чтобы оценить параметры по имитированным данным. Обозначим *SI* множество индексов последовательных интервалов. В нашем примере оценивание производится на данных за один год, состоящий из 26 двухнедельных интервалов. Сведения о случаях заболевания (известные входные данные) задаются в виде C_i, а переменные ε_i^I – разности между измеренными и вычисленными значениями. Полная формулировка задачи имеет вид:

$$\min \sum_{i \in SI} (\varepsilon_i^I)^2$$

$$I_i = \frac{\beta I_{i-1}^{\alpha} S_{i-1}}{N} \quad \forall i \in SI \setminus \{1\}$$

$$S_i = S_{i-1} - I_i \quad \forall i \in SI \setminus \{1\}$$

$$C_i = I_i + \varepsilon_i^I$$

$$0 \leqslant I_i, S_i \leqslant N$$

$$0.5 \leqslant \beta \leqslant 70$$

$$0.5 \leqslant \alpha \leqslant 1.5$$

В следующем листинге показана абстрактная модель для этой нелинейной задачи оценивания методом наименьших квадратов.

```
# disease_estimation.py
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.S_SI = pyo.Set(ordered=True)

model.P_REP_CASES = pyo.Param(model.S_SI)
model.P_POP = pyo.Param()

model.I = pyo.Var(model.S_SI, bounds=(0,model.P_POP), initialize=1)
model.S = pyo.Var(model.S_SI, bounds=(0,model.P_POP), initialize=300)
model.beta = pyo.Var(bounds=(0.05, 70))
model.alpha = pyo.Var(bounds=(0.5, 1.5))
model.eps_I = pyo.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 = pyo.Objective(rule=_objective, \
    sense=pyo.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 pyo.Constraint.Skip

model.InfDynamics = pyo.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 pyo.Constraint.Skip
model.SusDynamics = pyo.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 = pyo.Constraint(model.S_SI, rule=_Data)

# создать экземпляр ConcreteModel
instance = model.create_instance('disease_estimation.dat')
status = pyo.SolverFactory('ipopt').solve(instance)
pyo.assert_optimal_termination(status)
print(' ***')
print(' *** Optimal beta Value: %.2f' % pyo.value(instance.beta))
print(' *** Optimal alpha Value: %.2f' % pyo.value(instance.alpha))
print(' ***')
```

Файл данных Pyomo для экземпляра этой модели приведен ниже:

```
# 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
;
```

Для решения задачи оценивания выполним команду

```
python disease_estimation.py
```

которая дает следующий результат:

```
***
*** Optimal beta Value: 1.99
*** Optimal alpha Value: 1.00
***
```

При генерировании данных мы использовали значения $\beta = 2$ и $\alpha = 1$, так что полученные результаты выглядят вполне разумно.

7.4.4. Проектирование реактора

Химические реакторы обычно являются самым важным оборудованием на химическом заводе. Есть много разных видов реакторов, но самые распространенные идеализации – реактор с постоянным перемешиванием (CSTR) и реактор идеального вытеснения. CSTR часто используют в исследованиях моделей, его можно смоделировать как систему с сосредоточенными параметрами. В этом примере мы рассмотрим схему реакции ван де Вуссе:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$

$$2A \xrightarrow{k_3} D$$

Диаграмма системы показана на рис. 7.2, где F – объемная скорость потока. Предполагается, что реактор заполнен до постоянного объема, а смесь имеет постоянную плотность, поэтому объемная скорость входящего потока равна объемной скорости исходящего потока. Так как предполагается, что содержимое реактора хорошо перемешивается, концентрации в реакторе эквивалентны концентрациям каждого исходящего из реактора компонента и равны c_A, c_B, c_C и c_D.

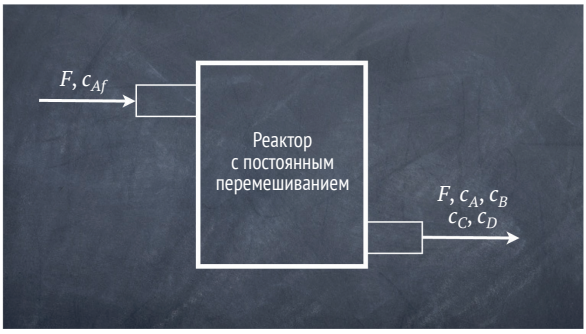

Рис. 7.2 ❖ Реактор с постоянным перемешиванием, производящий нужный продукт B и побочные продукты C и D из реагента A

Рассмотрим следующую задачу о реакторе, взятую из работы Bequette [7]. Цель – произвести продукт B из исходного сырья, содержащего реагент A. Если реактор будет слишком маленьким, то мы получим недостаточное преобразование A в требуемый продукт B. Однако если при данной схеме реакции реактор будет слишком большим (т. е. будет реагировать сразу много вещества), то значительный выход нужного продукта B подвергнется дальнейшей реакции с выходом нежелательного побочного продукта C. Поэтому

цель данного упражнения – найти оптимальный объем реактора, дающего на выходе максимальную концентрацию продукта B.

Стационарные мольные балансы каждого из четырех компонентов равны

$$0 = \frac{F}{V}c_{Af} - \frac{F}{V}c_A - k_1 c_A - 2k_3 c_A^2;$$

$$0 = -\frac{F}{V}c_B + k_1 c_A - k_2 c_B;$$

$$0 = -\frac{F}{V}c_C + k_2 c_B;$$

$$0 = -\frac{F}{V}c_D + k_3 c_A^2.$$

Известные параметры системы равны

$$c_{Af} = 10\,000\ \frac{\text{г-моль}}{\text{м}};\quad k_1 = \frac{5}{6}\ \text{мин}^{-1};\quad k_2 = \frac{5}{3}\ \text{мин}^{-1};\quad k_3 = \frac{1}{6000}\ \frac{\text{м}^3}{\text{г-моль min}}.$$

Поскольку объемная скорость потока F всегда встречается в числителе дроби, знаменателем которой является объем реактора V, принято рассматривать это отношение как одну переменную, которая называется объемной скоростью (sv). В нашей задаче оптимизации требуется найти такую объемную скорость, которая максимизирует концентрацию нужного продукта B на выходе.

Ниже показана функция, которая строит конкретную модель задачи о проектировании реактора, а также код, решающий эту задачу путем непосредственного выполнения Python-файла:

```
import pyomo.environ
import pyomo.environ as pyo

def create_model(k1, k2, k3, caf):
    # создать конкретную модель
    model = pyo.ConcreteModel()

    # создать переменные
    model.sv = pyo.Var(initialize=1.0, within=pyo.PositiveReals)
    model.ca = pyo.Var(initialize=5000.0, within=pyo.PositiveReals)
    model.cb = pyo.Var(initialize=2000.0, within=pyo.PositiveReals)
    model.cc = pyo.Var(initialize=2000.0, within=pyo.PositiveReals)
    model.cd = pyo.Var(initialize=1000.0, within=pyo.PositiveReals)

    # создать целевую функцию
    model.obj = pyo.Objective(expr = model.cb, sense=pyo.maximize)

    # создать ограничения
    model.ca_bal = pyo.Constraint(expr = (0 == model.sv * caf \
                - model.sv * model.ca - k1 * model.ca \
                - 2.0 * k3 * model.ca ** 2.0))

    model.cb_bal = pyo.Constraint(expr=(0 == -model.sv * model.cb \
```

```
                + k1 * model.ca - k2 * model.cb))

    model.cc_bal = pyo.Constraint(expr=(0 == -model.sv * model.cc \
                + k2 * model.cb))

    model.cd_bal = pyo.Constraint(expr=(0 == -model.sv * model.cd \
                + k3 * model.ca ** 2.0))

    return model

if __name__ =='__main__':
    # решить один экземпляр задачи
    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

    m = create_model(k1, k2, k3, caf)
    status = pyo.SolverFactory('ipopt').solve(m)
    pyo.assert_optimal_termination(status)
    m.pprint()
```

Для решения задачи следует выполнить команду

```
python ReactorDesign.py
```

Оптимальная объемная скорость равна 1.34, при этом получается выходная концентрация B, равная 1072.

Легко сконструировать модель и выполнить для нее скрипт. Например, если бы мы хотели решить эту задачу проектирования для разных значений концентрации исходного вещества, то могли бы воспользоваться следующим кодом:

```
import pyomo.environ as pyo
from ReactorDesign import create_model

# задать данные (встроенных в Python типов)
k1 = 5.0/6.0 # min^-1
k2 = 5.0/3.0 # min^-1
k3 = 1.0/6000.0 # m^3/(gmol min)

# решить модель для различных значений caf и вывести результаты
print('{:>10s}\t{:>10s}\t{:>10s}'.format('CAf', 'SV', 'CB'))
for cafi in range(1,11):
    caf = cafi*1000.0 # gmol/m^3

    # создать модель с новыми данными
    # отметим, что это можно было бы сделать более эффективно
    # с изменяемыми параметрами
    m = create_model(k1, k2, k3, caf)

    # решить задачу
    status = pyo.SolverFactory('ipopt').solve(m)
    print("{:10g}\t{:10g}\t{:10g}".\
        format(caf, pyo.value(m.sv), pyo.value(m.cb)))
```

В результате выполнения этого скрипта получаются следующие результаты:

CAf	SV	CB
1000	1.21294	157.564
2000	1.23903	294.346
3000	1.25993	416.943
4000	1.27729	529.051
5000	1.29209	632.993
6000	1.30495	730.339
7000	1.31629	822.212
8000	1.32641	909.447
9000	1.33553	992.687
10 000	1.34381	1072.44

Этот пример демонстрирует скриптовые возможности Pyomo. Дополнительные примеры скриптов и более подробное описание этих возможностей см. в главе 5.

Глава 8

Структурное моделирование с помощью блоков

В этой главе описывается построение иерархически структурированных моделей с помощью компонента Pyomo `Block`. Во многих моделях отчетливо прослеживается иерархическая структура, т. е. они состоят из повторяющихся групп концептуально связанных компонентов моделирования. Pyomo позволяет определить фундаментальные структурные элементы, а затем сконструировать задачу в целом, соединив эти элементы объектно ориентированным способом. В этой главе мы опишем фундаментальный компонент `Block` и типичные примеры его использования, в т. ч. повторяющиеся компоненты и управление областью действия модели.

8.1. Введение

Решатели обычно ожидают получить модель в стандартизированной форме. Например, линейные решатели принимают модели в такой стандартной форме:

$$\begin{aligned} &\min && c^T x \\ &\text{при условиях} && Ax \leqslant b \\ & && x \geqslant 0 \end{aligned}$$

Здесь переменные собраны в один вектор x, а ограничения представлены в матричной форме. Хотя эта форма удобна для алгоритмов, работающих с этими матрицами непосредственно, автору модели нелегко генерировать ее, отлаживать и работать с ней. Языки алгебраического моделирования (AML) решают эту проблему в лоб, позволяя авторам давать различающиеся имена компонентам моделирования (например, переменным или ограни-

чениям) и определять модель над множеством индексов. Поскольку модели часто состоят из повторяющихся математических выражений, это открывает возможность выражать большие модели с помощью сравнительно небольшого числа строк кода, который легко документировать, понять, модифицировать и отлаживать.

По мере укрупнения и усложнения моделей часто возникает желание развить эту концепцию, воспользовавшись принципом *композиции* из объектно ориентированного программирования.

При таком подходе мы группируем (компонуем) концептуально связанные переменные и ограничения в один объект. Важно не то, что ограничения созданы каким-то общим генератором выражений, а что переменные и ограничения описывают некоторую концепцию (зачастую физическую). Например, группа переменных и ограничений могла бы представлять поведение работающего электрогенератора (пределы нарастания, пределы спадания, кривые затрат) или химического оборудования, например дистилляционной колонны (уравнения массы, равновесного состояния и энергетического баланса). Из других примеров упомянем многопериодные задачи оптимизации, где одна и та же фундаментальная модель повторяется на протяжении многих временных периодов, или задачи стохастического программирования, где одна и та же базовая модель повторяется в разных сценариях с различными параметрами. В Pyomo мы используем компонент `Block`, поддерживающий композицию компонентов моделирования общего вида наподобие описанных выше.

Компонент `Block` – это контейнер для организации групп переменных и ограничений, он может содержать произвольное число именованных компонентов Pyomo – точно так же, как сами модели. На самом деле `ConcreteModel` и `AbstractModel` являются частными случаями компонента `Block`. Компоненты `Block` можно добавлять в модель или в другой блок, что позволяет конструировать иерархические модели из фундаментальных структурных элементов.

Эта идея иллюстрируется на рис. 8.1, где показано одно из возможных блочно-ориентированных представлений модели сети электроснабжения. В этом примере отдельные блоки определяют необходимые переменные и уравнения, описывающие один вид оборудования, будь то генератор, шина или линия передачи. Затем блоки компонуются в единую (однопериодическую) модель потока мощности в электрической сети. Эту модель можно решить саму по себе. Но ее можно использовать и как структурный элемент модели более высокого уровня, например задачи многопериодного планирования, показанной на этом рисунке. Таким способом Pyomo может представлять очень сложные модели, построенные из меньших не столь сложных частей, допускающих повторное использование.

Блоки дают авторам моделей и разработчикам алгоритмов несколько полезных преимуществ:

- *композиция*. Авторы моделей могут строить – и тестировать – небольшие модели или компоненты моделей, а затем собирать из них модели более крупных и сложных систем;
- *аннотация*. Модель может явно аннотировать свою структуру и давать «указания» как алгоритмам (например, алгоритмам декомпозиции

типа разложения Бендерса или прогрессивного построения), так и преобразованиям (например, обобщенное дизъюнктивное программирование) насчет того, как работать с моделью;

- *песочницы*. Поскольку каждый блок является своим собственным контейнером, он позволяет разработчикам моделей (компонентов) и алгоритмов строить и манипулировать компонентами модели в их собственных «частных» песочницах, не беспокоясь о конфликтах имен и иных непредусмотренных взаимодействиях с другими разработчиками.

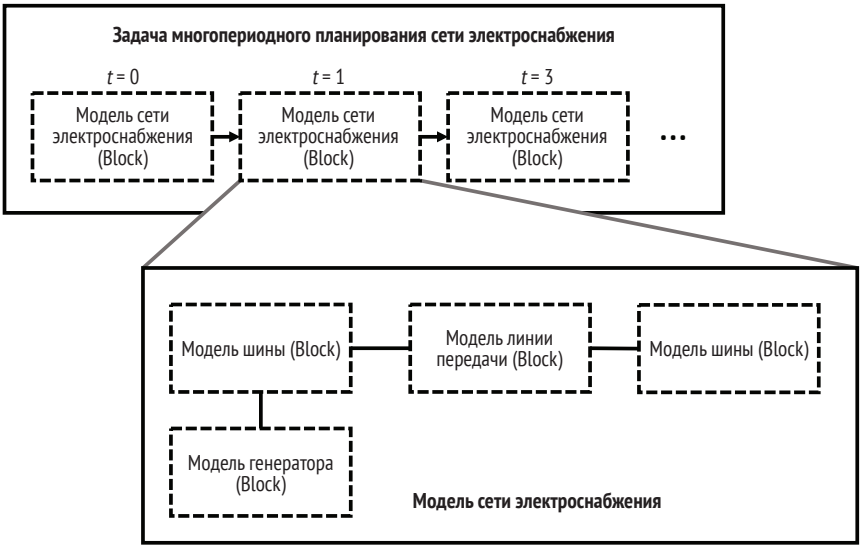

Рис. 8.1 ❖ Модель сети электроснабжения можно составить из отдельных блоков, представляющих генераторы, шины и линии передачи. Многопериодную модель можно построить иерархически, рассматривая модель сети электроснабжения как фундаментальный структурный элемент

8.2. Блочные структуры

Компонент Pyomo `Block` можно рассматривать практически так же, как модель: компоненты добавляются непосредственно в блок в виде атрибутов. Поскольку компоненты `Block` могут содержать другие компоненты моделирования, в т. ч. другие блоки, можно строить иерархические структуры с произвольной вложенностью.

В следующем фрагменте показана простая иерархия блоков.

```
model = pyo.ConcreteModel()
model.x = pyo.Var()
model.P = pyo.Param(initialize=5)
model.S = pyo.RangeSet(model.P)
```

```
model.b = pyo.Block()
model.b.I = pyo.RangeSet(model.P)
model.b.x = pyo.Var(model.b.I)
model.b.y = pyo.Var(model.S)
model.b.b = pyo.Block([1,2])
model.b.b[1].x = pyo.Var()
model.b.b[2].x = pyo.Var()
```

Здесь модель содержит переменную (`model.x`), параметр (`model.P`) и множество (`model.S`). Она также содержит блок (`Block`) `model.b`. Этот блок сам содержит множество, две переменные и еще один индексированный блок. Обратите внимание, что компоненты блока могут ссылаться на компоненты других блоков или родительской модели. Например, переменная `model.b.x` пользуется множеством из своего родительского блока, а переменная `model.b.y` ссылается на множество из родителя родительского блока (в данном случае – модели). Компоненты и выражения могут ссылаться на компоненты из любого места иерархии.

> **ПРИМЕЧАНИЕ** Внутри одного блока Pyomo поддерживает ссылки на компоненты из любого другого блока. Однако в объектно ориентированном проектировании считается хорошим тоном ссылаться только на компоненты, принадлежащие текущему или более низким уровням иерархии. Это открывает возможность повторно использовать блоки в других моделях, не делая сильных предположений о структуре блока-владельца или его родителей.

Обратите внимание, что во фрагменте кода выше каждый блок определяет свое собственное пространство имен; хотя имена компонентов должны быть уникальны в пределах одного блока, глобальная уникальность не требуется. Это позволяет конструировать блок, не опасаясь, что определения в одном блоке будут конфликтовать с определениями в других блоках. Таким образом, имена всех компонентов имеют две формы: локальное имя, которое может повторяться в других местах модели, и глобально уникальное полное имя, включающее имена родительских блоков, разделенные точками:

```
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[1].x.local_name) # x
print(model.b.b[1].x.name)       # b.b[1].x
```

При обсуждении иерархии блоков мы применяем терминологию, заимствованную из древовидных структур, и называем блок, находящийся на предыдущем уровне иерархии (ближе к модели верхнего уровня), *родительским*, а все находящиеся внутри блока компоненты – *дочерними*. *Корнем* иерархии блоков всегда является текущая модель. Все компоненты Pyomo предоставляют набор стандартных методов для перемещения по иерархии.

- `parent_component()`. У каждого объекта модели Pyomo есть единственный компонент-владелец. Вызов метода `parent_component()` объекта моделирования возвращает его владельца. Для элементов индексированного

компонента `parent_component()` возвращает сам этот индексированный компонент. Для скалярных компонентов родительским компонентом является сам скалярный компонент.

- `parent_block()`. Каждый компонент моделирования присоединен к единственному блоку. Вызов `parent_block()` для объекта моделирования возвращает блок, владеющий родительским компонентом объекта.
- `model()`. Вызов метода `model()` объекта моделирования поднимается вверх по цепочке вызовов `parent_block()` и возвращает блок самого верхнего уровня (корневой).
- `_getitem_`. Как и в случае моделей, к дочерним компонентам можно обращаться из программы путем просмотра атрибутов.
- `component()`. Дочерние компоненты можно также получать по имени, пользуясь методом блоков `component()`.

Приведенный ниже код иллюстрирует перемещение по иерархии:

```
model.b.b[1].x.parent_component() # model.b.b[1].x
model.b.b[1].x.parent_block()     # model.b.b[1]
model.b.b[1].x.model()            # model
model.b.b[1].component('x')       # model.b.b[1].x
model.b.x[1].parent_component()   # model.b.x
model.b.x[1].parent_block()       # model.b
model.b.x[1].model()              # model
model.b.component('x')            # model.b.x
```

Компонент `Block` можно также создать и заполнить, а потом добавить в модель. Это показано в коде ниже. Также этот код иллюстрирует, как родительская модель может воспользоваться множествами и параметрами, содержащимися в дочернем блоке.

```
new_b = pyo.Block()
new_b.x = pyo.Var()
new_b.P = pyo.Param(initialize=5)
new_b.I = pyo.RangeSet(10)

model = pyo.ConcreteModel()
model.b = new_b
model.x = pyo.Var(model.b.I)
```

> **ПРИМЕЧАНИЕ** В этом примере объект `new_b` типа `Block` не инициализируется в момент объявления. То есть он остается абстрактным до момента добавления в объект `ConcreteModel`. А в этот момент сразу же инициализируется. Аналогично, когда объект `Block` добавляется в `AbstractModel`, он не инициализируется, пока не будет инициализирован объект объемлющей модели.

8.3. Блоки как индексированные компоненты

Как и другие компоненты Pyomo, блоки можно индексировать и инициализировать с помощью правила конструирования. Однако в случае блоков для

этих правил действует немного иное соглашение: первым аргументом правила конструирования блока является подлежащий заполнению блок, а не блок-владелец. Этот блок уже присоединен к модели, поэтому методы `model()`, `parent_block()` и им подобные работают, как ожидается. Внутри правила можно либо непосредственно заполнить блок, включив в него компоненты, либо создать новый блок и вернуть его.

В следующем примере иллюстрируется использование правил конструирования для блоков:

```
model = pyo.ConcreteModel()
model.P = pyo.Param(initialize=3)
model.T = pyo.RangeSet(model.P)

def xyb_rule(b, t):
    b.x = pyo.Var()
    b.I = pyo.RangeSet(t)
    b.y = pyo.Var(b.I)
    b.c = pyo.Constraint(expr=b.x == 1 - sum(b.y[i] for i in b.I))
model.xyb = pyo.Block(model.T, rule=xyb_rule)
```

Здесь определен индексированный компонент `Block`, содержащий по одному блоку для каждого элемента множества `model.T`. В правиле конструирования мы создаем две переменные и множество. Эти правила конструирования такие же гибкие, как для других компонентов. Множества `b.I`, создаваемые в этом примере, для каждого блока различны, и, следовательно, переменная `b.y` также содержит различную длину для каждого блока. Тем самым иллюстрируется еще одна особенность блоков. Часто различные блоки имеют в точности одинаковую структуру, но содержат разные данные. Можно также конструировать блоки с разной структурой, зависящей от данных, доступных в правиле конструирования.

Этот пример можно дополнить, добавив печать тела ограничения `c` для каждого блока:

```
for t in model.T:
    print(model.xyb[t].c.body)
```

Ограничения расширяются и содержат полные имена переменных в каждом подблоке:

```
-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. Правила конструирования внутри блоков

Как и объекты моделей, блоки могут содержать другие компоненты моделирования, в т. ч. объекты `Set` и `Param`. Кроме того, блоки можно инициализировать компонентами моделирования, которые сами сконструированы

с помощью правил. Однако при этом проявляется одна тонкость, связанная с правилами конструирования компонентов в Pyomo.

До сих пор мы часто называли первый аргумент правила конструирования компонент «model», что не вполне корректно. На самом деле первым аргументом является *блок, владеющий* конструируемым компонентом. Для «плоских» моделей (не содержащих подблоков) блок-владелец действительно является моделью, но для иерархических моделей это уже не так. При необходимости объект модели можно получить от блока-владельца, вызвав вышеупомянутый метод `model()`.

Например, рассмотрим альтернативное объявление блока `xyb` из предыдущего примера:

```
def xyb_rule(b, t):
    b.x = pyo.Var()
    b.I = pyo.RangeSet(t)
    b.y = pyo.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 = pyo.Constraint(rule=_b_c_rule)
model.xyb = pyo.Block(model.T, rule=xyb_rule)
```

Здесь блок `xyb` включает ограничение, определенное правилом `_b_c`. Блок-владелец `_b`, переданный этому правилу, совпадает с `b`. Однако переменная `b` определена локально. Это позволяет использовать правило, даже если блок-владелец сконструирован по-другому.

Поскольку блок-владелец (или модель) НЕ передается правилу конструирования блока, автору модели необходим иной механизм для доступа к компонентам родительского или других блоков в иерархии. Методы компонентов `parent_block` и `model` позволяют перемещаться вверх по иерархии блоков. Метод `parent block()` любого компонента или информационного объекта компонента возвращает блок, к которому присоединен компонент. Метод `model()` любого компонента или информационного объекта компонента возвращает блок, находящийся в корне дерева.

8.5. Извлечение значений из иерархических моделей

Хотя блоки дают удобный механизм выражения составных концепций (например, периода времени, сценария), у них есть неприятное последствие: данные оказываются разбросаны по всей модели. Впрочем, мы можем получать доступ к компонентам, явно обходя блоки и связанные с ними переменные:

```
for t in model.xyb:
    for i in model.xyb[t].y:
        print("%s %f" % (model.xyb[t].y[i], pyo.value(model.xyb[t].y[i])))
```

Кроме того, нотацию срезов можно использовать для динамического выделения подмножества блоков или значений переменных:

```
for y in model.xyb[:].y[:]:
    print("%s %f" % (y, pyo.value(y)))
```

8.6. Пример использования блоков: оптимальный многопериодный размер партии

Далее мы продемонстрируем полную основанную на блоках модель на примере хорошо известной задачи многопериодной оптимизации размера партии [31]. Цель – определить, какой оптимальный размер партии изделий X_t следует произвести в каждом временном интервале $t \in T$, если известен спрос d_t. Обозначим y_t двоичную переменную, показывающую, следует или не следует производить что-нибудь в интервале t, и предположим, что если решено производить, то затраты c_t фиксированы. Наличное количество I_t в конце каждого временного интервала является функцией предыдущего наличия, произведенного и проданного количества:

$$I_t = I_{t-1} + X_t - d_t.$$

Если разрешить отрицательное наличие (т. е. мы не смогли удовлетворить спрос и ведем очередь невыполненных заказов), то наличие можно представить в виде $I_t = I_t^+ - I_t^-$, где I_t^+ и I_t^- неотрицательны. Здесь I_t^+ – наличие на складе, а I_t^- – очередь заказов. Мы можем определить затраты на складское хранение h_t^+ и затраты, связанные с нехваткой изделий (затраты на ведение очереди заказов) h_t^-.

При таком описании задачу оптимизации можно сформулировать следующим образом:

$$\min \sum_{t \in T} c_t y_t + h_t^+ I_t^+ + h_t^- I_t^- \qquad \text{(LS.1)}$$

$$\text{при условиях } I_t = I_{t-1} + X_t - d_t \qquad \forall t \in T \qquad \text{(LS.2)}$$

$$I_t = I_t^+ - I_t^- \qquad \forall t \in T \qquad \text{(LS.3)}$$

$$X_t \leqslant P y_t \qquad \forall t \in T \qquad \text{(LS.4)}$$

$$X_t, I_t^+, I_t^- \geqslant 0 \qquad \forall t \in T \qquad \text{(LS.5)}$$

$$y_t \in \{0, 1\} \qquad \forall t \in T \qquad \text{(LS.6)}$$

где ограничение (LS.4) разрешает производить продукцию во временном интервале t, только если индикаторная переменная $y_t = 1$. Данные для нашей задачи представлены в табл. 8.1.

Таблица 8.1. Данные для задачи о размере партии

Параметр	Описание	Значение
c	фиксированные производственные затраты	4.6
I_0^+	начальное значение положительного наличия	5.0
I_0^-	начальное значение заказов в очереди	0.0
h^+	затраты на складское хранение (на единицу товара)	0.7
h^-	затраты, связанные с нехваткой (на единицу товара)	1.2
P	максимальный объем производства (M большое)	5
d	спрос	[5, 7, 6.2, 3.1, 1.7]

8.6.1. Формулировка без блоков

Формулировка задачи о размере партии без блоков показана ниже.

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.T = pyo.RangeSet(5) # time periods

i0 = 5.0 # начальное наличие
c = 4.6 # начальные затраты
h_pos = 0.7 # затраты на складское хранение
h_neg = 1.2 # затраты, связанные с нехваткой
P = 5.0 # максимальный объем производства

# спрос в интервале t
d = {1: 5.0, 2:7.0, 3:6.2, 4:3.1, 5:1.7}

# определить переменные
model.y = pyo.Var(model.T, domain=pyo.Binary)
model.x = pyo.Var(model.T, domain=pyo.NonNegativeReals)
model.i = pyo.Var(model.T)
model.i_pos = pyo.Var(model.T, domain=pyo.NonNegativeReals)
model.i_neg = pyo.Var(model.T, domain=pyo.NonNegativeReals)

# определить связи
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 = pyo.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 = pyo.Constraint(model.T, rule=pos_neg_rule)

# создать ограничение M большое для индикаторной переменной производства
def prod_indicator_rule(m,t):
    return m.x[t] <= P*m.y[t]
```

```
model.prod_indicator = pyo.Constraint(model.T, rule=prod_indicator_rule)

# определить целевую функцию
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 = pyo.Objective(rule=obj_rule)

# решить задачу
solver = pyo.SolverFactory('glpk')
solver.solve(model)

# напечатать результаты
for t in model.T:
    print('Period: {0}, Prod. Amount: {1}'.format(t, pyo.value(model.x[t])))
```

В этом примере используется стандартный синтаксис Pyomo, рассмотренный в предыдущих главах. Если бы нас интересовала задача о размере партии на протяжении только одного временного интервала, то объявления переменных выглядели бы следующим образом:

```
# определить переменные
model.y = pyo.Var(domain=pyo.Binary)
model.x = pyo.Var(domain=pyo.NonNegativeReals)
model.i = pyo.Var()
model.i_pos = pyo.Var(domain=pyo.NonNegativeReals)
model.i_neg = pyo.Var(domain=pyo.NonNegativeReals)
```

В многопериодном случае мы имеем те же фундаментальные переменные и ограничения, определенные в каждом временном интервале. Определения переменных выглядят так:

```
# определить переменные
model.y = pyo.Var(model.T, domain=pyo.Binary)
model.x = pyo.Var(model.T, domain=pyo.NonNegativeReals)
model.i = pyo.Var(model.T)
model.i_pos = pyo.Var(model.T, domain=pyo.NonNegativeReals)
model.i_neg = pyo.Var(model.T, domain=pyo.NonNegativeReals)
```

Если бы мы рассматривали многопериодную и многосценарную задачу (например, стохастическую формулировку задачи о размере партии в условиях неопределенности), то объявления переменных выглядели бы так:

```
# определить переменные
model.y = pyo.Var(model.T, model.S, domain=pyo.Binary)
model.x = pyo.Var(model.T, model.S, domain=pyo.NonNegativeReals)
model.i = pyo.Var(model.T, model.S,)
model.i_pos = pyo.Var(model.T, model.S, domain=pyo.NonNegativeReals)
model.i_neg = pyo.Var(model.T, model.S, domain=pyo.NonNegativeReals)
```

В каждом из этих примеров при увеличении сложности или добавлении нового уровня в модель мы добавляем новый индекс к переменным и ограничениям. Этот подход широко применяется в исследовании операций. К сожалению, при этом приходится полностью переопределять модель при

добавлении каждого нового уровня, а конструирование иерархических моделей с повторно используемым кодом толком не поддерживается. Блоки предлагают другой подход, который позволяет легко поддержать повторное использование модели объектно ориентированным способом.

8.6.2. Формулировка с блоками

Теперь покажем, как для решения этой задачи можно использовать блоки. Большинство ограничений в многопериодной задаче о размере партии определены для $t \in T$, и их можно логически сгруппировать по времени. Pyomo позволяет определять блоки, каждый со своими переменными и ограничениями, только для одного периода, а затем связать их, образовав полную модель.

В нашей задаче переменные и ограничения внутри правила можно определить в правиле конструирования блока для одного интервала:

```
# создать блок для одного временного интервала
def lotsizing_block_rule(b, t):
    # определить переменные
    b.y = pyo.Var(domain=pyo.Binary)
    b.x = pyo.Var(domain=pyo.NonNegativeReals)
    b.i = pyo.Var()
    b.i0 = pyo.Var()
    b.i_pos = pyo.Var(domain=pyo.NonNegativeReals)
    b.i_neg = pyo.Var(domain=pyo.NonNegativeReals)

    # определить ограничения
    b.inventory = pyo.Constraint(expr=b.i == b.i0 + b.x - d[t])
    b.pos_neg = pyo.Constraint(expr=b.i == b.i_pos - b.i_neg)
    b.prod_indicator = pyo.Constraint(expr=b.x <= P * b.y)
model.lsb = pyo.Block(model.T, rule=lotsizing_block_rule)
```

Здесь внутри правила определены переменные и ограничения для одного временного интервала t. Затем компонент `Block` индексируется множеством `model.T` и конструируется блок задачи о размере партии для каждого элемента `model.T`. Теперь объект модели содержит блок для каждого временного интервала t. Последнее, что нужно сделать, – задать ограничения, связывающие блоки вместе (начальное наличие следующего блока равно конечному наличию предыдущего), и определить целевую функцию, учитывающую все блоки. Полный листинг версии модели многопериодной задачи о размере партии с блоками показан ниже.

```
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.T = pyo.RangeSet(5) # time periods

i0 = 5.0 # начальное наличие
c = 4.6 # начальные затраты
h_pos = 0.7 # затраты на складское хранение
h_neg = 1.2 # затраты, связанные с нехваткой
P = 5.0 # максимальный объем производства
```

```
# спрос в интервале t
d = {1: 5.0, 2:7.0, 3:6.2, 4:3.1, 5:1.7}

# создать блок для одного периода
def lotsizing_block_rule(b, t):
    # определить переменные
    b.y = pyo.Var(domain=pyo.Binary)
    b.x = pyo.Var(domain=pyo.NonNegativeReals)
    b.i = pyo.Var()
    b.i0 = pyo.Var()
    b.i_pos = pyo.Var(domain=pyo.NonNegativeReals)
    b.i_neg = pyo.Var(domain=pyo.NonNegativeReals)

    # определить ограничения
    b.inventory = pyo.Constraint(expr=b.i == b.i0 + b.x - d[t])
    b.pos_neg = pyo.Constraint(expr=b.i == b.i_pos - b.i_neg)
    b.prod_indicator = pyo.Constraint(expr=b.x <= P * b.y)
model.lsb = pyo.Block(model.T, rule=lotsizing_block_rule)

# связать переменные наличия соседних блоков
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 = pyo.Constraint(model.T, rule=i_linking_rule)

# сконструировать целевую функцию по всем блокам
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 = pyo.Objective(rule=obj_rule)

### решить задачу
solver = pyo.SolverFactory('glpk')
solver.solve(model)

# напечатать результаты
for t in model.T:
    print('Period: {0}, Prod. Amount: {1}'.format(t, \
        pyo.value(model.lsb[t].x)))
```

Это небольшая задача, поэтому трудно разглядеть все преимущества блоков. Однако с ростом размера и сложности модели такой объектно ориентированный подход к моделированию позволяет определять небольшие части модели в виде автономного кода, а затем строить всю модель, соединяя части. Пример был выбран отчасти потому, что это хорошо изученная классическая многоступенчатая модель управления запасами. Легко можно представить себе обобщения модели, включающие дополнительные ограничения и затраты. На самом деле в научной литературе описано много таких моделей, и некоторые из них применяются на практике. В больших моделях принято писать методы или классы для определения индивидуальных блоков и повторно использовать код в нескольких различных высокоуровневых формулировках задачи оптимизации.

Глава 9

Производительность: конструирование модели и интерфейсы с решателями

В этой главе описываются средства профилирования на этапе конструирования модели и повышения производительности как процесса конструирования, так и взаимодействия с решателями. Мы начнем с обсуждения различных инструментов профилирования, которые можно использовать для отыскания узких мест. В Pyomo имеются встроенные средства профилирования, но есть также Python-пакеты, например `cProfile` и `line_profiler`, специально предназначенные для изучения производительности. В разделе 9.2 обсуждается класс `LinearExpression`, который для некоторых приложений позволяет резко улучшить время конструирования модели. В разделе 9.3 описывается, как можно использовать интерфейсы с хранимыми решателями, чтобы с высокой эффективностью повторно решать модели после внесения небольших изменений. Наконец, в разделе 9.4 обсуждаются разреженные множества индексов.

9.1. Выявление узких мест с помощью профилирования

Если производительность вызывает вопросы, то необходимо сначала выявить узкие места, а только потом вносить улучшения. В этом разделе описываются различные инструменты профилирования, которые можно использовать для выявления узких мест.

В качестве примера мы снова будем использовать задачу о размещении складов, описанную в разделе 3.2. Но здесь нас будет интересовать ослабление задачи применительно к непрерывному случаю. Мы перепишем задачу, встроив модель в функцию и сделав максимальное число складов изменяемым параметром.

```
import pyomo.environ as pyo # импортировать окружение pyomo
import cProfile
import pstats
import io
from pyomo.common.timing import TicTocTimer, report_timing
from pyomo.opt.results import assert_optimal_termination
from pyomo.core.expr.numeric_expr import LinearExpression
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(0)

def create_warehouse_model(num_locations=50, num_customers=50):
    N = list(range(num_locations)) # местоположения складов
    M = list(range(num_customers)) # потребители

    d = dict() # расстояния от складов до потребителей
    for n in N:
        for m in M:
            d[n, m] = np.random.randint(low=1, high=100)
    max_num_warehouses = 2

    model = pyo.ConcreteModel(name="(WL)")
    model.P = pyo.Param(initialize=max_num_warehouses,
                        mutable=True)

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

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

    def demand_rule(mdl, m):
        return sum(mdl.x[n,m] for n in N) == 1
    model.demand = pyo.Constraint(M, rule=demand_rule)

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

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

    return model
```

9.1.1. Хронометраж

В Pyomo имеется очень полезная функция, `report_timing`, для профилирования процесса конструирования модели. Во фрагменте ниже демонстрируется ее использование в задаче о расположении складов.

```
report_timing()
print('Building model')
print('--------------')
m = create_warehouse_model(num_locations=200, num_customers=200)
```

Ниже показано, что она выводит.

```
Building model
--------------
       0 seconds to construct Block ConcreteModel; 1 index total
       0 seconds to construct Set Any; 1 index total
       0 seconds to construct Param P; 1 index total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
       0 seconds to construct Set SetProduct_OrderedSet; 1 index total
       0 seconds to construct Set SetProduct_OrderedSet; 1 index total
    0.09 seconds to construct Var x; 40000 indicies total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
       0 seconds to construct Var y; 200 indicies total
    0.19 seconds to construct Objective obj; 1 index total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
    0.11 seconds to construct Constraint demand; 200 indicies total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
       0 seconds to construct Set OrderedSimpleSet; 1 index total
       0 seconds to construct Set SetProduct_OrderedSet; 1 index total
       0 seconds to construct Set SetProduct_OrderedSet; 1 index total
    0.62 seconds to construct Constraint warehouse_active; 40000 indicies total
       0 seconds to construct Constraint num_warehouses; 1 index total
```

В результате вызова `report_timing` Pyomo печатает время построения каждого компонента. В предпоследней строке мы видим, что почти все время конструирования модели потрачено на построение ограничения `warehouse_active`. Отметим, что время обработки данных в правилах конструирования включается, поэтому не все время тратится в Pyomo. Но в данном примере никакой обработки данных в правиле нет, так что мы можем точно сказать, что узким местом является построение выражений для ограничения `warehouse_active`.

9.1.2. TicTocTimer

Pyomo также предоставляет класс `TicTocTimer` для удобства хронометража. В примере ниже мы сравниваем время, необходимое для построения моде-

ли, со временем записи в файл и применяем для решения задачи решатель Gurobi. Предположим, что для решения модели имеется следующая функция:

```
def solve_warehouse_location(m):
    opt = pyo.SolverFactory('gurobi')
    res = opt.solve(m)
    assert_optimal_termination(res)
```

Для хронометража с помощью `TicTocTimer` можно написать такой скрипт:

```
timer = TicTocTimer()
timer.tic('start')
m = create_warehouse_model(num_locations=200, num_customers=200)
timer.toc('Built model')
solve_warehouse_location(m)
timer.toc('Wrote LP file and solved')
```

Вывод показан ниже.

```
[     0.00] start
[+    1.22] Built model
[+    4.05] Wrote LP file and solved
```

Мы видим, что для построения модели Pyomo понадобилось 1.22 с, а для решения задачи и записи LP-файла – 4.05 с. Использование `TicTocTimer` показало, что запись LP-файла и решение задачи занимают гораздо больше времени, чем конструирование модели. Однако пока еще не ясно, какая часть этих 4.05 с потрачена на запись LP-файла, и вот тут может пригодиться `cProfile`.

9.1.3. Профилировщики

Есть много Python-пакетов для профилирования программ. Два из них – `cProfile` (https://docs.python.org/3/library/profile.html) и `line_profiler` (https://github.com/pyutils/line_profiler). Пакет `cProfile` сообщает время, проведенное в каждой функции, а пакет `line_profiler` – время, проведенное в каждой строке функции.

Вернемся к задаче о расположении складов. Сначала напишем функцию, которая пробует параметры, раз за разом решая задачу при различных значениях максимального числа складов, `m.P`.

```
def solve_parametric():
    m = create_warehouse_model(num_locations=50, num_customers=50)
    opt = pyo.SolverFactory('gurobi')
    p_values = list(range(1, 31))
    obj_values = list()
    for p in p_values:
        m.P.value = p
        res = opt.solve(m)
        assert_optimal_termination(res)
        obj_values.append(res.problem.lower_bound)
```

Сначала вызовем функцию `solve_parametric` и напечатаем время, потраченное на перебор значений параметра.

```
solve_parametric()
timer.toc('Finished parameter sweep')
```

Напечатана строка

```
[+ 7.28] finished parameter sweep
```

`TicTocTimer` сообщает, что на перебор значений параметра потрачено 7.28 с. Далее напишем вспомогательную функцию для печати результатов `cProfile`.

```
def print_c_profiler(pr, lines_to_print=15):
    s = io.StringIO()
    stats = pstats.Stats(pr, stream=s).sort_stats('cumulative')
    stats.print_stats(lines_to_print)
    print(s.getvalue())
    s = io.StringIO()
    stats = pstats.Stats(pr, stream=s).sort_stats('tottime')
    stats.print_stats(lines_to_print)
    print(s.getvalue())
```

Мы пользуемся пакетом `pstats` для сортировки результатов `cProfile` и печати заданного числа строк. Мы печатаем статистику, отсортированную по совокупному времени и по полному времени. В документации по cProfile сказано, что совокупное время – это время, проведенное в соответствующей функции, включая все вызываемые из нее функции, а полное время – время, проведенное в соответствующей функции, исключая вызываемые из нее функции.

Пакет `cProfile` используется следующим образом:

```
pr = cProfile.Profile()
pr.enable()
solve_parametric()
pr.disable()
print_c_profiler(pr)
```

Ниже показан вывод, содержащий результат хронометража приведенного выше кода.

```
     14897207 function calls (14894602 primitive calls) in 10.229 seconds

Ordered by: cumulative time
List reduced from 559 to 15 due to restriction <15>

ncalls tottime percall cumtime percall filename:lineno(function)
   1 0.001 0.001 10.229 10.229 wl.py:112(solve_parametric)
  30 0.006 0.000 10.107 0.337 /.../pyomo/pyomo/opt/base/solvers.py:513(solve)
  30 0.000 0.000 5.758 0.192 /.../pyomo/pyomo/solvers/plugins/solvers/GUROBI.py:189(_presolve)
  30 0.001 0.000 5.758 0.192 /.../pyomo/pyomo/opt/solver/shellcmd.py:189(_presolve)
  30 0.001 0.000 5.726 0.191 /.../pyomo/pyomo/opt/base/solvers.py:653(_presolve)
  30 0.000 0.000 5.725 0.191 /.../pyomo/pyomo/opt/base/solvers.py:721(_convert_problem)
  30 0.002 0.000 5.724 0.191 /.../pyomo/pyomo/opt/base/convert.py:31(convert_problem)
```

```
  30 0.001 0.000 5.692 0.190 /.../pyomo/pyomo/solvers/plugins/converter/model.py:43(apply)
  30 0.001 0.000 5.675 0.189 /.../pyomo/pyomo/core/base/block.py:1736(write)
  30 0.016 0.001 5.673 0.189 /.../pyomo/pyomo/repn/plugins/cpxlp.py:84(__call__)
  30 0.804 0.027 5.650 0.188 /.../pyomo/pyomo/repn/plugins/cpxlp.py:380(_print_model_LP)
  30 0.003 0.000 3.528 0.118 /.../pyomo/pyomo/opt/solver/shellcmd.py:224(_apply_solver)
  30 0.002 0.000 3.524 0.117 /.../pyomo/pyomo/opt/solver/shellcmd.py:290(_execute_command)
  30 0.008 0.000 3.522 0.117 /.../pyutilib/pyutilib/subprocess/processmngr.py:433(run_command)
  30 0.001 0.000 3.292 0.110 /.../pyutilib/pyutilib/subprocess/processmngr.py:829(wait)

     14897207 function calls (14894602 primitive calls) in 10.229 seconds

Ordered by: internal time
List reduced from 559 to 15 due to restriction <15>

ncalls tottime percall cumtime percall filename:lineno(function)
    30 3.285 0.110 3.285 0.110 {built-in method posix.waitpid}
    30 0.804 0.027 5.650 0.188 /.../pyomo/pyomo/repn/plugins/cpxlp.py:380(_print_model_LP)
 76560 0.423 0.000 0.534 0.000 /.../pyomo/pyomo/repn/standard_repn.py:433(_collect_sum)
 76560 0.419 0.000 0.700 0.000 /.../pyomo/pyomo/repn/plugins/cpxlp.py:181(_print_expr_canonical)
 76560 0.339 0.000 1.049 0.000 /.../pyomo/pyomo/repn/standard_repn.py:982(_generate_standard_repn)
306000 0.338 0.000 0.586 0.000 /.../pyomo/pyomo/core/base/set.py:581(bounds)
    30 0.252 0.008 0.375 0.013 /.../pyomo/pyomo/solvers/plugins/solvers/GUROBI.py:363(process_soln_file)
 76560 0.197 0.000 1.411 0.000 /.../pyomo/pyomo/repn/standard_repn.py:254(generate_standard_repn)
 76560 0.159 0.000 1.812 0.000 /.../pyomo/pyomo/repn/plugins/cpxlp.py:572(constraint_generator)
225090 0.157 0.000 0.206 0.000 /.../pyomo/pyomo/core/base/constraint.py:206(has_ub)
153060 0.148 0.000 0.256 0.000 /.../pyomo/pyomo/core/expr/symbol_map.py:82(createSymbol)
 77220 0.124 0.000 0.272 0.000 {built-in method builtins.sorted}
153000 0.123 0.000 0.454 0.000 /.../pyomo/pyomo/core/base/var.py:407(ub)
153000 0.122 0.000 0.457 0.000 /.../pyomo/pyomo/core/base/var.py:394(lb)
229530 0.116 0.000 0.222 0.000 /.../pyomo/pyomo/repn/plugins/cpxlp.py:41(_get_bound)
```

В первой строке показано общее число вызовов функций и общее время выполнения профилируемого кода. Заметим, что 10.229 с, напечатанные `cProfile`, – это намного больше 7.28 с, о которых сообщает `TicTocTimer`. Использование `cProfile` добавляет накладные расходы, поэтому `cProfile` следует применять только для поиска узких мест (а не, к примеру, для сравнения двух алгоритмов). Первый блок результатов отсортирован по совокупному времени. В каждой строке показано совокупное время, потраченное в функции, имя которой приведено в конце строки. Как и следовало ожидать, все 10.229 с потрачены в функции `solve_parametric`. Из этого общего времени 10.107 с потрачено внутри вызова функции Pyomo `solve`. Уже сейчас можно сделать вывод, что на конструирование модели, изменение значений `m.P` и проверку условия завершения решателя ушло очень мало времени. Кроме того, мы видим, что на обращение к функции `write` (которая записывает LP-файл) потрачено 5.675 с из 10.107 с, проведенных в `solve`. И только 3.528 с потрачено в функции `apply_solver`, где выполняется подпроцесс, вызывающий Gurobi. Эти результаты наводят на мысль, что в этом приложении был бы полезен интерфейс с хранимым решателем. Такие интерфейсы обсуждаются в разделе 9.3.

9.2. Повышение производительности конструирования модели с помощью класса LinearExpression

В этом разделе мы обсудим прямое использование класса `LinearExpression` для ускорения конструирования модели. Хотя перегрузка операторов – очень удобный способ конструирования выражений, связанные с ней накладные расходы могут быть велики. Поэтому Pyomo поддерживает создание линейных выражений с помощью встроенных в Python списков, что может оказаться значительно быстрее перегрузки операторов. Конструктор класса `LinearExpression` принимает три именованных аргумента:

- `constant`: постоянный член;
- `linear_vars`: список переменных Pyomo, встречающихся в линейных выражениях;
- `linear_coefs`: список коэффициентов для каждой из переменных в списке `linear_vars`.

В следующем примере сравнивается время создания линейного выражения с помощью перегрузки оператора и конструктора `LinearExpression`. Для получения надежного хронометража мы создаем выражение 100 000 раз.

```
import pyomo.environ as pyo
from pyomo.common.timing import TicTocTimer
from pyomo.core.expr.numeric_expr import LinearExpression

N1 = 10
N2 = 100000

m = pyo.ConcreteModel()
m.x = pyo.Var(list(range(N1)))

timer = TicTocTimer()
timer.tic()

for i in range(N2):
    e = sum(i*m.x[i] for i in range(N1))
timer.toc('created expression with sum function')

for i in range(N2):
    coefs = [i for i in range(N1)]
    lin_vars = [m.x[i] for i in range(N1)]
    e = LinearExpression(constant=0, linear_coefs=coefs, linear_vars=lin_vars)
timer.toc('created expression with LinearExpression constructor')
```

Результат показан ниже.

```
[  0.00] Resetting the tic/toc delta timer
[+ 3.52] created expression with sum function
[+ 0.52] created expression with LinearExpression constructor
```

Хотя использование `LinearExpression` не так понятно и занимает больше строк кода, в данном примере оно оказалось в 6 раз быстрее. Отметим, что выигрыш сильно зависит от количества членов в выражении.

9.3. Многократное решение с применением хранимых решателей

Всякий раз, как модель Pyomo решается с применением стандартных интерфейсов с решателем, обобщенные компоненты моделирования, определяющие модель, должны быть преобразованы в какую-то форму, понятную решателю. В тех случаях, когда решатель завершается сравнительно быстро, накладные расходы на такое преобразование модели могут оказаться значительными. Интерфейсы с хранимым решателем предлагают механизм уменьшения общего времени преобразования для моделей, которые решаются многократно с инкрементными изменениями. Это возможно благодаря функциональности, позволяющей пользователю эффективно уведомлять решатель о таких изменениях в модели Pyomo, после начального этапа, на котором модель преобразуется целиком.

При использовании хранимых решателей нужно внимательно следить за тем, чтобы модель Pyomo и ее преобразованное для решателя представление были синхронизированы. Это ручная процедура, выполняемая пользователем. Но даже при таком осложнении повышение производительности возможно для многих типичных подходов к оптимизации.

В разделе 9.3.1 мы кратко обсудим, когда хранимые решатели наиболее полезны. В разделах 9.3.2–9.3.4 описано, как использовать интерфейс с хранимым решателем. В разделе 9.3.5 пример с перебором значения параметра из раздела 9.1.3 будет переписан с применением интерфейса с хранимым решателем.

9.3.1. Когда использовать хранимый решатель

Интерфейс с хранимым решателем предназначен для тех случаев, когда одна и та же модель многократно решается с небольшими изменениями. Он наиболее полезен, когда время решения не намного больше времени преобразования модели Pyomo во входной формат решателя. Конечно, хранимые решатели можно использовать независимо от того, как долго решается задача. Но если время решения задачи значительно больше времени преобразования,

то выигрыш будет невелик, а сложность кода возрастет. Линейные программы – отличные кандидаты для использования в сочетании с хранимыми решателями, потому что большинство линейных задач решаются очень эффективно. С другой стороны, многие смешанно-целочисленные программы решаются долго, поэтому интерфейс с хранимым решателем для них бесполезен. Окончательным арбитром при решении вопроса о том, стоит ли применять хранимый решатель в конкретном приложении, является профилировщик (см. раздел 9.1).

9.3.2. Основы использования

Первый шаг при использовании хранимого решателя – создать модель Pyomo, как обычно.

```
import pyomo.environ as pyo
m = pyo.ConcreteModel()
m.x = pyo.Var()
m.y = pyo.Var()
m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
```

Для создания экземпляра хранимого решателя можно воспользоваться функцией `SolverFactory`.

```
opt = pyo.SolverFactory('gurobi_persistent')
```

Этот вызов возвращает экземпляр класса `GurobiPersistent`. Теперь нужно сообщить решателю о модели.

```
opt.set_instance(m)
```

В результате создается объект модели `gurobipy` и включаются необходимые переменные и ограничения. Теперь можно решить модель.

```
results = opt.solve()
```

Отметим, что методу `solve` не нужно передавать модель, как в большинстве интерфейсов с решателями. Можно также добавлять или удалять переменные, ограничения или блоки и задавать целевые функции. Например, код

```
m.c2 = pyo.Constraint(expr=m.y >= m.x)
opt.add_constraint(m.c2)
```

говорит решателю, что нужно добавить одно новое ограничение, но в остальном оставить модель неизменной. Затем можно снова решить модель.

```
results = opt.solve()
```

Для удаления компонента нужно просто вызвать соответствующий метод `remove`.

```
opt.remove_constraint(m.c2)
del m.c2
results = opt.solve()
```

Если компонент Pyomo заменяется другим компонентом с таким же именем, то первый компонент необходимо удалить из решателя. В противном случае у решателя окажется несколько компонентов. Например, следующий код выполняется без ошибок, но у решателя имеется дополнительное ограничение. Решатель должен удовлетворить оба ограничения, $y \geqslant -2^*x + 5$ и $y \leqslant x$, хотя мы вовсе не имели этого в виду!

```
m = pyo.ConcreteModel()
m.x = pyo.Var()
m.y = pyo.Var()
m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
opt = pyo.SolverFactory('gurobi_persistent')
opt.set_instance(m)
# ОШИБКА:
del m.c
m.c = pyo.Constraint(expr=m.y <= m.x)
opt.add_constraint(m.c)
```

Правильно было сделать так:

```
m = pyo.ConcreteModel()
m.x = pyo.Var()
m.y = pyo.Var()
m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
opt = pyo.SolverFactory('gurobi_persistent')
opt.set_instance(m)
# Правильно:
opt.remove_constraint(m.c)
del m.c
m.c = pyo.Constraint(expr=m.y <= m.x)
opt.add_constraint(m.c)
```

В большинстве случаев есть только один способ модифицировать компонент – удалить его из экземпляра решателя, модифицировать в Pyomo, а затем вернуть в решатель. Единственное исключение – переменные. Переменные можно модифицировать, после чего обновить в решателе:

```
m = pyo.ConcreteModel()
m.x = pyo.Var()
m.y = pyo.Var()
m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
opt = pyo.SolverFactory('gurobi_persistent')
opt.set_instance(m)
m.x.setlb(1.0)
opt.update_var(m.x)
```

Короче говоря, всякий раз, как модель Pyomo изменяется, интерфейс хранимого решателя необходимо уведомить и синхронизировать. В табл. 9.1 перечислены методы, вызываемые при различных модификациях модели Pyomo. Отметим, что при модификации изменяемых параметров или именованных выражений `Expression` необходимо обновить (удалить и снова добавить) все ограничения, в которые входят измененные параметры или выражения.

Таблица 9.1. Методы интерфейса с хранимым решателем

Добавить ограничение	`opt.add constraint()`
Добавить переменную	`opt.add var()`
Добавить блок	`opt.add block()`
Задать целевую функцию	`opt.set objective()`
Удалить ограничение	`opt.remove constraint()`
Удалить переменную	`opt.remove var()`
Удалить блок	`opt.remove block()`
Изменить переменную	`opt.update var()`

Изменить изменяемый параметр	`opt.remove constraint()`
	`m.param.value = val`
	`opt.add constraint()`

Изменить выражение	`opt.remove constraint()`
	`m.expr += val`
	`opt.add constraint()`

9.3.3. Работа с индексированными переменными и ограничениями

Во всех примерах из раздела 9.3.2 использовались скалярные переменные и ограничения; для использования индексированных переменных или ограничений код необходимо немного изменить:

```
m.v = pyo.Var([0, 1, 2])
m.c2 = pyo.Constraint([0, 1, 2])
for i in range(3):
    m.c2[i] = m.v[i] == i
for v in m.v.values():
    opt.add_var(v)
for c in m.c2.values():
    opt.add_constraint(c)
```

Это нужно делать также при удалении индексированных переменных и ограничений. Отметим, что для автоматизации процесса можно использовать метод `is_indexed`.

9.3.4. Повышение производительности

Чтобы добиться максимальной производительности от хранимых решателей, используйте аргумент `save results`:

```
m = pyo.ConcreteModel()
m.x = pyo.Var()
m.y = pyo.Var()
m.obj = pyo.Objective(expr=m.x**2 + m.y**2)
m.c = pyo.Constraint(expr=m.y >= -2*m.x + 5)
opt = pyo.SolverFactory('gurobi_persistent')
opt.set_instance(m)
results = opt.solve(save_results=False)
```

Отметим, что если флаг `save_results` равен `False`, то показанный ниже код не поддерживается.

```
results = opt.solve(save_results=False, load_solutions=False)
if results.solver.termination_condition == pyo.TerminationCondition.optimal:
    try:
        m.solutions.load_from(results)
    except AttributeError:
        print('AttributeError was raised')
```

При этом печатается сообщение

```
AttributeError was raised
```

Однако такой код работает:

```
results = opt.solve(save_results=False, load_solutions=False)
if results.solver.termination_condition == pyo.TerminationCondition.optimal:
    opt.load_vars()
```

Кроме того, в модель можно загрузить подмножество значений переменных:

```
results = opt.solve(save_results=False, load_solutions=False)
if results.solver.termination_condition == pyo.TerminationCondition.optimal:
    opt.load_vars([m.x])
```

9.3.5. Пример

В этом разделе мы перепишем пример с перебором значений параметра из раздела 9.1.3, применив интерфейс с хранимым решателем. Перебор выполняет следующая функция.

```
def solve_parametric_persistent():
    m = create_warehouse_model(num_locations=50, num_customers=50)
    opt = pyo.SolverFactory('gurobi_persistent')
    opt.set_instance(m)
    p_values = list(range(1, 31))
```

```
    obj_values = list()
    for p in p_values:
        m.P.value = p
        opt.remove_constraint(m.num_warehouses)
        opt.add_constraint(m.num_warehouses)
        res = opt.solve(save_results=False)
        assert_optimal_termination(res)
        obj_values.append(res.problem.lower_bound)
```

Мы добавили несколько строк по сравнению с функцией из раздела 9.1.3, а именно обращения к методам интерфейса с решателем `set_instance`, `remove_constraint`, `add_constraint`.

Ниже продемонстрирован хронометраж этой функции:

```
timer.tic()
solve_parametric_persistent()
timer.toc('Finished parameter sweep with persistent interface')
```

Результат получился таким:

```
[ 22.79] Resetting the tic/toc delta timer
[+ 0.91] finished parameter sweep with persistent interface
```

Как видим, эта функция приблизительно в 8 раз быстрее варианта без хранимого решателя (0.91 с и 7.28 с). Напомним, что производительность сильно зависит от приложения (см. раздел 9.3.1).

9.4. Разреженные множества индексов

Часто в модели используется только часть декартова произведения нескольких множеств индексов. Так, иногда множество элементов по одному измерению зависит от значения по другому измерению, в результате чего образуются «зубчатые множества».

Например, пусть автору модели нужно задать ограничение, выполняющееся для

$i \in \mathcal{I}; k \in \mathcal{K}; v \in \mathcal{V}_k$.

Добиться этого можно разными способами, один из которых – создать множество кортежей, содержащее все допустимые значения пар «`model.k`, `model.V[k]`». Это показано в следующем примере, где используется зубчатое множество `KV`.

```
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.I = pyo.Set()
model.K = pyo.Set()
```

```
model.V = pyo.Set(model.K)

def kv_init(m):
    return ((k,v) for k in m.K for v in m.V[k])
model.KV = pyo.Set(dimen=2, initialize=kv_init)

model.a = pyo.Param(model.I, model.K)

model.y = pyo.Var(model.I)
model.x = pyo.Var(model.I, model.KV)

# включить ограничение вида
# x[i,k,v] <= a[i,k]*y[i] для i из I, k из K, v из V[k]
def c1Rule(m,i,k,v):
    return m.x[i,k,v] <= m.a[i,k]*m.y[i]
model.c1 = pyo.Constraint(model.I, model.KV, rule=c1Rule
```

Альтернативная стратегия – объявить ограничение, индексированное множествами I, K, V, а затем использовать метод `pyo.Constraint.Skip()`, чтобы пропускать отсутствующие индексы. Однако при большом числе измерений и для больших множеств это может привести к значительному замедлению.

Глава 10

Абстрактные модели и их решение

В этой главе описывается, как объявлять и использовать объект типа `AbstractModel` и файлы данных для инициализации абстрактных моделей. И в конце главы описывается команда `pyomo`, которая значительно упрощает решение абстрактной модели с использованием файлов данных. Хотя конкретные и абстрактные модели предлагают схожую функциональность, в последних проводится строгое различие между формулировкой модели и ее данными, что концептуально правильно и в некоторых контекстах практически полезно.

10.1. Общие сведения

Во многих примерах мы используем функцию, которая принимает данные и возвращает объект `ConcreteModel`. Это позволяет разделить понятия модели и данных. Класс `AbstractModel` в Pyomo обеспечивает такое разделение благодаря тому, что модель заполняется данными только после создания объекта абстрактной модели.

10.1.1. Абстрактные и конкретные модели

Pyomo поддерживает две стратегии объявления моделей: конкретные модели, в которых компоненты модели конструируются сразу, и абстрактные модели, в которых конструирование компонентов отложено. Абстрактные модели отражают структуру многих формулировок математической оптимизации. Например, формулировка задачи о расположении складов в разделе 3.2 представлена в общем виде и описывает целый класс задач оптимизации. Однако мы не можем решить эту задачу, пока не будут заданы конкретные данные (N, M, d и P). Решателю необходимо передать конкретный экземпляр задачи (вместе с данными).

В Pyomo абстрактная модель сначала объявляется, а конструирование компонентов откладывается до того момента, когда будут загружены данные и Pyomo создаст экземпляр модели. Этот подход к моделированию показан на рис. 10.1 вверху. Создается объект `AbstractModel`, затем Pyomo передаются данные для конкретной задачи, после чего Pyomo выполняет процесс конструирования и создает экземпляр модели со всеми переменными, выражениями ограничений и целевых функций, который можно передать решателю. Для этого нужно два прохода: на первом проходе модель объявляется, а на втором конструируется с использованием отдельно заданных значений данных. Для поддержки отложенного конструирования модель следует определять с применением правил конструирования.

С другой стороны, конкретные модели поддерживают программное создание экземпляра модели, при котором компоненты конструируются и инициализируются на первом проходе, когда Python исполняет скрипт модели. Этот подход показан на рис. 10.1 внизу. Создается объект `ConcreteModel`, и данные должны присутствовать *до* объявления каждого компонента. Когда Python исполняет скрипт модели, конкретный экземпляр модели и его компоненты создаются сразу же, как только Python встречает объявление компонента. После обработки файла интерпретатором Python модель готова к передаче решателю (т. е. мы имеем один проход). В этот момент объект `ConcreteModel` содержит конкретный экземпляр модели.

ПРИМЕЧАНИЕ Правила конструирования можно использовать и с конкретными моделями (правила выполняются сразу, как только встретятся в Python-файле модели).

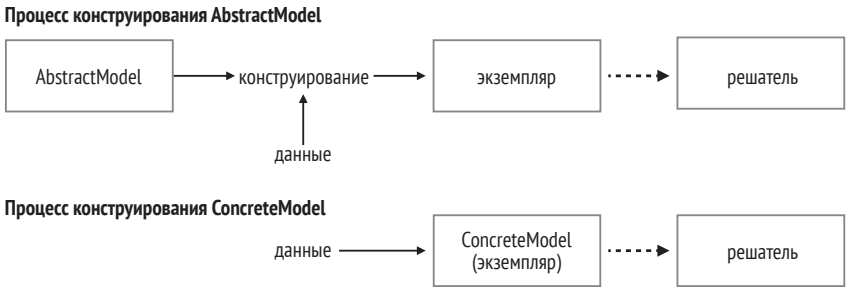

Рис. 10.1 ❖ На этом рисунке описан процесс конструирования абстрактной и конкретной моделей. Вверху представлен декларативный стиль описания абстрактных моделей. Сначала создается объект `AbstractModel`. Затем, получив конкретные данные, Pyomo выполняет процесс конструирования, создавая *экземпляр* модели, который можно передать решателю. Внизу показан программный (императивный) стиль создания конкретных моделей. По мере выполнения скрипта модели интерпретатором Python объекты компонентов конструируются немедленно с использованием ранее объявленных данных. Конкретный *экземпляр* модели готов к передаче решателю сразу после завершения первого прохода

Выбор типа модели (`AbstractModel` или `ConcreteModel`) – в основном дело вкуса. Самое важное различие между использованием `AbstractModel` и `ConcreteModel` связано с заданием данных. При использовании `AbstractModel` имена

и структура данных модели (но не сами значения) объявляются до конструирования. Это значит, что Pyomo знает о существовании множеств и параметров, которые встретятся при конструировании экземпляра. Еще важнее, что Pyomo знает имена и типы этих величин и имеет некоторую информацию о связях между ними (например, что переменная *y* индексирована множеством *N*). Это дает пользователю возможность задавать данные, используя все поддерживаемые Pyomo форматы данных и обращаясь к этим величинам по именам. В Pyomo есть много способов передать данные абстрактной модели, в т. ч. с помощью *файлов команд данных* для множеств и параметров. Синтаксис файлов команд данных Pyomo очень похож на синтаксис, поддерживаемый AMPL [2].

В конкретных моделях использовать встроенные в Python типы данных при создании экземпляра проще. Поэтому если вам больше нравится строить модели в процедурном программном окружении (как Python или MATLAB) или если ваше приложение требует более сложного технологического процесса, чем поддерживает команда `pyomo`, то лучше остановиться на `ConcreteModel`. В особенности если данные легко загрузить с помощью других Python-пакетов (например, pandas).

> **ПРИМЕЧАНИЕ** В общем случае `AbstractModel` проще для пользователей, незнакомых с Python или предпочитающих работать в каком-то более традиционном окружении AML. `ConcreteModel` часто требует от пользователя кодирования загрузки данных в исходном формате на Python (например, с помощью имеющегося Python-пакета), но зато предлагает более прозрачный контроль над порядком выполнения.

10.1.2. Абстрактная формулировка модели (H)

Рассмотрим модель (H) из раздела 2.2, которая здесь для удобства повторена:

$$\max_x \sum_{i\in\mathcal{A}} h_i \cdot (x_i - (x_i/d_i)^2) \quad (H)$$
$$\text{при условиях } \sum_{i\in\mathcal{A}} c_i x_i \leqslant b$$
$$0 \leqslant x_i \leqslant u_i,\ i \in \mathcal{A}$$

Поскольку модель (H) абстрактная, на Pyomo ее естественно выразить с помощью класса Pyomo `AbstractModel`.

Рассмотрим следующую модель Pyomo для этой задачи:

```
# AbstractH.py - реализация модели (H)
import pyomo.environ as pyo

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

model.A = pyo.Set()

model.h = pyo.Param(model.A)
model.d = pyo.Param(model.A)
model.c = pyo.Param(model.A)
model.b = pyo.Param()
```

```
model.u = pyo.Param(model.A)

def xbounds_rule(model, i):
    return (0, model.u[i])
model.x = pyo.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 = pyo.Objective(rule=obj_rule, sense=pyo.maximize)

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

Имея конкретные параметры модели, интересно найти оптимальные значения `model.x`. Есть много способов предоставить Pyomo данные для абстрактной модели. Ниже приведены данные (сохраненные в файле `AbstractH.dat`), отвечающие представлению одного из авторов этой книги о счастье:

```
# Файл данных Pyomo для модели 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 ;
```

Это файл данных Pyomo, который включает команды `set` и `param`, очень напоминающие команды AMPL. Описание этих команд данных приведено в разделе 10.3.

Для оптимизации абстрактной модели добавим следующие строки в определяющий ее файл Python:

```
opt = pyo.SolverFactory('glpk')

instance = model.create_instance("AbstractH.dat")
results = opt.solve(instance) # решает и обновляет экземпляр
instance.display()
```

Или же можно решить модель, воспользовавшись командой `pyomo`, как описано в разделе 10.2.

10.1.3. Абстрактная модель для задачи о расположении складов

Задачу о расположении складов (см. раздел 3.2) можно представить в виде абстрактной модели следующим образом:

```
# wl_abstract.py: версия задачи о расположении складов в виде AbstractModel
import pyomo.environ as pyo

model = pyo.AbstractModel(name="(WL)")
model.N = pyo.Set()
model.M = pyo.Set()
model.d = pyo.Param(model.N,model.M)
 model.P = pyo.Param()
model.x = pyo.Var(model.N, model.M, bounds=(0,1))
model.y = pyo.Var(model.N, within=pyo.Binary)

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 = pyo.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 = pyo.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 = pyo.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 = pyo.Constraint(rule=num_warehouses_rule)
```

Множества объявлены как компоненты Pyomo `Set`, а параметрические данные – как компоненты `Param` без указания способа предоставления данных. Мы проинформировали Pyomo о том, что компоненты `Param` и `Var` будут индексированы множествами, но содержимое этих множеств не объявлено.

Правило конструирования целевой функции определено в виде функции Python `obj_rule`, а затем `model.obj` объявлена как компонент Pyomo `Objective`. Поскольку это абстрактная модель, правило `obj_rule` пока не вызывается. В этой точке Pyomo знает, какое правило вызывать для конструирования компонента целевой функции, но не вызывает конструктор, т. к. это абстрактная модель. Правила конструирования ограничений и компоненты ограничений объявляются аналогично.

Данные могут быть представлены в нескольких форматах. Например, можно использовать такой файл данных Pyomo:

```
# wl_data.dat: формат данных Pyomo для задачи о расположении складов

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 ;
```

Этот скрипт можно выполнять командой `python`, но ничего полезного при этом не произойдет. Скрипт объявляет модель, но не определяет данные модели и не создает экземпляр задачи для решателя. Акт применения файла данных к абстрактной модели можно явно запрограммировать в коде на Python или выполнить с помощью команды `pyomo`. Например:

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

Флаг `--summary` можно использовать для вывода более подробной информации о решении.

Команда `pyomo` выполняет скрипт `wl_abstract.py`, чтобы создать объект `AbstractModel` с именем `model`. Этот объект модели содержит объявленные ранее компоненты моделирования. Затем `pyomo` читает файл данных `wl_data.dat`, применяет эти данные к компонентам `Set` и `Param` в том же порядке, в каком они объявлены в модели. Далее `pyomo` конструирует все остальные компоненты в порядке объявления: переменные, целевую функцию и ограничения. После того как модель сконструирована, `pyomo` вызывает решатель.

Абстрактные модели можно использовать в скриптах, но конкретные экземпляры должны создаваться из объекта `AbstractModel` путем вызова метода `create_instance`. В следующем примере мы берем `AbstractModel`, конструируем экземпляр, пользуясь файлом данных `wl_data.dat`, решаем модель и печатаем результаты:

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

Если вы не хотите писать скрипт сами, а предпочитаете использовать команду `pyomo`, то добавлять эти строки не нужно.

10.2. Команда pyomo

В состав дистрибутива Pyomo входит скрипт `pyomo`, выполняющий различные подкоманды для облегчения работы с Pyomo. В Pyomo 6.0 поддерживаются следующие подкоманды:

- `check`: проверяет модель на наличие ошибок. Особенно это полезно для проверки логики правил в абстрактных моделях;
- `convert`: преобразует модель Pyomo в другой формат, например в lp- или nl-файл;
- `help`: печатает информацию о конфигурации и установке Pyomo. Например, флаг `-s` печатает информацию о доступных решателях:

  ```
  pyomo help -s
  ```

- `run`: выполняет команду из каталога Pyomo bin (или Scripts). Например, это дает удобный механизм запуска Python из-под Pyomo:

  ```
  pyomo run python
  ```

- `solve`: конструирует и оптимизирует модель;
- `test-solvers`: прогоняет различные тесты для проверки возможностей решателя.

В следующих разделах документируются подкоманды `check`, `convert`, `help` и `solve`, которые имеют различные параметры.

10.2.1. Подкоманда help

Подкоманда `help` печатает информацию о возможностях Pyomo, в т. ч. об установленных плагинах и доступных решателях. Если задан флаг `-h`, то печатается информация о поддерживаемых параметрах, в т. ч.:

- `--checkers`: печатаются установленные вместе с Pyomo модули проверки моделей;
- `--commands, -c`: печатается список команд, установленных вместе с Pyomo. Хотя большая часть функциональности Pyomo доступна с помощью команды `pyomo`, кое-что вынесено в отдельные команды;
- `--components`: печатается список доступных компонентов моделирования и виртуальных множеств;
- `--data-managers, -d`: печатаются интерфейсы данных, поддерживаемые классом `DataPortal`. Они позволяют импортировать данные из файлов данных Pyomo;
- `--info, -i`: печатается информация о переменной окружения PATH и об установке Python. Эта команда помогает диагностировать проблемы, возникающие при выполнении Pyomo;
- `--solvers, -s`: печатается информация о решателях и диспетчерах решателей, которую можно использовать для оптимизации моделей Pyomo. Отметим, что информация о решателях NEOS включается, только если Pyomo может подключиться к серверу NEOS;
- `--transformations, -t`: печатаются преобразования моделей, поддерживаемые Pyomo;
- `--writers, -w`: печатаются модули записи моделей, поддерживаемые Pyomo, точнее различные файловые форматы, в которые может быть преобразована модель.

Например, чтобы получить список доступных преобразований, выполните команду

```
pyomo help --transformations
```

10.2.2. Подкоманда solve

Подкоманда `pyomo solve` автоматически выполняет модель Pyomo в следующем порядке.

1. Сконструировать модель.
2. Прочитать данные экземпляра (если таковые имеются).
3. Сгенерировать экземпляр модели (если модель абстрактная).
4. Применить к экземпляру модели простые препроцессоры.
5. Применить к экземпляру модели решатель.
6. Загрузить результаты в экземпляр модели.
7. Отобразить результаты решателя.

Например, следующая команда решает задачу о расположении складов, определенную в файле `wl_abstract.py`, применяя ЛП-решатель `glpk` с данными из файла `wl_data.dat`:

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

На шаге конструирования требуется *файл модели Pyomo*, представляющий собой Python-файл с определением объекта модели. Поэтому подкоманду `solve` можно рассматривать как обобщенный скрипт для анализа модели, определенной в файле модели Pyomo. У нее имеется ряд аргументов командной строки для настройки процесса оптимизации; для получения информации о них задайте флаг `--help`.

Однако подкоманду `solve` можно также выполнить, задав конфигурационный файл в формате YAML или JSON[1], что устраняет необходимость в задании аргументов командной строки. Рассмотрим следующий конфигурационный файл:

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

Этот конфигурационный файл можно использовать для настройки выполнения подкоманды `pyomo`:

```
pyomo solve concrete1.yaml
```

[1] YAML и JSON – стандарты сериализации данных. Для JSON в Python есть встроенная поддержка, а информацию об этом формате можно получить на сайте www.json.org. Конфигурационные файлы в формате YAML поддерживаются, если установлен пакет `PyYAML`, а информация о YAML имеется на сайте www.yaml.org.

Он задает ту же логику, что и первая команда в предыдущем абзаце, а следующий конфигурационный файл задает ту же логику, что вторая команда:

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

При использовании конфигурационного файла не нужны никакие аргументы командной строки, потому что каждому из них соответствует элемент конфигурационного файла. Более того, существуют конфигурационные параметры, которые можно выразить только в конфигурационном файле. Шаблонный конфигурационный файл можно сгенерировать с помощью параметра `--generate-config-template`.

Таблица 10.1. Параметры подкоманды pyomo solve

Параметр	Описание
`-c, --catch-errors`	Завершать программу в случае исключения и печатать стек вызовов
`--json`	Сохранять результаты в формате JSON
`-k, --keepfiles`	Сохранять временные файлы
`-l, --log`	Печатать журнал решателя после выполнения оптимизации
`--logfile FILE`	Перенаправлять вывод в указанный файл
`--logging LEVEL`	Уровень протоколирования: quiet, warning, info, verbose, debug
`--model-name NAME`	Имя объекта модели, создаваемого в указанном модуле Python
`--path PATH`	Путь к Python-файлам Pyomo
`--report-timing`	Печатать хронометраж во время конструирования модели
`--results-format FORMAT`	Формат результата: json или yaml
`--save-results FILE`	Имя файла, в котором сохраняются результаты
`--show-results`	Печатать объект `results` после оптимизации
`--solver SOLVER`	Имя решателя
`--solver-executable FILE`	Исполняемый файл, используемый интерфейсом с решателем
`--solver-io FORMAT`	Тип ввода-вывода, используемый решателем. Различные решатели поддерживают разные типы ввода-вывода, наиболее употребительны следующие: lp – генерировать LP-файлы, np – генерировать NP-файлы, python – прямой интерфейс с Python
`--stream-output`	Потоком выводить все, что печатает решатель, чтобы иметь информацию о ходе его работы
`--solver-options STRING`	Строка параметров решателя

Таблица 10.1 *(окончание)*

Параметр	Описание
`--solver-suffix SUFFIXES`	Суффиксы решения, которые будет извлекать решатель (например, rc, dual или slack)
`--summary`	Напечатать сводную информацию об окончательном решении после оптимизации
`--symbolic-solver-labels`	В интерфейсе с решателем использовать символические имена, выведенные на основе модели. Например, «my special variable[1 2 3]» вместо «v1». При использовании ASL-решателей этот параметр генерирует соответствующие файлы .row (ограничения) и .col (переменные)
`--tempdir TEMPDIR`	Каталог, в котором создаются временные файлы

Параметры `--help` и `--generate-config-template` подкоманды `solve` требуют параметра `--solver`. Эти два параметра выдают зависящую от решателя сводную информацию об аргументах командной строки и о конфигурационных файлах соответственно. Например, следующая команда печатает аргументы командной строки для решателя `glpk`:

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

В табл. 10.1 перечислены основные, наиболее употребительные параметры подкоманды `solve`.

10.2.2.1. Задание объекта модели

Файлом модели может быть любой Python-скрипт, но команда `pyomo solve` ожидает, что он генерирует объект, содержащий модель Pyomo. Подкоманда `solve` исполняет файл модели в предложении Python `import`, поэтому он интерпретируется как любой Python-файл.

В простейшем случае файл модели Pyomo содержит предложения Python, которые создают объект модели и сохраняют его в переменной `model`. Рассмотрим, к примеру, следующую модель линейного программирования (ЛП):

```
# abstract5.py
import pyomo.environ as pyo

model = pyo.AbstractModel()

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

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

def obj_rule(model):
    return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = pyo.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 = pyo.Constraint(model.M, rule=con_rule)
```

Эта абстрактная модель Pyomo хранится в переменной `model`.

Если пользователь определяет свою модель в переменной с другим именем, то об этом нужно сообщить Pyomo с помощью параметра `--model-name`. Так, мы можем модифицировать предыдущий пример, сохранив модель в переменной `Model`:

```
# abstract6.py
import pyomo.environ as pyo

Model = pyo.AbstractModel()

Model.N = pyo.Set()
Model.M = pyo.Set()
Model.c = pyo.Param(Model.N)

Model.a = pyo.Param(Model.N, Model.M)
Model.b = pyo.Param(Model.M)
Model.x = pyo.Var(Model.N, within=pyo.NonNegativeReals)

def obj_rule(Model):
    return sum(Model.c[i]*Model.x[i] for i in Model.N)
Model.obj = pyo.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 = pyo.Constraint(Model.M, rule=con_rule)
```

Для оптимизации этой модели выполните команду

```
pyomo solve --solver=glpk --model-name=Model abstract6.py abstract6.dat
```

Эта возможность не только дает пользователю дополнительную гибкость, но и позволяет определять в одном файле Pyomo несколько моделей, а затем выбирать для оптимизации одну из них при выполнении подкоманды `solve`.

10.2.2.2. Выбор данных с помощью пространств имен

В разделе 10.3.4 мы познакомимся с командой `namespace` в файлах данных Pyomo. Она служит для определения блоков команд данных, которые факультативно интегрируются в модель. Подкоманда `solve` принимает параметр `--namespace` (короткий синоним `--ns`) для задания одного или нескольких пространств имен, используемых при конструировании экземпляра абстрактной модели. Например, команда

```
pyomo solve --solver=glpk --namespace=data1 abstract5.py abstract5-ns1.dat
```

создает и оптимизирует абстрактную модель в файле abstract5.py, пользуясь следующими командами данных:

```
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 ;
}
```

Эта команда определяет пространство имен `data1`, для которого оптимальное решение равно 0.8. Аналогично команда

```
pyomo solve --solver=glpk --namespace=data2 abstract5.py abstract5-ns1.dat
```

создает и оптимизирует ту же модель, но с пространством имен `data2`, для которого оптимальное решение равно 8. В наборе данных `data2` другое множество индексов и другие коэффициенты целевой функции.

В примере выше показано, как пространства имен дают пользователю возможность задавать разные наборы данных в одном файле. Отметим, что модель можно конструировать с данными из разных пространств имен, в т. ч.

данными, не помещенными в пространство имен. Рассмотрим следующие команды данных:

```
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 ;
}
```

Здесь мы видим четыре пространства имен и данные вне пространства имен. Команда

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

создает и оптимизирует абстрактную модель в файле `abstract5.py`, пользуясь данными из пространств имен `c1` и `data2`, а также значением `N`, находящимся вне всех пространств имен. Отметим, что если в нескольких пространствах

имен имеются данные для одного и того же компонента, то этот компонент инициализируется данными из первого пространства имен. Если же ни в одном пространстве имен нет соответствующей команды данных, то для инициализации компонента берутся данные, находящиеся вне пространств имен.

10.2.2.3. Настройка технологического процесса Pyomo

Шаги, выполняемые подкомандой `solve`, определяют общий процесс конструирования и оптимизации модели. Этот технологический процесс можно настроить, применяя различные функции обратного вызова, определенные в файле модели Pyomo. Эти функции позволяют задать дополнительные шаги анализа, а также заменить некоторые шаги по умолчанию в стандартном процессе.

Таблица 10.2. Функции обратного вызова, которые можно использовать в файле модели Pyomo для настройки технологического процесса в подкоманде `pyomo solve`

Функция	Описание
`pyomo_preprocess`	Выполнить шаг препроцессирования до конструирования модели
`pyomo_create_model`	Сконструировать и вернуть объект модели
`pyomo_create_modeldata`	Сконструировать и вернуть объект `DataPortal`
`pyomo_print_model`	Вывести информацию об объекте модели
`pyomo_modify_instance`	Модифицировать экземпляр модели
`pyomo_print_instance`	Вывести информацию об экземпляре модели
`pyomo_save_instance`	Сохранить экземпляр модели
`pyomo_print_results`	Напечатать результаты оптимизации
`pyomo_save_results`	Сохранить результаты оптимизации
`pyomo_postprocess`	Выполнить шаг постпроцессирования после оптимизации модели

В табл. 10.2 перечислены функции обратного вызова команды `pyomo` и их предназначение. Каждая функция принимает один или несколько именованных аргументов вида `keyword=value`. Например, функция `pyomo_print_results` принимает три аргумента: `options`, `instance` и `results`.

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

У функций обратного вызова, описанных в табл. 10.2, есть несколько стандартных аргументов. Аргумент `options` – расширенный словарь Python, содержащий параметры командной строки, передаваемые подкоманде `solve`. Аргумент `model` – объект модели, а аргумент `instance` – сконструированный экземпляр этой модели. Если пользователь определил конкретную модель, то аргументы `model` и `instance` – один и тот же объект. Прочие аргументы описаны вместе с соответствующими функциями обратного вызова.

pyomo_preprocess

Эта функция обратного вызова вызывается до конструирования модели и выполняет шаги препроцессирования. У этой функции всего один аргумент: `options`. Например, следующая функция просто печатает параметры командной строки:

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

pyomo_create_model

Эта функция обратного вызова предназначена для конструирования модели. Она принимает два аргумента: `options` и `model_options`. Второй из них содержит параметры конструирования модели, заданные в параметре командной строки, – `--model-options`. Эта функция должна возвращать созданный объект модели, абстрактной или конкретной. Например, следующая функция создает модель, импортируя файл `abstract6.py`, а затем возвращает объект `Model`:

```
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

Эта функция обратного вызова создает объект данных, используемый для создания экземпляра модели. Объекты данных модели полезны в случаях, когда для конструирования модели нужно задавать несколько различных данных. Функция принимает два аргумента, `options` и `model`, и должна возвращать объект типа `DataPortal`. Например, следующая функция обратного вызова создает объект `DataPortal` из данных в файле `abstract6.dat`:

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

pyomo_print_model

Эта функция обратного вызова печатает абстрактную модель перед созданием экземпляра модели. Она принимает два аргумента: `options` и `model`. В следующем примере вызывается метод `pprint`, который печатает подробную информацию об абстрактной модели:

```
def pyomo_print_model(options=None, model=None):
    if options['runtime']['logging']:
        model.pprint()
```

pyomo_modify_instance

Эта функция обратного вызова модифицирует экземпляр модели, после того как он был сконструирован. Она принимает три аргумента: `options`, `model` и `instance`. Следующая функция исправляет переменную после конструирования модели:

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

pyomo_print_instance

Эта функция обратного вызова печатает экземпляр модели Pyomo. Она используется для печати экземпляра конкретной модели, а не абстрактной модели. Функция принимает два аргумента: `options` и `instance`. В следующем примере вызывается метод `pprint`, который печатает подробную информацию об экземпляре модели:

```
def pyomo_print_instance(options=None, instance=None):
    if options['runtime']['logging']:
        instance.pprint()
```

pyomo_save_instance

Эта функция обратного вызова сохраняет экземпляр модели Pyomo. Она принимает два аргумента: `options` и `instance`. Отметим, что Pyomo не определяет, как сохраняется модель. Однако очень удобно использовать для этой цели имеющийся в Python механизм pickle:

```
def pyomo_save_instance(options=None, instance=None):
    OUTPUT = open('abstract7.pyomo','w')
    OUTPUT.write(str(pickle.dumps(instance)))
    OUTPUT.close()
```

pyomo_print_results

Эта функция обратного вызова печатает результаты оптимизации. Она принимает три аргумента: `options`, `instance` и `results`. Объект `results` поддерживает вывод сводной информации о найденных решениях, статистики решателя и т. д. в форматах JSON и YAML. Таким образом, эта функция может просто распечатать данные:

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

Однако у подкоманды `solve` имеется параметр `--print-results`, который выполняет эту операцию. А вообще, эта функция обратного вызова включается для того, чтобы пользователь мог представить результаты оптимизации с учетом специфики задачи.

pyomo_save_results

Эта функция обратного вызова сохраняет результаты оптимизации. Она принимает три аргумента: `options`, `instance` и `results`. Функция просто печатает результаты в файл:

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

У подкоманды `solve` имеется параметр `--save-results`, который выполняет эту операцию. А вообще, эта функция обратного вызова включается для того, чтобы пользователь мог сохранить результаты оптимизации с учетом специфики задачи.

pyomo_postprocess

Эта функция обратного вызова выполняется после оптимизации, чтобы выполнить шаги постпроцессирования. Она принимает три аргумента: `options`, `instance` и `results`. Например, следующая функция печатает простую сводку результатов оптимизации:

```
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(pyo.value(instance.obj)))
```

10.2.2.4. Настройка поведения решателя

Общий технологический процесс, поддерживаемый подкомандой `solve`, включает выполнение решателя для оптимизации (или иного анализа) модели. Для управления поведением решателя используются различные параметры командной строки. Параметр `--solver` позволяет задать имя решателя в одном из двух вариантов: имя исполняемого файла (путь к нему определяется переменной окружения PATH) или предопределенный интерфейс с решателем.

Предполагается, что исполняемые файлы выполняют ввод-вывод с помощью NL-файлов. Поэтому для оптимизации подойдет любой решатель, исполняемый файл которого собран с библиотекой AMPL.

Параметры решателя задаются единообразно в параметре командной строки `--solver-options`. Его значением является строка, интерпретируемая как набор пар (параметр=значение). Например, в следующей команде решателю glpk передается параметр `mipgap`:

```
pyomo solve --solver=glpk --solver-options='mipgap=0.01' concrete1.py
```

Кроме того, для задания максимального времени работы решателя можно использовать параметр `--timelimit`. Обычно он передается самому решателю, поэтому лимит времени задается способом, не зависящим от решателя.

Результаты формируются на основе информации, предоставленной решателем, и, возможно, информации из журнала решателя. По умолчанию Pyomo сохраняет сведения о найденных решателем значениях переменных. Но иногда пользователю нужна дополнительная информация, например двойственные значения для ограничений в линейной программе. Из соображений производительности эти данные не собираются автоматически подкомандой `solve`, но для задания имен нужных данных можно воспользоваться параметром `--solver-suffixes`. *Суффикс* – это просто данные для ограничения или переменной, найденные в результате применения решателя. Суффиксы можно задавать по имени или с помощью регулярного выражения. Например, следующая команда запрашивает все суффиксы, сгенерированные решателем:

```
pyomo solve --solver=glpk --solver-suffix='.*' concrete2.py
```

В настоящее время Pyomo поддерживает следующие суффиксы:

- `dual` – двойственные значения ограничений;
- `rc` – приведенные затраты;
- `slack` – значения невязок для ограничений.

Отметим, что конкретный решатель может предоставлять только подмножество этих суффиксов.

Для архивации временных файлов, создаваемых Pyomo, можно использовать параметры `--tempdir` и `--keepfiles`. По умолчанию Pyomo пользуется системными каталогами для временных файлов. Параметр `--tempdir` задает каталог, в котором должны создаваться временные файлы. По умолчанию временные файлы удаляются после завершения оптимизации. Параметр `--keepfiles` отменяет удаление, поэтому пользователь может посмотреть, какие данные Pyomo передала оптимизатору.

10.2.2.5. Анализ результатов решателя

Параметр `--postprocess` позволяет задать модуль Python, выполняемый после завершения решателя. Обычно эта возможность используется для определения шагов постпроцессирования с целью интерпретации результатов с учетом специфики задачи.

Для определения шагов постпроцессирования следует объявить в модуле Python функцию `pyomo_postprocess`. На рис. 10.2 приведен пример функции постпроцессирования, которая записывает найденные решения в файл в формате CSV.

10.2.2.6. Управление диагностической печатью

У подкоманды `solve` имеется ряд параметров для управления выводом диагностической и другой полезной информации о технологическом процессе.

По умолчанию вывод подкоманды `solve` представляет собой краткую сводку основных выполненных шагов. Параметры `--log` и `--stream-output` позволяют напечатать вывод решателя. Параметр `--log` печатает вывод решателя после его завершения, а параметр `--stream-output` печатает вывод по мере

его генерирования. Параметры `--summary` и `--show-results` служат для печати различной сводной информации о результатах оптимизации. Параметр `--summary` печатает сводную информацию о модели Pyomo после загрузки результатов.

```
import csv

def pyomo_postprocess(options=None, instance=None, results=None):
    #
    # Собрать данные
    #
    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
    #
    # Записать в CSV-файл, по одной строке на каждое решение.
    # Первый столбец содержит значение функции, остальные -
    # значения отличных от нуля переменных.
    #
    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()
```

Рис. 10.2 ❖ Плагин постпроцессирования, записывающий найденные решения в CSV-файл

Параметр `--show-results` печатает окончательные результаты. Если установлен пакет PyYAML, то по умолчанию результаты печатаются в формате YAML и сохраняются в файле `results.yml`. В противном случае результаты печатаются в формате JSON и сохраняются в файле `results.json`. Если нужно, чтобы для результатов использовался формат JSON, даже если установлен пакет PyYAML, задайте флаг `--json`. Чтобы задать другой файл результатов, воспользуйтесь параметром `--save-results`.

В Pyomo используется стандартная система протоколирования Python для управления печатью сообщений. По умолчанию печатаются сообщения об ошибках и предупреждения. Параметр `--quiet` подавляет печать всего, кроме сообщений об ошибках. Параметр `--warning` включает печать предупреждений. Параметр `--info`, помимо сообщений об ошибках и предупреждений, печатает еще информационные сообщения.

Параметр `--verbose` включает печать отладочных сообщений. Его можно повторить несколько раз, чтобы включить печать сообщений для различных частей Pyomo: (1) отладка только Pyomo и (2) отладка всех пакетов Pyomo. Параметр `--debug` включает отладочную печать, а при возникновении исключений добавляет в распечатку стек программы.

10.2.3. Подкоманда convert

Многие оптимизаторы, поддерживаемые Pyomo, читают временный файл, генерируемый Pyomo в стандартном для конкретной задачи формате. Например, формат `NL` понимают решатели, используемые совместно со средством моделирования AMPL, а формат `LP` – различные коммерческие и открытые решатели задач целочисленного программирования.

Часто бывает полезно генерировать эти файлы напрямую – как для того, чтобы диагностировать проблемы модели, так и для прямого управления выполнением решателя. Подкоманду `convert` можно использовать для преобразования модели Pyomo в стандартный формат. Например, рассмотрим команду

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

Она преобразует модель в файле `concrete1.py` в формат LP и сохраняет в файле `unknown.lp`:

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

Для задания имени файла можно также использовать параметр `--output`, при этом расширение имени определяет формат файла. Например, команда

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

создает файл `concrete1.lp`, в котором модель из файла `concrete1.py` представлена в формате `LP`.

Команда

```
pyomo help -w
```

выводит перечень файловых форматов, поддерживаемых Pyomo.

10.3. Команды данных для AbstractModel

Компоненты `Set` и `Param` модели Pyomo служат для определения значений данных, используемых при конструировании ограничений и целевых функций. В предыдущих главах было показано, что эти компоненты необязательны для разработки сложных моделей. Однако компоненты `Set` и `Param` можно использовать в абстрактных объявлениях данных, когда никакие значения не задаются, например:

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

Командные файлы данных можно использовать для инициализации объявлений данных в моделях Pyomo, особенно абстрактных. Отметим, однако, что сложные отображения чаще реализуются посредством скриптов, а не файлов данных.

В файлах команд данных Pyomo используется предметно-ориентированный язык, синтаксис которого близок к синтаксису команд данных AMPL [2]. Файл команд данных состоит из последовательности команд, описывающих значения множеств и параметров или указывающих, как можно получить данные из внешних источников. Для объявления данных можно использовать следующие команды:

- команда `set` объявляет данные множества;
- команда `param` объявляет таблицу данных параметров, которая может также включать объявление данных множества, индексирующего параметрические данные;
- команда `load` загружает данные из внешних источников, например электронной таблицы или базы данных;
- команда `table` объявляет двумерную таблицу параметрических данных.

В командных файлах можно использовать также следующие команды:

- команда `include` задает файл команд данных, который обрабатывается немедленно;
- команды `data` и `end` не выполняют никаких действий, а просто обеспечивают совместимость со скриптами AMPL для определения команд данных.

Наконец, объявление `namespace` позволяет организовывать команды данных в именованные группы, каждая из которых может быть активирована или деактивирована на этапе конструирования модели.

Отметим, что между командами данных Pyomo и командами данных AMPL нет точного соответствия. Синтаксис и семантика команд `set` и `param` очень близки к AMPL. Однако они поддерживают только подмножество функциональности соответствующих команд AMPL. Поддержать другие команды AMPL невозможно, потому что в Pyomo команды данных рассматриваются как объявления данных, а в AMPL – как часть скриптового языка.

В следующих подразделах описывается синтаксис команд данных Pyomo, кроме команд `load` и `table`, описания которых имеются в онлайновой до-

кументации Pyomo. Синтаксис команд данных весьма разнообразен, и для иллюстрации мы приводим подробные примеры. Отметим, что все команды данных в Pyomo завершаются точкой с запятой, а пробелы в них не несут синтаксической нагрузки. Таким образом, команды данных могут располагаться в нескольких строках – знаки новой строки и табуляции игнорируются, а при их форматировании можно использовать сколько угодно пробелов.

10.3.1. Команда set

10.3.1.1. Простые множества

Команда данных `set` явно определяет элементы одного множества или массива множеств, т. е. индексированного множества. Одиночное множество задается списком значений своих элементов. Формально команда `set` имеет такой синтаксис:

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

Значениями элементов множества могут быть числа, простые строки или закавыченные строки.

- *Числовое значение* – это любая строка, которую Python может вычислить как число – целое, с плавающей точкой, в научной нотации или булево.
- *Простая строка* – это последовательность буквенно-цифровых символов.
- *Закавыченная строка* – это простая строка, заключенная в одиночные или двойные кавычки. Закавыченная строка может включать внутренние кавычки.

На значения в объявлении множества не налагается никаких ограничений. Множество может быть пустым или содержать произвольную комбинацию числовых и строковых значений. Проверка данных множества осуществляется при конструировании модели Pyomo, а не при разборе командного файла данных. Ниже приведены примеры допустимых команд `set`:

```
# Пустое множество
set A := ;

# Множество чисел
set A := 1 2 3;

# Множество строк
set B := north south east west;

# Множество смешанных типов
set C :=
0
-1.0e+10
'foo bar'
```

```
infinity
"100"
;
```

Отметим, что числовые значения автоматически преобразуются в значения Python – целые или с плавающей точкой – на этапе разбора спецификации данных. Кавычки можно использовать для определения строки, содержащей числовое значение. Но если строка задает числовое значение, Python преобразует его в числовой тип. Например, в множество C включена строка «100», но это значение преобразуется в числовое.

10.3.1.2. Множество кортежей

С помощью команды данных set можно также определять кортежи, применяя стандартную нотацию. Например, предположим, что множество A содержит 3-кортежи:

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

Тогда следующая команда set означает, что A содержит кортежи (1,2,3) и (4,5,6):

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

Или же можно просто перечислить элементы подряд в том же порядке:

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

Очевидно, что число элементов при таком синтаксисе должно быть кратно размерности множества.

Множества, состоящие из 2-кортежей, можно еще задавать в виде матрицы, обозначающей вхождение в множество. Например, следующая команда данных set объявляет 2-кортежи, входящие в A, применяя знак + для обозначения допустимых кортежей и знак - для обозначения недопустимых.

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

Эта команда данных объявляет следующие пять 2-кортежей: (‘A1’,1), (‘A1’,2), (‘A2’,3), (‘A3’,2), (‘A4’,1).

Наконец, множество кортежей можно определить кратко, воспользовавшись шаблонами кортежей, представляющими срез данных. Например, предположим, что множество A содержит 4-кортежи:

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

Следующая команда данных set объявляет группы кортежей, определенные с помощью шаблона и данных, подставляемых в этот шаблон:

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

Шаблон состоит из кортежа, содержащего один или несколько символов * вместо значения. Они представляют позиции, в которые подставляются значения из списка, следующего за шаблоном. В данном случае множество A будет состоять из следующих кортежей:

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

10.3.1.3. Массивы множеств

Команду данных set можно также использовать для объявления массивов множеств. Каждое входящее в массив множество должно быть объявлено отдельной командой set с таким синтаксисом:

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

Массивы множеств можно индексировать произвольным множеством, а значением индекса может быть число, нечисловая строка или список строк через запятую. Предположим, что множество A используется для индексирования множества B:

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

Тогда множество B индексируется объявленными элементами множества 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';
```

10.3.2. Команда param

Простые, или неиндексированные, параметры объявляются очевидным способом, как показывают следующие примеры:

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

Для определения параметров можно использовать числовые и строковые данные. Числовые данные задаются строкой, которую Python вычисляет как число; это могут быть целые числа, числа с плавающей точкой, числа в на-

учной нотации или булевы значения. Булевы значения можно задавать разными строками: TRUE, true, True, FALSE, false и False. Отметим, что параметры нельзя определять без данных, т. е. аналога пустого множества не существует.

Большая часть параметрических данных индексируется одним или несколькими множествами, а использовать команды данных для задания индексированных параметров можно несколькими способами.

10.3.2.1. Одномерные параметрические данные

Одномерные параметрические данные индексируются одним множеством. Пусть параметр B индексирован множеством A:

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

Команда данных param может задать значения B с помощью списка пар (индекс, значение):

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

Поскольку пробелы игнорируются, этот командный файл данных можно переписать в табличном формате:

```
set A := a c e;

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

В одной команде param можно определить несколько параметров. Например, предположим, что B, C, D – одномерные параметры, индексированные множеством A:

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

Значения этих параметров можно задать с помощью одной команды param, в которой вслед за этими именами параметров идет список их индексов и значений:

```
set A := a c e;

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

Значения в команде данных `param` интерпретируются как список списков, в котором каждый вложенный список состоит из индекса, за которым следует соответствующее числовое значение.

Отметим, что значения параметров необязательно задавать для всех индексов. Например, следующий файл данных написан корректно:

```
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
;
```

Индекс `g` в команде `param` опущен, и, значит, этот индекс не может встречаться в экземпляре модели с этими данными. В более сложных случаях отсутствие данных можно обозначать символом «.». Этот синтаксис полезен, когда нужно задать несколько параметров, но не для каждого параметра значения присутствуют для всех индексов, например:

```
set A := a c e;

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

В этом примере кратко представлены параметры, которые индексированы одним множеством, но значения имеют для разных индексов.

Отметим, что в этом файле данных элементы множества `A` задаются дважды: (1) при определении `A` и (2) при определении параметров. Имеется альтернативный синтаксис `param`, позволяющий совместить определение множества индексов с определением параметров:

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

Наконец, отметим, что с помощью ключевого слова `default` можно задать значения по умолчанию для отсутствующих данных:

```
set A := a c e;

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

Но значения по умолчанию можно задавать только при определении значений одного параметра.

10.3.2.2. Многомерные параметрические данные

Многомерные параметрические данные индексируются либо несколькими множествами, либо многомерным множеством. Предположим, что параметр B индексирован множеством A размерности 2:

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

Синтаксис команды param для задания списка значений индексов и параметров для B, по существу, не меняется:

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

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

Отсутствующие данные и значения по умолчанию обрабатываются для многомерных множеств индексов точно так же:

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

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

Как и раньше, несколько параметров можно определить в одной команде param. Пусть B, C, D – параметры, индексированные множеством A размерности 2:

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

Значения этих параметров можно определить в одной команде param, объявляющей имена параметров, за которыми следует список их индексов и значений:

```
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
;
```

Аналогично следующая команда param совмещает определение множества индексов с определением параметров:

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

Команда `param` поддерживает также матричный синтаксис задания значений параметра с помощью двумерного индекса. Пусть параметр `B` индексирован множеством `A` размерности 2:

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

Следующая команда `param` определяет матрицу значений параметра:

```
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
;
```

Имеется также следующий синтаксис для задания транспонированной матрицы значений параметров:

```
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
;
```

Это упрощает представление параметрических данных в естественном формате. В частности, благодаря синтаксису с транспонированием можно задавать таблицы, строки которых размещаются в одной строчке файла. Однако каждый столбец матрицы можно разбить на более короткие строки, поскольку в командах данных Pyomo лишние пробелы игнорируются.

Для параметров с тремя и более индексами значения следует задавать в виде последовательности срезов. Каждый срез определяется шаблоном, за которым следует список индексов и значений параметров. Пусть параметр `B` индексирован множеством `A` размерности 4:

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

Следующая команда `param` определяет матрицу значений параметра с несколькими шаблонами:

```
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
;
```

Параметр B имеет четыре значения: B[a,1,a,1]=10, B[b,1,b,1]=20, B[a,2,a,2]=30 и B[b,2,b,2]=40.

10.3.3. Команда include

Команда include позволяет включать в командный файл данных команды из другого файла. Например, следующая команда исполняет команды данных сначала из файла ex1.dat, а затем из ex2.dat:

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

Порядок выполнения команд существен, поскольку команды данных могут переопределять множества и значения параметров. Команда include не нарушает это правило: все команды данных, находящиеся во включенном файле, выполняются раньше команд в текущем файле, следующих за include.

10.3.4. Пространства имен данных

Ключевое слово namespace не является командой данных, а служит для структурирования спецификации команд данных. Точнее, объявление пространства имен позволяет сгруппировать команды данных и снабдить группу меткой. Рассмотрим следующий файл данных:

```
set C := 1 2 3 ;

namespace ns1
{
    set C := 4 5 6 ;
}

namespace ns2
{
    set C := 7 8 9 ;
}
```

В этом файле определены два пространства имен: ns1 и ns2, – инициализирующих множество C. По умолчанию команды данных, находящиеся внутри пространства имен, при конструировании модели игнорируются; если пространство имен не задано, то множество C будет содержать элементы 1, 2, 3. Но если задано пространство имен ns1, то множество C будет содержать элементы 4, 5, 6. О том, как задается пространство имен при выполнении команды pyomo, см. раздел 10.2.2.2.

10.4. Компоненты построения

Python-код в функции конструирования экземпляра `ConcreteModel` может вмешаться в процесс в любом месте. Можно, например, исправить какую-то комбинацию параметров, напечатать значение параметра или возбудить исключение, если некоторая комбинация параметров недопустима. Чтобы обеспечить аналогичную функциональность для `AbstractModel`, Pyomo поддерживает набор компонентов, позволяющих выполнять Python-код в процессе построения.

Компонент `BuildAction` внедряет действия (написанные на Python) в процесс конструирования модели. Компонент `BuildCheck` используется для выполнения определенных пользователем проверок (тоже написанных на Python) в процессе конструирования и прерывания конструирования, если проверка не проходит. Эти компоненты добавляются в модель так же, как все остальные, но их назначение – дать пользователю возможность вставлять код в процесс конструирования модели.

Рассмотрим следующую абстрактную модель, иллюстрирующую применение компонентов `BuildAction` и `BuildCheck` для определения проверки ошибок и диагностической печати в задаче о расположении складов (см. разделы 3.2 и 10.1.3).

```
# buildactions.py: действия построения в задаче о расположении складов
import pyomo.environ as pyo

model = pyo.AbstractModel()

model.N = pyo.Set() # множество складов
model.M = pyo.Set() # множество потребителей
model.d = pyo.Param(model.N,model.M)
model.P = pyo.Param()

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

def checkPN_rule(model):
    return model.P <= len(model.N)
model.checkPN = pyo.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 = pyo.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 = pyo.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 = pyo.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 = pyo.Constraint(rule=num_warehouses_rule)

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

Здесь мы добавили компонент `BuildCheck` с правилом `CheckPN_rule`, которое проверяет, что мы размещаем не больше складов, чем имеется мест для их размещения. Мы также добавили компонент `BuildAction` с правилом `printM_rule` для печати множества местоположений потребителей.

Мы создали dat-файл, в котором параметр *P* больше числа доступных мест для размещения складов (поэтому он не должен пройти проверку `CheckPN_rule`):

```
# buildactions_fails.dat: файл данных в формате Pyomo для задачи о
# размещении складов
# Примечание: параметр P больше числа местоположений складов

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 ;
```

Попытка решения этой задачи командой pyomo

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

дает такой результат:

```
[ 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
```

Компоненты `BuildAction` и `BuildCheck`, как и любые другие, можно индексировать, что позволяет настраивать действия и проверки в зависимости от данных.

Часть III

РАСШИРЕНИЯ МОДЕЛИРОВАНИЯ

Глава 11

Обобщенное дизъюнктивное программирование

В этой главе описано, как записать и решить обобщенные дизъюнктивные программы (ОДП). Модели ОДП предлагают структурный подход к описанию логических связей в моделях оптимизации. Мы покажем, что блоки Pyomo образуют естественную основу для представления дизъюнктов и формирования дизъюнкций, и продемонстрируем, как решать модели ОДП путем использования автоматизированных преобразований задачи.

11.1. Введение

Общая черта многих задач дискретной оптимизации – выбор из двух или более дискретных вариантов. Эти решения налагают дополнительные ограничения на пространство разрешимых задач. Пусть, например, *полунепрерывная переменная* либо равна нулю, либо должна быть больше заданного значения. Это можно выразить алгебраически в виде $x = 0$ или $L \leqslant x \leqslant U$, где $L > 0$, а U может быть бесконечностью. Если U конечно, то это свойство можно записать, определив двоичную переменную y и наложив следующие ограничения:

$$Ly \leqslant x;$$
$$x \leqslant Uy;$$
$$y \in \{0, 1\}.$$

Если U бесконечно, то второе неравенство заменяется так называемым ограничением «M большое»:

$$x \leqslant My,$$

где значение M выбрано настолько большим, чтобы не ограничивать пространство допустимых решений (хотя из него все равно следует конечность U). Легко видеть, что при $y = 1$ из этих неравенств вытекает, что x принадлежит непрерывному диапазону $L \leqslant x \leqslant U$, а при $y = 0$ неравенства сводятся к $0 \leqslant x \leqslant 0$, или $x = 0$.

В этой постановке двоичная переменная y не является частью ограничений, а выражает логическое условие, показывая, какой из двух наборов взаимно исключающих ограничений ($L \leqslant x \leqslant U$ или $x = 0$) следует удовлетворить. Поэтому ее часто называют индикаторной переменной. Вообще, можно не без оснований сказать, что большинство двоичных переменных в математическом программировании на самом деле являются индикаторными переменными, отражающими логику модели.

Этот подход к формулированию решения о выборе можно обобщить на выбор между несколькими группами ограничений. Например, рассмотрим задачу о выделении блока, в которой целью оптимизации является нахождение такого графика включения и выключения генераторов электроэнергии из имеющегося пула, чтобы удовлетворить ожидаемый спрос и минимизировать затраты. В данном случае генератор может находиться в одном из нескольких состояний: включен, выключен, запускается, останавливается. Выбор конкретного состояния налагает дополнительные ограничения на работу генератора. Если генератор «включен», то выходная мощность заключена между минимальным (ненулевым) и максимальным уровнями энергии. Наоборот, если генератор «выключен», то выходная мощность должна быть равна 0. Кроме того, в любой период времени выходная мощность должна находиться в «пределах изменения» выходной мощности в предыдущем периоде времени.

В одной из возможных реализаций этих правил выбора состояния все ограничения включены, но ослаблены, исходя из двоичных переменных состояния (индикаторных переменных), с применением членов типа «M большое»:

$$Power_{g,t} \leqslant MaxPower_g \cdot GenOn_{g,t}; \tag{11.1}$$

$$Power_{g,t} \geqslant MibPower_g \cdot GenOn_{g,t}; \tag{11.2}$$

$$Power_{g,t} \leqslant Power_{g,t-1} + RampUpLimit_g + M_g \cdot (1 - GenOn_{g,t}); \tag{11.3}$$

$$Power_{g,t} \geqslant Power_{g,t-1} - RampDownLimit_g - M_g \cdot (1 - GenOn_{g,t}); \tag{11.4}$$

$$Power_{g,t} \leqslant MaxPower_g \cdot (1 - GenOff_{g,t}); \tag{11.5}$$

$$Power_{g,t-1} \leqslant ShutDownRampLimit_g + MaxPower_g \cdot (1 - GenOff_{g,t}); \tag{11.6}$$

$$Power_{g,t} \leqslant ShutUpRampLimit_g + MaxPower_g \cdot (1 - GenStartUp_{g,t}); \tag{11.7}$$

$$GenOn_{g,t} + GenOff_{g,t} + GenStartUp_{g,t} = 1; \tag{11.8}$$

$$GenOn_{g,t} \leqslant GenOn_{g,t-1} + GenStartUp_{g,t-1}; \tag{11.9}$$

$$GenStartUp_{g,t} \leqslant GenOff_{g,t-1}; \tag{11.10}$$

$$GenOn_{g,t}, GenOff_{g,t}, GenStartUp_{g,t}, GenShutDown_{g,t} \in \{0;1\}; \tag{11.11}$$

$$Power_{g,t} \in (0, MaxPower_g). \tag{11.12}$$

У такого подхода к формулированию решения о выборе есть два существенных недостатка. Во-первых, связи между двоичной переменной выбора и ограничениями, которые она выбирает, довольно запутанны. Во-вторых, использование ослаблений типа «M большое» – лишь один из возможных подходов к формулированию задачи. Зашивая это ослабление в модель, вы, по сути дела, лишаете себя возможности исследовать другие подходы (например, оболочечную релаксацию) без накладного переписывания модели.

Обобщенное дизъюнктивное программирование [54] – это альтернативный подход к представлению задач, имеющих нетривиальную логическую структуру. Он обобщает на нелинейные системы идеи дизъюнктивного программирования [6] для линейных целочисленных задач. Каноническая модель ОДП [38] дополняет целевую функцию, переменные и ограничения типичной задачи смешанно-целочисленного (не)линейного программирования булевыми переменными, дизъюнкциями и логическими ограничениями:

$$\min \sum_{k \in K} c_k + f(x) \tag{11.13}$$

при условиях $r(x) \leqslant 0$ (11.14)

$$\bigvee_{j \in J_k} \begin{bmatrix} Y_{jk} \\ g_{jk}(x) \leqslant 0 \\ c_k = y_{j,k} \end{bmatrix} \quad \forall k \in K \tag{11.15}$$

$$W(Y) = True \tag{11.16}$$

$$x \geqslant 0;\ c_k \geqslant 0,\ Y_{jk} \in \{True, False\} \tag{11.17}$$

В такой постановке логические решения представляются наборами дизъюнкций (формула 11.15) и логических ограничений (формула 11.16). Каждая дизъюнкция содержит ряд членов (*дизъюнктов*), соединенных оператором «OR». Каждый дизъюнкт содержит булеву индикаторную переменную (Y) и набор ограничений, которые должны выполняться, только когда Y равно $True$. Формула (11.16) вводит дополнительные ограничения, определяющие логические связи между индикаторными переменными.

Если переписать модель состояний генератора в виде ОДП, то получим следующую дизъюнкцию:

$$\begin{bmatrix} Y_{g,on} \\ MinPower_g \leqslant Power_{g,t} \leqslant MaxPower_g \\ -RampDownLimit_g \leqslant Power_{g,t} - Power_{g,t-1} \leqslant RampUpLimit_g \end{bmatrix}$$
$$\bigvee \begin{bmatrix} Y_{g,off} \\ Power_{g,t} = 0 \\ Power_{g,t-1} \leqslant ShutDownRampLimit_g \end{bmatrix}$$
$$\bigvee \begin{bmatrix} Y_{g,startup} \\ Power_{g,t} \leqslant StartUpRampLimit_g \end{bmatrix}. \tag{11.18}$$

Этот подход к моделированию устраняет оба обсуждавшихся выше ограничения типичных моделей смешанно-целочисленного (не)линейного программирования: связь между индикаторной переменной и налагаемым ей ограничением теперь явно выражена в структуре модели, и модель больше не привязана ни к какому конкретному ослаблению.

11.2. Моделирование ОДП в Pyomo

Пакет `pyomo.gdp` расширяет базовое окружение моделирования с целью представления моделей ОДП. Этот пакет определяет две новые конструкции: *дизъюнкт* и *дизъюнкция*. Они реализованы как два новых компонента: `Disjunct` и `Disjunction` – и импортируются из пакета ОДП:

```
from pyomo.gdp import Disjunct, Disjunction
```

Дизъюнкт логически является контейнером для индикаторной переменной и соответствующих ограничений. Здесь мы видим иерархический подход к моделированию, возможный благодаря компоненту `Block`: компонент `Disjunct` естественно наследует классу `Block`. Как и блоки, компоненты `Disjunct` можно как угодно индексировать и инициализировать с помощью правил. Кроме того, они могут содержать любой компонент моделирования Pyomo, включая не только `Set`, `Param`, `Var` и `Constraint`, но также `Block`, `Disjunct` и `Disjunction`. Единственное, что класс `Disjunct` добавляет к реализации `Block`, – неявное и автоматическое определение булевой индикаторной переменной дизъюнкта.

В нашем примере состояния генератора три обязательных дизъюнкта объявляются следующим образом:

```
model.NumTimePeriods = pyo.Param()
model.GENERATORS = pyo.Set()
model.TIME = pyo.RangeSet(model.NumTimePeriods)

model.MaxPower = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)
model.MinPower = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)
model.RampUpLimit = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)
model.RampDownLimit = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)
model.StartUpRampLimit = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)
model.ShutDownRampLimit = pyo.Param(model.GENERATORS, within=pyo.NonNegativeReals)

def Power_bound(m,g,t):
    return (0, m.MaxPower[g])
model.Power = pyo.Var(model.GENERATORS, model.TIME, bounds=Power_bound)

def GenOn(b, g, t):
    m = b.model()
    b.power_limit = pyo.Constraint(
        expr=pyo.inequality(m.MinPower[g], m.Power[g,t], m.MaxPower[g]) )
```

```
    if t == m.TIME.first():
        return
    b.ramp_limit = pyo.Constraint(
        expr=pyo.inequality(-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 = pyo.Constraint(
        expr=m.Power[g,t] == 0 )
    if t == m.TIME.first():
        return
    b.ramp_limit = pyo.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 = pyo.Constraint(
        expr=m.Power[g,t] <= m.StartUpRampLimit[g] )
    model.GenStartup = Disjunct(model.GENERATORS, model.TIME, rule=GenStartUp)
```

Отметим, что дизъюнкты могут быть полностью замкнутыми, со своими локальными переменными, параметрами и ограничениями, но могут и ссылаться на компоненты Pyomo, находящиеся вне их непосредственной области видимости.

Компонент `Disjunction` служит для ассоциирования множества дизъюнктов. Дизъюнкция аналогична ограничению в том смысле, что ее можно индексировать и определять с помощью правил. Однако, в отличие от `Constraint`, для которого правило возвращает реляционное выражение, правило для `Disjunction` должно возвращать список объектов `Disjunct`. Последующие преобразования модели преобразуют компонент `Disjunction` и порождают ограничение связывания между дизъюнктами. Хотя в общем виде ОДП связывает дизъюнкты оператором «OR», подавляющее большинство моделей ожидают связи «ровно один» (обобщение оператора «ИСКЛЮЧАЮЩЕЕ ИЛИ»). Это настолько типично, что по умолчанию компонент `Disjunction` порождает связь «ровно один». Однако авторы моделей могут задать исходное поведение оператора «OR», включив именованный параметр `xor=False` в объявление `Disjunction`.

Для нашего примера состояния генератора дизъюнкция – связь «ровно один», в Pyomo ее можно выразить следующим образом:

```
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)
```

11.3. Выражение логических ограничений

Блоки `Disjunct` хранят индикаторную связь между булевой переменной `indicator_var` и ограничениями внутри дизъюнкта, но мы также хотим выразить дополнительные логические ограничения между индикаторными переменными дизъюнктов. Для поддержки этой идеи Pyomo предлагает компонент `LogicalConstraint` для представления логических ограничений в модели. Как и компоненты `Constraint`, компоненты `LogicalConstraint` могут быть одиночными или индексированными и инициализируются с помощью явного именованного аргумента `expr` или функции-правила, переданной в именованном аргументе `rule`. Отличаются они типом принимаемого выражения: если `Constraint` принимает реляционные выражения, т. е. равенства или неравенства, то `LogicalConstraint` принимает логическое выражение.

Логические выражения строятся из булевых переменных и булевых постоянных, соединенных логическими операторами. В табл. 11.1 показаны операторы, поддерживаемые в логических выражениях. Эти операции намеренно не пересекаются с операциями, применяемыми в алгебраических выражениях, чтобы было очевидно семантическое различие между двоичными (алгебраическими) переменными и булевыми (логическими) переменными.

Таблица 11.1. Операции для порождения логических выражений. В примерах X, Y и Z объявлены как булевы переменные (`model.` опущено ради экономии места)

Операция	Функция	Метод	Оператор
Унарные операции			
отрицание	`pyo.lnot(X)`		`~X`
Бинарные операции			
конъюнкция		`X.land(Y)`	
дизъюнкция		`X.lor(Y)`	
исключающая дизъюнкция	`pyo.xor(X, Y)`	`X.xor(Y)`	
импликация	`pyo.implies(X, Y)`	`X.implies(Y)`	
эквиваленция	`pyo.equivalent(X, Y)`	`X.equivalent_to(X,Y)`	
***n*-арные операции**			
конъюнкция	`pyo.land(X, Y, Z)`		
дизъюнкция	`pyo.lor(X, Y, Z)`		
подсчет (не меньше *m*)	`pyo.atleast(m, X, Y, Z)`		
подсчет (не больше *m*)	`pyo.atmost(m, X, Y, Z)`		
подсчет (ровно *m*)	`pyo.exactly(m, X, Y, Z)`		

В примере состояния генератора из предыдущего раздела мы будем использовать логические ограничения для выражения правил перехода состояний, описывающих, как может измениться состояние генератора при переходе от одного временного периода к следующему. Правила смены состояний можно выразить так:

```
def onState(m, g, t):
    if t == m.TIME.first():
        return pyo.LogicalConstraint.Skip
    return m.GenOn[g, t].indicator_var.implies(pyo.lor(
```

```
        m.GenOn[g, t-1].indicator_var,
        m.GenStartup[g, t-1].indicator_var))
model.onState = pyo.LogicalConstraint(
    model.GENERATORS, model.TIME, rule=onState)

def startupState(m, g, t):
    if t == m.TIME.first():
        return pyo.LogicalConstraint.Skip
    return m.GenStartUp[g, t].indicator_var.implies(
        m.GenOff[g, t-1].indicator_var)
model.startupState = pyo.LogicalConstraint(
    model.GENERATORS, model.TIME, rule=startupState)
```

11.4. Решение моделей ОДП

Хотя специализированные решатели, разработанные на Pyomo, умеют разбирать и манипулировать моделями обобщенного дизъюнктивного программирования, стандартные интерфейсы с решателями не могут непосредственно выразить ни дизъюнкции, ни логические ограничения. Однако в Pyomo имеется возможность преобразовать дизъюнктивную модель в эквивалентную модель смешанно-целочисленного (не)линейного программирования путем преобразования логических ограничений в эквивалентную линейную форму и ослабления дизъюнктивных ограничений. Преобразованную (ослабленную) модель затем можно решить с помощью подходящего решателя и стандартных интерфейсов. Пакет ОДП Pyomo включает два автоматизированных ослабления: первое ослабляет дизъюнктивные ограничения путем добавления членов типа «M большое» (восстанавливая исходную структуру модели из раздела 11.1), а второе явно порождает оболочечную релаксацию индивидуальных дизъюнкций.

11.4.1. Преобразование типа «M большое»

Преобразование типа «M большое» выполняет ослабление исходной дизъюнктивной модели по каждому ограничению в отдельности. Это сохраняет размер (число переменных и ограничений) исходной модели ценой порождения, возможно, слабой непрерывной релаксации.

Данное преобразование начинается с переписывания каждого дизъюнкта в виде обычного блока путем прибавления к отдельным ограничениям члена «M большое». Для ограничений типа равенства и двустороннего неравенства (когда заданы верхняя и нижняя границы) преобразование дублирует ограничение в виде двух односторонних ограничений типа неравенства перед ослаблением каждого. Значения параметров M можно задать, поместив `BigM-Suffix` в объект `Disjunct`. Когда преобразуются линейные ограничения ограниченных сверху или снизу переменных, это значение можно автоматически оценить в самом преобразовании.

Наконец, преобразование типа «M большое» переписывает компоненты `Disjunction` в виде алгебраической формы эквивалентного логического ограничения, т. е. либо

$$\exists_k(d_k.indicator\_var \wedge \neg\exists_l(d_l.indicator\_var \wedge k \neq l)) \tag{11.19}$$

для исключающих дизъюнкций (по умолчанию), либо

$$\exists_k(d_k.indicator\_var) \tag{11.20}$$

для неисключающих дизъюнкций. У этих ограничений имеется эквивалентная алгебраическая форма

$$\sum_{k\in K} b_k = 1 \tag{11.21}$$

и

$$\sum_{k\in K} b_k \geqslant 1, \tag{11.22}$$

где b_k – двоичная переменная, ассоциированная с булевой переменной $d_k.indicator\_var$.

Преобразование типа «M большое» называется `gdp.bigm`.

11.4.2. Оболочечное преобразование

Оболочечное преобразование ослабляет исходную дизъюнктивную модель, порождая поднятое представление. Преобразование следует процедуре Баласа (Balas [6]) для линейных дизъюнкций и процедуре Ли и Гроссмана (Lee and Grossmann [38]) (с модификациями, описанными в работе Sawaya and Grossmann [55]) для нелинейных дизъюнкций. В обоих случаях переменные, встречающиеся в каждом дизъюнкте, «деагрегируются» путем определения новых переменных для каждого дизъюнкта и наложения на исходную переменную ограничения: она должна являться суммой деагрегированных переменных. В результате мы представляем дизъюнкцию в виде аффинной комбинации отдельных дизъюнктов. Для дизъюнкций, содержащих только выпуклые ограничения, аффинная комбинация деагрегированных (поднятых) дизъюнктов определяет выпуклую оболочку дизъюнктов. Логическая связь Disjunction преобразуется в алгебраическую форму с применением того же подхода, что в случае «M большого» (формулы 11.19–11.22).

Наложив на деагрегированные переменные ограничение равенства нулю, когда индикаторная переменная дизъюнкта равна `False`, мы сведем решение исходной дизъюнктивной задачи к решению дискретной ослабленной задачи. Это увеличивает общий размер модели (число переменных и ограничений), но дает более близкую непрерывную релаксацию, чем преобразование типа «M большое». Однако для применения оболочечной дизъюнкции все переменные должны быть ограниченными, а объекты `Disjunction` могут выражать только исключающие связи (т. е. `xor=True`).

Оболочечное преобразование называется `gdp.hull`.

11.5. Смешанная задача с полунепрерывными переменными

Следующая модель иллюстрирует простую смешанную задачу с тремя полунепрерывными переменными (x_1, x_2, x_3), представляющими величины, которые смешиваются, чтобы удовлетворить объемометрическому ограничению. В этом простом примере минимизируется количество источников:

```
# scont.py
import pyomo.environ as pyo
from pyomo.gdp import Disjunct, Disjunction

L = [1,2,3]
U = [2,4,6]
index = [0,1,2]

model = pyo.ConcreteModel()
model.x = pyo.Var(index, within=pyo.Reals, bounds=(0,20))
model.x_nonzero = pyo.Var(index, bounds=(0,1))

# Каждая дизъюнкция является полунепрерывной переменной
# x[k] == 0 или L[k] <= x[k] <= U[k]
def d_0_rule(d, k):
    m = d.model()
    d.c = pyo.Constraint(expr=m.x[k] == 0)
model.d_0 = Disjunct(index, rule=d_0_rule)

def d_nonzero_rule(d, k):
    m = d.model()
    d.c = pyo.Constraint(expr=pyo.inequality(L[k], m.x[k], U[k]))
    d.count = pyo.Constraint(expr=m.x_nonzero[k] == 1)
model.d_nonzero = Disjunct(index, rule=d_nonzero_rule)

def D_rule(m, k):
    return [m.d_0[k], m.d_nonzero[k]]
model.D = Disjunction(index, rule=D_rule)

# Минимизировать число ненулевых переменных x
model.o = pyo.Objective(
    expr=sum(model.x_nonzero[k] for k in index))

# Удовлетворить требование, которому отвечают эти переменные
model.c = pyo.Constraint(
    expr=sum(model.x[k] for k in index) >= 7)
```

Существует три способа применить преобразование типа «М большое» или оболочечное преобразование для решения этой модели:

1) с помощью командной утилиты `pyomo`;
2) с помощью скриптового интерфейса;
3) с помощью `BuildAction`.

В командной строке для применения преобразования нужно задать флаг `--transform`:

```
pyomo solve scont.py --transform gdp.bigm --solver=glpk
```

В скриптах для достижения того же эффекта нужно создать преобразование перед применением его к модели:

```
xfrm = pyo.TransformationFactory('gdp.bigm')
xfrm.apply_to(model)

solver = pyo.SolverFactory('glpk')
status = solver.solve(model)
```

Наконец, бывает, что требуется внедрить преобразование в модели, которые создаются и используются в окружениях, отличных от команды `pyomo` или написанных вами скриптов (например, скрипта `runph`). В таком случае для активации преобразования нужно включить в модель `BuildAction`:

```
def transform_gdp(m):
    xfrm = pyo.TransformationFactory('gdp.bigm')
    xfrm.apply_to(m)
model.transform_gdp = pyo.BuildAction(rule=transform_gdp)
```

Глава 12

Дифференциальные алгебраические уравнения

В этой главе описано, как ставить и решать задачи оптимизации с дифференциальными и алгебраическими уравнениями (ДАУ). Пакет `pyomo.dae` позволяет пользователям встраивать подробные динамические модели в каркас оптимизации; он обладает достаточной гибкостью для представления широкого спектра дифференциальных уравнений. Этот пакет включает несколько методов автоматизированного решения задач оптимизации, основанных на одновременной дискретизации.

12.1. Введение

Чтобы лучше разобраться в природных явлениях, ученые и инженеры часто строят динамические, т. е. основанные на дифференциальных уравнениях, модели. Имитационное моделирование до сих пор является областью активных исследований во многих отраслях, потому что их обсчет может быть трудным и вычислительно затратным делом. Но после разработки имитационной модели часто встает следующий вопрос: оптимизировать некоторый аспект динамической системы (например, оценить параметры при известных динамических данных или реализовать управление динамической системой для достижения нужного состояния). Рассмотрим простую задачу оптимального управления из работы [35]:

$$\min x_3(t_f) \tag{12.1}$$

при условиях

$$\dot{x}_1 = x_2 \tag{12.2}$$

$$\dot{x}_2 = -x_2 + u \tag{12.3}$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 \cdot u^2 \tag{12.4}$$

$$x_2 - 8 \cdot (t - 0.5)^2 + 0.5 \leqslant 0 \tag{12.5}$$

$$x_1(0) = 0, x_2(0) = -1, x_3(0) = 0, t_f = 1 \tag{12.6}$$

где целью является минимизация значения x_3 в конечный момент времени путем нахождения оптимальных значений входной переменной u. Эта задача включает три дифференциальных уравнения, выступающих в роли ограничений, и ограничение в виде неравенства на профиль x_2 (называемое также ограничением на траекторию).

Сформулировать задачи оптимизации с динамическими моделями легко, но решить их трудно. Готовые решатели не умеют обращаться с дифференциальными уравнениями непосредственно. Поэтому задачи оптимизации, включающие дифференциальные уравнения в роли ограничений, иначе говоря, *задачи динамической оптимизации*, необходимо предварительно переформулировать. Типичные подходы к решению основаны на методах однократной или многократной пристрелки или полной дискретизации. Но какую бы стратегию решения ни выбрать, реализация метода зачастую переплетена с конкретной решаемой моделью или задачей, поэтому для применения методики к новым задачам динамической оптимизации или для экспериментирования с разными стратегиями решения одной и той же задачи приходится затрачивать много времени.

Пакет `pyomo.dae` решает некоторые из описанных проблем. Он дает пользователям возможность отделить постановку задачи динамической оптимизации от стратегии ее решения. С этой целью вводятся компоненты моделирования для непосредственного представления непрерывных областей и производных. Пакет также включает реализации метода одновременной дискретизации, которые можно автоматически применять к модели Pyomo с дифференциальными уравнениями.

В этой главе дается краткий обзор пакета `pyomo.dae`. Мы отсылаем читателя к работе Nicholson et al. [44], где приведены более подробное описание и сведения о дизайне и степени новизны этого пакета. Пакет до сих пор активно разрабатывается и расширяется. Актуальную информацию о новых возможностях см. в онлайновой документации по Pyomo.

12.2. Компоненты моделирования ДАУ в Pyomo

Пакет `pyomo.dae` определяет два новых компонента для представления моделей ДАУ:

- `ContinuousSet` представляет непрерывные области определения, на которых берется производная;
- `DerivativeVar` представляет производную `Var` на данном `ContinuousSet`.

Для доступа к этим компонентам пакет нужно явно импортировать:

```
import pyomo.environ as pyo
import pyomo.dae as dae
```

Компонент `ContinuousSet` работает похоже на стандартный компонент Pyomo `Set`. Его можно использовать для индексирования других компонентов

Pyomo, в т. ч. `Var`, `Constraint` или `Expression`. Этот компонент можно рассматривать как ограниченный непрерывный диапазон вещественных значений. Он часто используется для представления временных или пространственных областей. Для конструирования `ContinuousSet` необходимо задать верхнюю и нижнюю границы области. В нашем примере задачи оптимального управления непрерывная область *t* объявляется следующим образом:

```
m.tf = pyo.Param(initialize=1)
m.t = dae.ContinuousSet(bounds=(0,m.tf))
```

Для каждой непрерывной области в модели необходимо объявлять отдельный объект `ContinuousSet`. После объявления его можно использовать для индексирования других компонентов Pyomo, а в сочетании с `DerivativeVar` – для объявления производных. Объект `DerivativeVar` необходимо объявлять для каждой производной, встречающейся в динамической модели. Кроме того, брать производную `Var` можно только относительно `ContinuousSet`, включенного как множество индексов этой переменной. Переменные и производные для нашей задачи оптимального управления объявляются следующим образом:

```
m.u = pyo.Var(m.t, initialize=0)
m.x1 = pyo.Var(m.t)
m.x2 = pyo.Var(m.t)
m.x3 = pyo.Var(m.t)

m.dx1 = dae.DerivativeVar(m.x1, wrt=m.t)
m.dx2 = dae.DerivativeVar(m.x2, wrt=m.t)
m.dx3 = dae.DerivativeVar(m.x3)
```

Отметим, что позиционный аргумент, переданный компоненту `DerivativeVar`, – это дифференцируемая переменная `Var`. Множества индексов для `DerivativeVar` наследуются от дифференцируемой `Var`. Если переменная индексируется сразу несколькими `ContinuousSet`, то именованный аргумент `wrt` или `withrespectto` задает требуемую производную. С помощью компонента `DerivativeVar` можно также объявлять производные более высокого порядка. Например, вторая производная задается следующим образом:

```
m.dx1dt2 = dae.DerivativeVar(m.x1, wrt=(m.t, m.t))
```

Дифференциальные уравнения можно формулировать с применением стандартных ограничений Pyomo. Например, дифференциальные уравнения для нашей задачи оптимального управления реализуются вот таким образом:

```
def _x1dot(m, t):
    return m.dx1[t] == m.x2[t]
m.x1dotcon = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2[t] == -m.x2[t] + m.u[t]
m.x2dotcon = pyo.Constraint(m.t, rule=_x2dot)
```

```
def _x3dot(m, t):
    return m.dx3[t] == m.x1[t]**2 + m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dotcon = pyo.Constraint(m.t, rule=_x3dot)
```

Пакет `pyomo.dae` не требует, чтобы дифференциальные уравнения имели конкретную форму или структуру. По умолчанию уравнения должны удовлетворяться на границах непрерывной области. Для некоторых динамических моделей это может быть нежелательно. Для переопределения требований к дифференциальному уравнению на одной или нескольких границах непрерывной области используется метод `deactivate()`:

```
m.x1dotcon[m.t.first()].deactivate()
m.x2dotcon[m.t.first()].deactivate()
m.x3dotcon[m.t.first()].deactivate()
```

И последний важный аспект любой задачи динамической оптимизации – задание начальных или граничных условий. Это можно сделать, зафиксировав `Var` или `DerivativeVar` на одной из границ `ContinuousSet`. Границы `ContinuousSet` можно получить от методов доступа `first()` или `last()`. Например, начальные условия для задачи оптимального управления можно задать так:

```
m.x1[0].fix(0)
m.x2[m.t.first()].fix(-1)
m.x3[m.t.first()].fix(0)
```

Альтернативно можно использовать ограничения для задания более сложных условий, например циклических граничных условий.

Наконец, целевая функция (12.1) и ограничение на траекторию (12.5) в нашей задаче оптимального управления реализуются следующим образом:

```
m.obj = pyo.Objective(expr=m.x3[m.tf])

def _con(m, t):
    return m.x2[t] - 8*(t - 0.5)**2 + 0.5 <= 0
m.con = pyo.Constraint(m.t, rule=_con)
```

12.3. Решения моделей Pyomo с ДАУ

Построив модель Pyomo с дифференциальными уравнениями, перейдем к описанию ее решения. Ни один из решателей, к которым у Pyomo есть интерфейс, в настоящее время не может работать с дифференциальными уравнениями напрямую. Единственная техника решения, включенная в `pyomo.dae`, – метод одновременной дискретизации, называемый также прямой транскрипцией. При этом производятся дискретизация непрерывных областей определения в модели и аппроксимация дифференциальных уравнений алгебраическими в точках дискретизации. В результате получается чисто алгебраическая модель, которую можно решить с помощью стандартного решателя задач нелинейного программирования.

В pyomo.dae включены две схемы дискретизации: конечно-разностная и коллокации. Они различаются алгебраическими уравнениями, используемыми для аппроксимации производных, но синтаксически почти одинаковы. Дискретизация применяется к конкретной непрерывной области и распространяется на все производные и ограничения в этой области. После задания схемы и разрешения (числа точек) дискретизации pyomo.dae автоматически добавляет необходимые точки дискретизации в объект ContinuousSet и включает дополнительные ограничения в модель Pyomo с дискретизированными уравнениями. В итоге модель ДАУ преобразуется в алгебраическую.

> **ПРИМЕЧАНИЕ** В отличие от других преобразований Pyomo, преобразования дискретизации из пакета pyomo.dae в настоящее время нельзя применить из командной утилиты pyomo; в Python-скрипте необходимо создать объект преобразования и применить его к своей модели.

12.3.1. Конечно-разностное преобразование

Конечно-разностные методы аппроксимируют производную в конкретной точке с помощью разностной формулы, это одна из простейших для понимания и реализации схем. Существует много вариаций, различающихся выбором точек, применяемых для аппроксимации производной. Самая распространенная – метод левых конечных разностей, который еще называется неявным, или обратным, методом Эйлера. Для иллюстрации формул дискретизации, полученных этим методом, сначала определим следующую производную и дифференциальное уравнение (ограничение):

$$\left(\frac{dx(t)}{dt}, f\left(x(t), u(t)\right)\right) = 0, \quad t \in [0, T]. \tag{12.7}$$

После применения метода левых разностей к непрерывной области определения t получается такая пара, состоящая из производной и ограничения:

$$\left.\frac{dx}{dt}\right|_{t_{k+1}} = \frac{x_{k+1} - x_k}{h}, \quad k = 0, \ldots, N-1; \tag{12.8}$$

$$g\left(\left.\frac{dx}{dt}\right|_{t_{k+1}}, f(x_{k+1}, u_{k+1})\right) = 0, \quad k = 0, \ldots, N-1, \tag{12.9}$$

где $x_k = x(t_k)$, $t_k = kh$, а h – шаг дискретизации, или размер каждого конечного элемента. Когда конечно-разностное преобразование применяется к модели Pyomo, автоматически генерируются формулы дискретизации типа (12.8), которые добавляются в модель в виде ограничений типа равенства.

В нашем примере оптимального управления применение метода левых конечных разностей реализуется таким кодом:

```
discretizer = pyo.TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, nfe=20, wrt=m.t, scheme='BACKWARD')
```

Именованный аргумент `nfe` («number of finite elements» – количество конечных элементов) определяет количество точек дискретизации. Именованный аргумент `scheme` задает используемый метод конечных разностей. В текущей версии пакета `pyomo.dae` реализованы три конечно-разностные схемы: левая (`BACKWARD`), центральная (`CENTRAL`) и правая (`FORWARD`).

12.3.2. Преобразование коллокации

Второй тип дискретизации, включенный в `pyomo.dae`, – коллокация, или, точнее, ортогональная коллокация над конечными элементами. В этом случае непрерывная область определения сначала разбивается на $N - 1$ отрезков, называемых конечными элементами. В каждом из этих отрезков профили дифференциальных переменных (переменных, производные которых встречаются в модели) аппроксимируются полиномами. Полиномы определены с помощью K точек коллокации, являющихся точками дискретизации внутри каждого конечного элемента. На границах конечных элементов обеспечивается непрерывность дифференциальных переменных. Формально-математическое представление этого подхода применительно к производной и дифференциальному уравнению (12.7) описывается формулами:

$$\left.\frac{dx}{dt}\right|_{t_{ij}} = \frac{1}{h_i}\sum_{j=0}^{K} x_{ij}\frac{d\ell_j(\tau_k)}{d\tau}, \quad k = 1, \ldots, K, \;\; i = 1, \ldots, N-1; \tag{12.10}$$

$$0 = g\left(\left.\frac{dx}{dt}\right|_{t_{ij}}, f(x_{ik}, u_{ik})\right), \quad k = 1, \ldots, K, \;\; i = 1, \ldots, N-1; \tag{12.11}$$

$$x_{i+1,0} = \sum_{j=0}^{K} \ell_j(1)x_{ij}, \quad i = 1, \ldots, N-1, \tag{12.12}$$

где $t_{ij} = t_{i-1} + \tau_j h_i$, $x(t_{ij}) = x_{ij}$. Далее заметим, что решение $x(t)$ интерполируется следующим образом:

$$x(t) = \sum_{j=0}^{K} \ell_j(\tau)x_{ij}, \quad t \in [t_{i-1}, t_i], \quad \tau \in [0,1]; \tag{12.13}$$

$$\ell_j(\tau) = \prod_{k=0,\neq j}^{K} \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)}. \tag{12.14}$$

Методы коллокации дают значительно более точную алгебраическую аппроксимацию, чем конечно-разностные. Однако их гораздо труднее реализовать вручную. Различные варианты методов коллокации отличаются функциональным представлением профиля дифференциальной переменной

над конечным элементом, а также выбором точек коллокации. На момент написания книги преобразования коллокации в пакете pyomo.dae пользуются многочленами Лагранжа для представления профилей дифференциальных переменных. Для выбора точек коллокации доступны два метода: сдвинутые корни Гаусса–Радау (LAGRANGE-RADAU) и сдвинутые корни Гаусса–Лежандра (LAGRANGE-LEGENDRE).

Для применения преобразования коллокации к модели Pyomo нужно написать:

```
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to(m,nfe=7,ncp=6,scheme='LAGRANGE-RADAU')
```

Именованный аргумент nfe определяет количество конечных элементов, а именованный аргумент ncp – количество точек коллокации внутри каждого конечного элемента.

12.4. Дополнительные возможности

В пакет pyomo.dae включено несколько дополнительных возможностей. В этом разделе мы упомянем две из них, которые могут быть интересны пользователям, занимающимся условной оптимизацией задач, описываемых дифференциальными уравнениями в частных производных, или еще более продвинутыми стратегиями оптимального управления.

12.4.1. Применение нескольких дискретизаций

Как уже отмечалось, к каждому объекту ContinuousSet, встречающемуся в модели, может быть применено отдельное преобразование дискретизации. Это значит, что к одной модели Pyomo можно применить разные конечно-разностные схемы, или схемы коллокации, или комбинацию того и другого. Например, модель Pyomo с двумя компонентами ContinuousSet (m.t1 и m.t2) можно было бы дискретизировать с помощью любой из показанных ниже комбинаций схем дискретизации:

```
# Применить несколько конечно-разностных схем
discretizer = pyo.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')

# Применить несколько схем коллокации
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to(m, wrt=m.t1, nfe=4, ncp=6, scheme='LAGRANGE-LEGENDRE')
discretizer.apply_to(m, wrt=m.t2, nfe=10, ncp=3, scheme='LAGRANGE-RADAU')

# Применить комбинацию конечно-разностной схемы и схемы коллокации
discretizer1 = pyo.TransformationFactory('dae.finite_difference')
```

```
discretizer2 = pyo.TransformationFactory('dae.collocation')
discretizer1.apply_to(m, wrt=m.t1, nfe=10, scheme='BACKWARD')
discretizer2.apply_to(m, wrt=m.t2, nfe=5, ncp=3, scheme='LAGRANGE-RADAU')
```

12.4.2. Ограничение формы управляющих входов

Одним из главных принципов проектирования пакета `pyomo.dae` была расширяемость, т. е. возможность включения общих реализаций типичных операций, применяемых к задачам динамической оптимизации. Одна такая операция в области оптимального управления – ограничение формы управляющих входов: как правило, разрешаются только кусочно-постоянные или кусочно-линейные входы. Часто при дискретизации модели методом коллокации над конечными элементами на управляющую переменную налагается ограничение постоянства в каждом конечном элементе. В пакет `pyomo.dae` включена функция, которая делает это после применения коллокационной дискретизации к модели. Она выполняет свою задачу, уменьшая количество свободных точек коллокации для конкретной переменной. Например, чтобы в нашем простом примере задачи оптимального управления управляющий вход *u* был кусочно-постоянным, нужно добавить следующую строку сразу после применения преобразования дискретизации:

```
discretizer.reduce_collocation_points(m, var=m.u, ncp=1, contset=m.t)
```

Именованный аргумент `ncp` задает количество свободных точек коллокации в одном конечном элементе для переменной, определяемой именованным аргументом `var`. Задание `ncp=1` разрешает *u* иметь только одну свободную точку коллокации (или степень свободы), т. е. делает ее постоянной в каждом конечном элементе. Эта функция работает путем добавления ограничений в дискретизированную модель, которые реализуют принудительную интерполяцию дополнительных – нежелательных – точек коллокации по остальным.

12.4.3. Построение графиков

После постановки, дискретизации и решения задачи динамической оптимизации пакет `pyomo.dae` позволяет без труда изобразить получившиеся динамические профили. Поскольку объект `ContinuousSet` содержит значения с плавающей точкой из непрерывной области, пользователь может создать на его основе списки Python для нанесения на график. После решения модели любая переменная, индексированная объектом `ContinuousSet`, будет иметь значение для каждой точки в `ContinuousSet`. Поэтому создание списка Python для значений переменной так же просто, как для `ContinuousSet`.

Показанный ниже Python-скрипт объединяет все воедино. В предположении, что модель Pyomo объявлена в отдельном файле, этот скрипт показывает, как применить дискретизацию и решить модель.

```
import pyomo.environ as pyo
from pyomo.dae import *
from path_constraint import m

# Дискретизировать модель методом ортогональной коллокации
discretizer = pyo.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=pyo.SolverFactory('ipopt')
solver.solve(m, tee=True)
```

Наконец, приведенный ниже код демонстрирует применение функции `plotter` для построения графика с использованием пакета `matplotlib`. Получающийся график также показан ниже.

```
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()
```

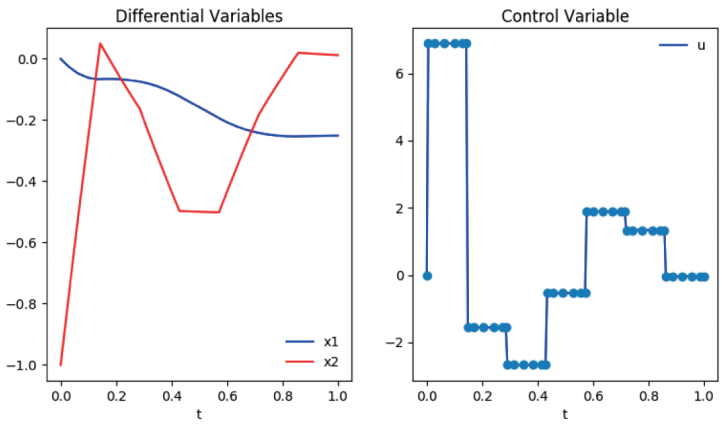

Рис. 12.1 ❖ График, построенный `matplotlib` для примера оптимального управления

Глава 13

Математические программы с ограничениями равновесия

В этой главе описано, как ставить задачи математического программирования с ограничениями равновесия (МПОР, *англ.* MPEC – mathematical programs with equilibrium constraints), которые естественно возникают во многих инженерных и экономических системах. Мы опишем компоненты Pyomo, связанные с условиями дополнительности, преобразования, которые автоматизируют переформулирование моделей МПОР, и метарешатели, в которых эти преобразования используются для поддержки глобальной и локальной оптимизаций моделей МПОР.

13.1. Введение

Математические программы с ограничениями равновесия (МПОР) возникают во многих приложениях инженерных и экономических систем [15, 40, 48]. МПОР – это задача оптимизации, включающая ограничения равновесия в форме условий дополнительности. Ограничения равновесия естественно возникают как решение подзадачи оптимизации (например, для двухуровневых программ), вариационных неравенств и задач дополнительности [29].

Поскольку задачи МПОР часто встречаются на практике, средства выражения условий дополнительности [43] встроены во многие языки алгебраического моделирования (AML), например в AIMMS [1], AMPL [2, 21], GAMS [22], MATLAB [42] и YALMIP [39]. В этой главе мы опишем методы выражения и оптимизации моделей МПОР. Такие модели можно легко выразить с помощью компонентов моделирования Pyomo для условий дополнительности. Кро-

ме того, благодаря своей объектной ориентированности Pyomo естественно поддерживает возможность автоматического переформулирования моделей МПОР в другой форме (например, в виде дизъюнктивной программы). Мы опишем метарешатели, которые преобразуют МПОР в задачи смешанно-целочисленного или нелинейного программирования, которые затем оптимизируются стандартными решателями. Далее мы опишем интерфейсы со специализированными решателями смешанных задач дополнительности, которые решают задачи МПОР, поставленные без целевой функции оптимизации.

13.2. Моделирование условий равновесия

13.2.1. Условия дополнительности

В работе Ferris et al. [16] отмечено существование нескольких основных форм, покрывающих широкий спектр условий дополнительности, встречающихся на практике. Рассмотрим переменную x и функцию $g(x)$. Классическая форма условия дополнительности

$$x \geqslant 0 \perp g(x) \geqslant 0$$

выражает ограничение дополнительности, заключающееся в том, что по крайней мере одно из этих неравенств должно обращаться в равенство. Если переменная x ограничена, т. е. $x \in [l, u]$, то *смешанное условие дополнительности* можно выразить в виде:

$$l \leqslant x \leqslant u \perp g(x).$$

Это означает, что должно быть истинно по меньшей мере одно из следующих утверждений:

$$\begin{aligned} x &= l \quad \text{и } g(x) \geqslant 0, \\ x &= u \quad \text{и } g(x) \leqslant 0, \\ l &< x < u \text{ и } g(x) = 0. \end{aligned}$$

Эти формы можно обобщить, подставив вместо переменной x функцию $f(x)$. Таким образом, *обобщенное смешанное условие дополнительности* можно выразить в форме

$$l \leqslant f(x) \leqslant u \perp g(x),$$

означающей, что должно быть истинно по меньшей мере одно из следующих утверждений:

$$\begin{aligned} f(x) &= l \quad \text{и } g(x) \geqslant 0, \\ f(x) &= u \quad \text{и } g(x) \leqslant 0, \\ l &< f(x) < u \text{ и } g(x) = 0. \end{aligned} \tag{13.1}$$

Для полноты картины отметим, что условие дополнительности

$f(x) \perp g(x)$

является частным случаем, когда $f(x)$ не ограничена.

13.2.2. Выражения дополнительности

Дизайн условий дополнительности в Pyomo опирается на спецификацию выражений ограничений. Выражение ограничения в Pyomo определяет равенство, простое неравенство или пару неравенств, например:

$expr_1 = expr_2$,
$expr_3 \leqslant expr_4$,
$const_1 \leqslant expr_5 \leqslant const_2$,

где $const_i$ – постоянные арифметические выражения, которые могут содержать только фиксированные переменные, а $expr_i$ – арифметические выражения, содержащие нефиксированные переменные.

Условие дополнительности определяется парой выражений ограничения:

$l_1 \leqslant expr_1 \leqslant u_1 \perp l_2 \leqslant expr_2 \leqslant u_2$,

где ровно две из постоянных границ l_1, u_1, l_2 и u_2 конечны. Бесконечные границы на практике опускаются, так что это условие попросту описывает классическое или смешанное условие дополнительности. Кроме того, условие дополнительности можно выразить простым неравенством, например:

$l_1 \leqslant expr_1 \perp expr_3 \leqslant expr_4$.

Это условие дополнительности неявно преобразуется в форму с постоянными границами:

$l_1 \leqslant expr_1 \perp expr_3 - expr_4 \leqslant 0$.

Если используется простое неравенство, то другое выражение ограничения тоже должно быть простым неравенством, чтобы условие дополнительности имело ровно две конечные границы.

13.2.3. Моделирование смешанных условий дополнительности

В пакете `pyomo.mpec` определен компонент `Complementarity`, применяемый для объявления условий дополнительности.

Например, рассмотрим задачу `ralph1` в MacMPEC [41]:

$$\min\ 2x - y$$
$$0 \leqslant y \perp y \leqslant x$$
$$x, y \geqslant 0$$

Следующий скрипт определяет модель Pyomo для `ralph1`:

```
# ralph1.py
import pyomo.environ as pyo
from pyomo.mpec import Complementarity, complements

model = pyo.ConcreteModel()

model.x = pyo.Var( within=pyo.NonNegativeReals )
model.y = pyo.Var( within=pyo.NonNegativeReals )

model.f1 = pyo.Objective( expr=2*model.x - model.y )

model.compl = Complementarity(
                expr=complements(0 <= model.y,
                                 model.y >= model.x) )
```

В первых двух строках импортируются пакеты Pyomo. Пакет `pyomo.environ` инициализирует окружение Pyomo, а пакет `pyomo.mpec` определяет компонент моделирования условий дополнительности. В последующих строках создается модель, объявляются переменные `x` и `y`, целевая функция `f1` и условие дополнительности `compl`.

Условие дополнительности объявляется с помощью компонента `Complementarity`. В простейшем случае конструктор этого Python-класса принимает именованный аргумент `expr`, значением которого является функция `complements`. Эта функция принимает два выражения ограничений и служит для объявления условия дополнительности.

Pyomo также поддерживает индексированные компоненты, когда множество компонентов с индексами инициализируется с помощью правила конструирования. Таким образом, компонент `Complementarity` можно объявить с множеством индексов. Например, рассмотрим следующую модель:

$$\min \sum_{i=1}^{n} i(x_i - 1)^2$$
$$0 \leqslant x_i \perp 0 \leqslant x_{i+1}, i = 1, \ldots, n-1$$

Следующий скрипт определяет реализацию модели с $n = 5$:

```
# ex1a.py
import pyomo.environ as pyo
from pyomo.mpec import Complementarity, complements

n = 5

model = pyo.ConcreteModel()
```

```
model.x = pyo.Var( range(1,n+1) )

model.f = pyo.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_ )
```

Условия дополнительности определяются одним компонентом `Complementarity`, индексированным множеством 1, ..., $n - 1$ и инициализированным правилом конструирования `compl_`. Это правило является функцией, которая принимает экземпляр модели и индекс, а возвращает i-е условие дополнительности.

Объявленное множество индексов может быть надмножеством индексов, определяющих условия дополнительности. Если правило конструирования возвращает значение `Complementarity.Skip`, то соответствующий индекс пропускается. Например:

```
# ex1d.py
import pyomo.environ as pyo
from pyomo.mpec import Complementarity, complements

n = 5

model = pyo.ConcreteModel()

model.x = pyo.Var( range(1,n+1) )

model.f = pyo.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_ )
```

Этот пример можно также выразить с помощью компонента `ComplementarityList`:

```
# ex1b.py
import pyomo.environ as pyo
from pyomo.mpec import ComplementarityList, complements

n = 5

model = pyo.ConcreteModel()

model.x = pyo.Var( range(1,n+1) )

model.f = pyo.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))
```

Этот компонент определяет список условий дополнительности. Индекс элемента списка использовать можно, но вообще-то данный компонент упрощает объявление моделей, в которых значения индексов несущественны. Компонент `ComplementarityList` можно также определить с помощью правила, которое последовательно отдает условия дополнительности:

```
# ex1c.py
import pyomo.environ as pyo
from pyomo.mpec import ComplementarityList, complements

n = 5

model = pyo.ConcreteModel()

model.x = pyo.Var( range(1,n+1) )

model.f = pyo.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_ )
```

Аналогично правило конструирования может быть списковым включением, которое генерирует последовательность условий дополнительности:

```
# ex1e.py
import pyomo.environ as pyo
from pyomo.mpec import ComplementarityList, complements

n = 5

model = pyo.ConcreteModel()

model.x = pyo.Var( range(1,n+1) )

model.f = pyo.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)) )
```

13.3. Преобразования МПОР

Pyomo поддерживает автоматизированное преобразование моделей. Pyomo умеет обходить компоненты модели точно так же, как вложенные блоки модели. Таким образом, компоненты модели легко преобразовать локально, а для поддержки глобальных преобразований можно собрать глобальные данные. Кроме того, компоненты Pyomo и блоки можно активировать и деактивировать, что упрощает преобразования *на месте*, не требующие создания отдельной копии исходной модели.

В пакете `pyomo.mpec` определено несколько легко применяемых преобразований модели. Например, если `model` определяет модель МПОР (как в примерах выше), то ниже показано, как применить преобразование модели:

```
xfrm = pyo.TransformationFactory("mpec.simple_nonlinear")
transformed = xfrm.create_using(model)
```

В этом примере применяется преобразование `mpec.simple_nonlinear`. В следующих разделах описаны преобразования, поддерживаемые в текущей версии `pyomo.mpec`.

13.3.1. Преобразование standard_form

В Pyomo условие дополнительности выражается парой выражений ограничений:

$$l_1 \leqslant expr_1 \leqslant u_1 \perp l_2 \leqslant expr_2 \leqslant u_2,$$

где ровно две из постоянных границ l_1, u_1, l_2 и u_2 конечны. Бесконечные границы обычно опускаются, но при желании их можно выразить с помощью значения `None`. Кроме того, каждое выражение ограничения можно выразить простым неравенством вида

$$expr_3 \leqslant expr_4.$$

Преобразование `mpec.standard_form` переписывает каждое встречающееся в модели условие дополнительности в стандартной форме:

$$l_1 \leqslant expr \leqslant u_1 \perp l_2 \leqslant var \leqslant u_2,$$

где ровно две из постоянных границ l_1, u_1, l_2 и u_2 конечны и либо l_2 равно нулю, либо оба значения l_2 и u_2 конечны.

Отметим, что это преобразование создает новые переменные и ограничения. Например, условие дополнительности

$$1 \leqslant x + y \perp 1 \leqslant 2x - y$$

преобразуется в

$$1 \leqslant x + y \leqslant 0 \leqslant v,$$

где $v \in \mathrm{R}$ и $v = 2x - y - 1$.

Для каждого объекта условия дополнительности новая переменная и ограничения добавляются в объект. Поэтому общая структура модели МПОР при таком преобразовании не изменяется.

13.3.2. Преобразование simple_nonlinear

Преобразование `mpec.simple_nonlinear` начинается с применения преобразования `mpec.standard_form`. Затем создается нелинейное ограничение, определяющее условие дополнительности. Это простое нелинейное преобразование, взятое из работы Ferris et al. [17], которое можно описать следующим образом:

- если l_1 конечно, то ограничение имеет вид

  $(expr - l_1) * v \leqslant \varepsilon$;

- если u_1 конечно, то ограничение имеет вид

  $(u_1 - expr) * v \leqslant \varepsilon$;

- если l_2 и u_2 конечны, то ограничение имеет вид

  $(var - l_2) * expr \leqslant \varepsilon$;
  $(var - u_2) * expr \leqslant \varepsilon$.

В каждом случае условие дополнительности гарантированно удовлетворяется при ε, равном нулю. Например, в первом случае мы знаем, что $0 \leqslant v$ и $0 \leqslant expr - l_1$. Если ε равно 0, то из этого ограничения следует, что либо v равно 0, либо $expr - l_1$ равно 0.

В этом преобразовании используется параметр `mpec_bound`, который определяет значение ε для каждого условия дополнительности. Это позволяет сформулировать ослабленную нелинейную задачу, которую, возможно, будет проще оптимизировать некоторыми решателями задач нелинейного программирования. По умолчанию `mpec_bound` равен 0.

13.3.3. Преобразование simple_disjunction

Преобразование `mpec.simple_disjunction` выражает условие дополнительности в виде дизъюнктивной программы. Заданное условие дополнительности определяется парой выражений ограничения:

$$l_1 \leqslant expr_1 \leqslant u_1 \perp l_2 \leqslant expr_2 \leqslant u_2,$$

где ровно две из постоянных границ l_1, u_1, l_2 и u_2 конечны. Без ограничения общности предположим, что конечна l_1 или u_1.

Это преобразование порождает ограничения, соответствующие условиям, вытекающим из условий дополнительности (см. формулу 13.1). Следует рассмотреть три случая:

- если первое ограничение – равенство, то условие дополнительности тривиальным образом заменяется ограничением равенства;
- если обе границы в первом ограничении конечны, но различны, то дизъюнкция имеет вид:

$$\begin{bmatrix} Y_1 \\ l_1 = expr_1 \\ expr_1 \geqslant 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_2 \\ expr_1 = u_1 \\ expr_2 \leqslant 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_3 \\ l_1 \leqslant expr_1 \leqslant u_1 \\ expr_2 = 0 \end{bmatrix},$$

$$Y_1 \veebar Y_2 \veebar Y_3 = True,$$

$$Y_1, Y_2, Y_3 \in \{True, False\};$$

- в противном случае каждое ограничение является простым неравенством. Условие дополнительности переформулируется в виде

$$0 \leqslant \overline{expr_1} \perp 0 \leqslant \overline{expr_2},$$

а дизъюнкция принимает вид

$$\begin{bmatrix} Y \\ 0 = \overline{expr_1} \\ 0 \leqslant \overline{expr_2} \end{bmatrix} \bigvee \begin{bmatrix} \neg Y \\ 0 \leqslant \overline{expr_1} \\ 0 = \overline{expr_2} \end{bmatrix},$$

$$Y \in \{True, False\}.$$

В этом преобразовании используются компоненты моделирования и преобразования из пакета `pyomo.gdp`. Преобразование выражает каждый дизъюнктивный член явно с помощью компонентов `Disjunct` и выбирает ровно одно логическое условие с помощью компонента `Disjunction`. Преобразование добавляет компоненты `Disjunct` и `Disjunction` в объекты, представляющие условия дополнительности. Затем модифицированные компоненты дополнительности преобразуются в простые компоненты `Block`. В результате все изменения модели локализуются в индивидуальных компонентах `Complementarity`. Последующее преобразование дизъюнктивных выражений в алгебраические ограничения можно выполнять с помощью преобразования типа «М большое» (`gdp.bigm`) или преобразования выпуклой оболочки (`gdp.chull`).

13.3.4. Интерфейс с AMPL-решателями

Такие решатели, как PATH [14], были адаптированы для работы с библиотекой AMPL Solver Library (ASL). В AMPL для взаимодействия с решателями используются `nl`-файлы, которые читаются с помощью библиотеки ASL.

Pyomo тоже умеет создавать nl-файлы, а преобразование mpec.nl приводит компоненты Complementarity к канонической форме, пригодной для этого формата [16].

13.4. Интерфейсы с решателями и метарешатели

Pyomo поддерживает интерфейсы со сторонними решателями, а также метарешатели, которые применяют преобразование и сторонние решатели, возможно, итеративно. Пакет pyomo.mpec включает интерфейс с решателем PATH и несколько метарешателей. Все это описано в данном разделе, и приведены примеры с использованием командной строки.

13.4.1. Нелинейные переформулирования

Преобразование mpec.simple_nonlinear дает общий способ преобразования МПОР в нелинейную программу. Если МПОР включает только непрерывные переменные решения, то результирующую модель можно оптимизировать разными решателями.

Например, командная утилита pyomo позволяет задать нелинейный решатель и применяемое к модели преобразование:

```
pyomo solve --solver=ipopt \
            --transform=mpec.simple_nonlinear ex1a.py
```

Этот пример иллюстрирует использование решателя ipopt, применяющего метод внутренней точки, для решения задачи, порожденной преобразованием mpec.simple_nonlinear. Когда преобразование используется напрямую, как здесь, результаты, возвращенные пользователю, включают переменные решения для преобразованной модели. В Pyomo отсутствуют общие средства для отображения решения назад в пространство исходной модели. В данном примере объект results включает значения переменных x, а также переменных v, появившихся в процессе применения преобразования к стандартной форме, как было показано выше.

Pyomo включает метарешатель mpec_nlp, который применяет нелинейное преобразование, выполняет оптимизацию, а затем возвращает результаты для исходных переменных решения. Так, mpec_nlp реализует ту же логику, что и предыдущий пример вызова pyomo:

```
pyomo solve --solver=mpec_nlp ex1a.py
```

Кроме того, этот метарешатель способен подбирать значение ε в модели, начав с больших значений и постепенно уменьшая их для построения более точной модели.

```
pyomo solve --solver=mpec_nlp \
            --solver-options="epsilon_initial=0.1 epsilon_final=1e-7" \
            ex1a.py
```

Этот подход может быть полезен при работе с нелинейным решателем, который не умеет оптимизировать ограничения типа равенства.

13.4.2. Дизъюнктивные переформулирования

Преобразование `mpec.simple_disjunction` дает общий способ преобразования МПОР в дизъюнктивную программу. Решатель `mpec_minlp` применяет это преобразование для создания нелинейной дизъюнктивной программы, а затем переформулирует дизъюнктивную модель, применяя преобразование типа «M большое» из пакета `pyomo.gdp`. Получающееся преобразование похоже на переформулирование в виде двухуровневых моделей, описанное в работе Fortuny-Amat and McCarl [20]. Если исходная модель была нелинейной, то результирующая модель оказывается задачей смешанно-целочисленного нелинейного программирования (mixed-integer nonlinear program – MINLP). Pyomo включает интерфейсы с решателями, использующими библиотеку AMPL Solver Library (ASL), поэтому `mpec_minlp` может оптимизировать нелинейные МПОР с помощью решателей типа Couenne [10].

Если исходная модель была линейной МПОР, то результирующая модель оказывается задачей смешанно-целочисленного линейного программирования (MIP), которая может быть глобально оптимизирована (см., например, Hu et al. [33], Júdice [36]). Например, команду `pyomo` можно использовать для выполнения решателя `mpec_minlp` с помощью заданного MIP-решателя:

```
pyomo solve --solver=mpec_minlp \
            --solver-options="solver=glpk" ralph1.py
```

Отметим, что Pyomo включает интерфейсы к нескольким MIP-решателям, в том числе CPLEX, Gurobi, CBC и GLPK.

13.4.3. PATH и интерфейс с ASL-решателем

Интерфейс Pyomo с библиотекой AMPL Solver Library (ASL) применяет преобразование `mpec.nl`, записывает `nl`-файл AMPL, выполняет ASL-решатель, а затем загружает решение в исходную модель. Pyomo предлагает специализированный интерфейс PATH с решателем [14], который позволяет просто задать решатель как путь `path`, при условии что исполняемый файл решателя называется `pathamp`.

Команда `pyomo` может выполнить PATH-решатель, просто задав имя `path`. Например, рассмотрим задачу `munson1` из MCPLIB:

```
# munson1.py
import pyomo.environ as pyo
```

```
from pyomo.mpec import Complementarity, complements

model = pyo.ConcreteModel()

model.x1 = pyo.Var()
model.x2 = pyo.Var()
model.x3 = pyo.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))
```

Эту задачу можно решить следующей командой:

```
pyomo solve --solver=path munson1.py
```

13.5. Обсуждение

Pyomo поддерживает условия дополнительности модели примерно так же, как другие AML. Например, в репозитории `pyomo-model-libraries` [52] имеются формулировки Pyomo для многих моделей MacMPEC [41] и MCPLIB [12], которые первоначально были построены для GAMS и AMPL. Однако в настоящее время Pyomo не поддерживает средств моделирования для моделей равновесия, вариационных неравенств и встроенных моделей, которые поддерживаются расширенным каркасом математического программирования GAMS [18].

Приложение A

Краткое руководство по Python

Эта глава содержит краткое пособие по языку программирования Python: переменные, выражения, поток управления, функции и классы. Задача заключается в том, чтобы предоставить информацию, достаточную для понимания конструкций, встречающихся в этой книге. Полное введение в Python следует искать в других источниках, в т. ч. перечисленных в конце главы.

A.1. Обзор

Python – мощный и простой для изучения язык программирования. Python – интерпретируемый язык, поэтому для разработки и тестирования программ на нем не нужны этапы компиляции и компоновки, необходимые для таких традиционных языков, как FORTRAN и C. Кроме того, в состав Python входит интерпретатор команд, с которым можно работать интерактивно. Это позволяет пользователю работать непосредственно со структурами данных Python – вещь, поистине бесценная для изучения возможностей структур данных и диагностики программных ошибок.

Python может похвастаться элегантным синтаксисом, позволяющим писать компактные программы, которые легко читать. Программы, написанные на Python, как правило, гораздо короче эквивалентных программ на языках типа C, C++ или Java по следующим причинам:

- Python поддерживает многие высокоуровневые типы данных, что упрощает выражение сложных операций;
- в Python для группировки предложений применяются отступы, это диктует разумный стиль кодирования;
- в Python применяется динамическая типизация данных, поэтому объявлять типы переменных и аргументов необязательно.

Python – в высшей степени структурированный язык программирования, пригодный для написания больших программ. Поэтому Python обладает го-

раздо большей мощностью, чем многие скриптовые языки (в частности, языки оболочки и пакетные файлы Windows). Python также включает современные средства объектно ориентированного программирования и богатый набор стандартных библиотек, что позволяет быстро создавать сложные программы.

Цель настоящего приложения заключается в том, чтобы предоставить справочную информацию, достаточную для понимания конструкций, встречающихся в этой книге. Полное введение в Python имеется в других источниках, в т. ч. перечисленных в конце главы.

A.2. Установка и выполнение Python

Написанный на Python код исполняется интерпретатором. После запуска интерпретатор печатает приглашение и ждет ввода команды пользователем. Например, можно запустить интерпретатор Python из оболочки, а затем напечатать строку «Hello World»:

```
% python
Python 3.7.4 (default, Aug 13 2020, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more n
information.
>>> print ("Hello World")
Hello World
>>>
```

В Windows команду `python` можно запустить из окна команд (или еще какой-то оболочки), а в *nix (включая Mac) – из оболочки bash или csh (или из терминала). Интерактивная оболочка Python похожа на эти оболочки ОС: когда пользователь вводит корректное предложение Python, оно сразу же выполняется, и печатается результат.

Интерактивная оболочка полезна для опроса состояния сложных типов данных. Чаще всего выполняются однострочные предложения, как, например, функция `print` в примере выше. Но можно выполнять и многострочные предложения. Чтобы показать, что предложение нуждается в продолжении, Python выводит приглашение «...». Например, в условном предложении нужен блок предложений, которые вводятся в строках продолжения:

```
>>> x = True
>>> if x:
...     print("x is True")
... else:
...     print("x is False")
...
x is True
```

ПРИМЕЧАНИЕ В многострочных предложениях, которые вводятся в интерактивной оболочке, нужно следить за правильностью отступов.

ПРИМЕЧАНИЕ `True` – предопределенный литерал в Python, поэтому предложение `x = True` присваивает это значение переменной `x` точно так же, как предложение `x = 6` присвоило бы `x` значение `6`.

Интерпретатор Python можно также использовать для выполнения предложений, находящихся в файле, что открывает возможность автоматического выполнения сложных Python-программ. Исходные файлы Python – это текстовые файлы, имена которых, по соглашению, заканчиваются суффиксом `.py`. Например, рассмотрим файл `example.py`:

```
# Это строка комментария, Python ее игнорирует
print("Здравствуй, мир")
```

Этот код можно выполнить несколькими способами. Пожалуй, самый распространенный – вызвать интерпретатор Python из оболочки:

```
% python example.py
Hello World
%
```

В Windows Python-программы можно выполнить двойным щелчком по файлу с расширением `.py`; при этом открывается окно консоли, в котором выполняется интерпретатор Python. Окно консоли закрывается сразу после завершения интерпретатора, но этот пример легко подправить, так чтобы он ожидал ввода от пользователя, прежде чем завершиться:

```
# Модифицированная программа example.py
print("Hello World")

import sys
sys.stdin.readline()
```

A.3. Формат строки в Python

В Python нет символов начала и окончания блока. Вместо этого двоеточие обозначает конец предложения, после которого начинается блок, а предложения внутри блока должны начинаться отступом. Рассмотрим, к примеру, файл, содержащий следующие Python-команды:

```
# Этот комментарий - первая строка файла LineExample.py.
# Все символы в строке после знака # игнорируются интерпретатором.
print("Здравствуй, мир, сколько в тебе людей?")

population = 400

if population > 300:
    print("Ух ты!")
    print("Да это же целая куча народу")
else:
    print("Это меньше, чем я ожидал...")
```

Будучи передана Python, эта программа приведет к выводу какого-то текста.

Поскольку отступы синтаксически значимы, их нужно строго соблюдать. Следующая программа выдает сообщение об ошибке из-за несогласованности отступов внутри блока if:

```
# Этот комментарий – первая строка скрипта BadIndent.py,
# который приведет к выдаче сообщения об ошибке
# из-за неправильного отступа.
print("Здравствуй, мир, что происходит?")

ans = "Много чего"

if ans == "Много чего":
    print("Очень интересно")
        print("Но немного рискованно.")
else:
    print("Расслабься!")
```

Чаще всего каждая строка Python-скрипта содержит одно предложение. Длинные предложения, содержащие длинные имена переменных, могут порождать очень длинные строки. Синтаксически это допустимо, но иногда удобно разбить предложение на две или более строк. Знак обратной косой черты (\) говорит Python, что текст логически является частью предложения, которое будет продолжено на следующей строке. С точки зрения пользователей Pyomo, самый важный случай, когда обратная косая черта не нужна, – список аргументов функции. Аргументы функции разделяются запятыми, а после запятой аргументы можно продолжать на следующей строке, не употребляя обратной косой черты.

Обратно, иногда несколько предложений Python можно написать в одной строке. Но мы против такой практики, т. к. она затрудняет сопровождение кода.

A.4. Переменные и типы данных

Переменные Python не нуждаются в явном объявлении. Если в предложении присваивается значение неопределенному символу, то неявно объявляется переменная. Тип переменной определяется тем, какие данные она содержит. Предложение

```
population=200
```

создает переменную `population` целого типа, потому что 200 – целое число. Python чувствителен к регистру, поэтому предложение

```
Population = "Больше, чем вчера."
```

создает переменную, отличную от `population`. После присваивания

```
population = Population
```

переменная `population` будет иметь то же значение, что `Population`, а следовательно, тот же тип.

A.5. Структуры данных

В этом разделе кратко описаны структуры данных Python, полезные при написании приложений Pyomo. Ко многим структурам данных Python и Pyomo можно обращаться путем индексирования их элементов. В Pyomo индексы и диапазоны обычно начинаются с единицы, а в Python – с нуля.

A.5.1. Строки

Строковые литералы можно заключать в одиночные или двойные кавычки, что позволяет легко включать их в строку. Python поддерживает много функций и операций со строками. Например, оператор сложения (+) конкатенирует строки. Для приведения другого типа к строке служит функция `str`. Предложение

```
NameAge = "SPAM was introduced in " + str(1937)
```

присваивает строковое значение переменной Python с именем `NameAge`.

A.5.2. Списки

Списки Python приблизительно соответствуют массивам в других языках программирования. Можно обращаться к отдельным элемента списка, к списку как единому целому или к части списка. В качестве символа вырезания используется двоеточие (:), а отрицательный индекс означает отсчет с конца. Ниже иллюстрируются операции со списками:

```
>>> 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]
```

Оператор сложения конкатенирует списки, а оператор умножения на целое число повторяет список заданное число раз.

> **ПРИМЕЧАНИЕ** В Python списки могут содержать элементы разных типов, например числа с плавающей точкой и целые, как в примере выше.

Существует много функций для работы со списками. Самой употребительной, пожалуй, является функция `append`, которая добавляет элемент в конец списка:

```
>>> a = []
>>> a.append(16)
>>> a.append(22.4)
>>> a
[16, 22.4]
```

A.5.3. Кортежи

Кортежи похожи на списки, но предназначены для описания многомерных объектов. Например, имеет смысл создать список кортежей. Кортеж отличатся от списка тем, что для инициализации используются круглые, а не квадратные скобки. Кроме того, элементы списка можно изменять путем присваивания, а элементы кортежа нельзя (т. е. кортежи неизменяемы, тогда как списки изменяемы). Хотя для инициализации кортежей применяются круглые скобки, для обращения к отдельным элементам используются квадратные (так же, как для списков, и это значит, что для обращения к элементам списка кортежей можно использовать код, который выглядит так же, как код доступа к массиву).

Предположим, что кортеж представляет местоположение точки в трехмерном пространстве. Началу координат соответствует кортеж `(0,0,0)`. Рассмотрим следующий сеанс работы с Python:

```
>>> orig = (0,0,0)
>>> pt = (-1.1, 9, 6)
>>> pt[1]
9
>>> pt = orig
>>> pt
(0, 0, 0)
```

Предложение

```
pt[1] = 4
```

завершится ошибкой, потому что элементы кортежа нельзя перезаписывать. Разумеется, весь кортеж перезаписать можно, поскольку такое присваивание затрагивает только переменную, ссылающуюся на кортеж.

A.5.4. Множества

Множества в Python очень похожи на компоненты Set в Pyomo. Множества не могут содержать дубликатов и являются неупорядоченными. Они объявляются функцией set, которая принимает список (возможно, пустой) в качестве аргумента. Для множества определены различные методы, в т. ч. add (добавить один элемент), update (принимающий несколько новых элементов) и discard (удалить существующие элементы). В следующем сеансе Python иллюстрирует функциональность объектов set:

```
>>> A = set([1, 3])
>>> B = set([2, 4, 6])
>>> A.add(7)
>>> C = A | B
>>> print(C)
set([1, 2, 3, 4, 6, 7])
```

> **ПРИМЕЧАНИЕ** set в нижнем регистре относится к встроенному в Python объекту. Set, начинающееся с заглавной буквы, относится к компоненту Pyomo.

A.5.5. Словари

Словари Python чем-то похожи на списки, однако они не упорядочены и могут индексироваться любым неизменяемым типом (например, строками, числами, кортежами, состоящими из строк или чисел, и более сложными объектами). Индексы называются ключами, и в любом словаре ключи должны быть уникальны. Словари создаются с помощью фигурных скобок и могут инициализироваться списком пар ключ-значение, разделенных запятыми. Элементы можно добавлять в словарь путем присваивания значения ключу. Значениями могут быть произвольные объекты (даже другие словари), но нас будут интересовать простые словари. Ниже приведен пример:

```
>>> 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. Условные предложения

Python поддерживает условное выполнение кода с помощью такой конструкции:

```
if CONDITIONAL1:
    предложения
elif CONDITIONAL2:
    предложения
else:
    предложения
```

Части `elif` и `else` необязательны, количество частей `elif` произвольно. Любой блок условного кода может содержать произвольное число предложений. Условные предложения можно заменить логическим выражением, обращением к булевой функции или булевой переменной (ее можно даже назвать `CONDITIONAL1`). В таких выражениях иногда используются булевы литералы `True` и `False`. В следующей программе иллюстрируются некоторые из этих идей:

```
x = 6
y = False

if x == 5:
    print("x оказалось равно 5")
    print("так уж вышло")
elif y:
    print("x не равно 5, но по крайней мере y равно True")
else:
    print("Эта программа мало что нам говорит.")
```

A.7. Итерации и циклы

Как и в большинстве современных языков программирования, в Python есть циклы `for` и `while`, а также предложения `continue` и `break`. Если в цикле `for` или `while` имеется часть `else`, то по завершении цикла выполняется содержащийся в ней блок кода. Предложение `continue` завершает выполнение текущего блока кода и передает управление на начало цикла. Предложение `break` приводит к выходу из цикла.

Эти конструкции иллюстрируются в следующем примере:

```
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("Конец цикла. Клиз в списке!")
        break;
    print(i+" "+str(D[i]))
else:
    print("Клиза нет в списке.")
```

Здесь мы в цикле `for` обходим все ключи словаря. Ключевое слово `in` особенно полезно, поскольку упрощает обход итерируемых типов, таких как списки и словари. Обратите внимание, что порядок ключей не определен; чтобы отсортировать их, применяется функция `sorted()`.

Эта программа печатает список ключей и соответствующих им значений, кроме ключа «Alice», а затем печатает сообщение «Клиза нет в списке». Если бы среди ключей была строка «John», то на ней цикл был бы прерван, и в таком случае часть `else` была бы пропущена, потому что `break` приводит к выходу из всей конструкции цикла, включая `else`.

A.8. Генераторы и списковые включения

Генераторы и списковые включения тесно связаны. Списковые включения часто используются в моделях Pyomo, потому что создают список «на лету», обходясь лаконичным синтаксисом. Генераторы позволяют обходить список, не создавая его.

Прежде чем переходить к обсуждению списковых включений и генераторов, полезно упомянуть о функции Python `range`. Она принимает от одного до трех целых аргументов: `start`, `beyond` и `step` – и возвращает список, первым элементом которого является `start`, а каждый следующий равен предыдущему плюс `step`; список заканчивается по достижении `beyond`. По умолчанию `start` равно 0, а `step` равно 1. Если задан только один аргумент, считается, что это `beyond`, если два, то это `start` и `beyond`.

Списковое включение – это выражение в квадратных скобках, описывающее создание списка. В следующем примере иллюстрируется использование спискового включения для порождения квадратов первых пяти натуральных чисел:

```
>>> a = [i*i for i in range(1,6)]
>>> a
[1, 4, 9, 16, 25]
```

Генераторы используются в выражениях итерирования и напоминают списковое включение, только никакого списка на самом деле не создается. В некоторых случаях экономия памяти и времени, проистекающая из применения генератора вместо спискового включения, может быть значительной.

A.9. Функции

Функции Python могут принимать объекты в качестве аргументов и возвращать объекты. Поскольку в Python уже встроены типы кортежа, списка и словаря, функция может возвращать несколько значений естественным образом. Автор функции может предоставить значения по умолчанию для

незаданных аргументов, поэтому часто можно встретить функции, вызываемые с разным числом аргументов. В Python функция является также и объектом, поэтому функции можно передавать в качестве аргументов другим функциям.

Аргументы передаются функции по ссылке, но в Python многие типы неизменяемы, поэтому начинающему программисту нелегко определить, аргументы каких типов могут быть изменены функцией. У разработчиков на Python не принято писать функции, которые изменяют значения своих аргументов. Однако если функция является членом класса, то она часто изменяет данные, принадлежащие объекту, от имени которого вызвана.

Пользовательские функции объявляются в предложении `def`. Предложение `return` приводит к завершению функции и возврату указанных значений. Функция не обязана что-то возвращать; конец блока функции с отступом от `def` также сигнализирует о конце функции. Сказанное проиллюстрировано в примере ниже:

```
def Apply(f, a):
    r = []
    for i in range(len(a)):
        r.append(f(a[i]))
    return r

def SqifOdd(x):
    # если x нечетно, то 2*int(x/2) не равно x
    # поскольку операция x/2 - деление нацело
    if 2*int(x/2) == x:
        return x
    else:
        return x*x

ShortList = range(4)
B = Apply(SqifOdd, ShortList)
print(B)
```

Эта программа печатает `[0, 1, 2, 9]`. Функция `Apply` ожидает получить функцию и список; она возвращает новый список, полученный применением функции к каждому элементу переданного списка. Функция `SqifOdd` возвращает свой аргумент (`x`), если `2*int(x/2)` не равно `x`. Если число `x` нечетное, то `int(x/2)` отбрасывает дробную часть `x/2`, поэтому результат не равен `x`.

Более трудная тема – написание и использование функциональных оберток. В Python есть несколько способов сделать это, но мы кратко остановимся только на *декораторах*, потому что иногда они встречаются в моделях и скриптах Pyomo. Определение декоратора может быть сложным, но использовать его просто: над объявлением декорируемой функции помещается знак `@`, за которым следует имя декоратора.

Ниже приведен пример определения и использования нелепого декоратора, который заменяет все буквы ‘c’ буквами ‘b’ в возвращаемом функцией значении.

```
# Пример нелепого декоратора, заменяющего 'c' на 'b'
# в возвращаемом функцией значении.
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()) # печатается: Flying Birbus
```

В определении декоратора `ctob_decorate` функция-обертка `func_wrapper` пользуется стандартным механизмом Python, позволяющим задавать произвольные аргументы. Обертка предполагает, что функция, переданная в формальном аргументе `func`, возвращает строку (но не проверяет этого). Раз определенную функцию-обертку можно использовать для декорирования любого числа функций. В примере выше декорирована функция `Last_Words`, в результате чего изменяется возвращенное ей значение.

A.10. Объекты и классы

Классы определяют объекты. Иначе говоря, объекты являются экземплярами классов. Объекты могут содержать сколько угодно членов: данных и функций. В этом контексте функции часто называются методами. Попутно отметим, что в Python данные и функции технически являются объектами, поэтому будет правильно сказать, что членами объектов являются объекты.

Пользовательские классы объявляются в предложении `class`, и все, что содержится во внутреннем блоке, является частью определения класса. В качестве очень простого примера класса приведем контейнер, печатающий хранящееся в нем значение:

```
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() # печатается: The Int Locker has 3
```

```
a.set_value(5)
a.pprint() # печатается: The Int Locker has 5
```

В классе `IntLocker` имеется член данных `sint` и две функции-члена. При вызове функции-члена Python автоматически передает объект в первом аргументе. Поэтому имеет смысл назвать первый аргумент функции-члена `self`, т. к. с его помощью класс ссылается сам на себя. Метод `__init__` – это специальная функция-член, которая автоматически вызывается при создании объекта; ее присутствие необязательно.

К объекту Python легко присоединить новые атрибуты. Например, к объекту с именем `a` можно присоединить атрибут `name` со значением «spam»:

```
a.name = "spam"
```

Можно также спросить у объекта, какие атрибуты у него уже есть.

A.11. Присваивание, copy и deepcopy

A.11.1. Ссылки

Предложение присваивания в Python ассоциирует ссылку с именем переменной, указанным слева от знака равенства. Если в правой части находится литерал, то Python создает такой объект в памяти и присваивает ссылку на него. Например, после выполнения предложений

```
>>> x = [1,2,3]
>>> y = x
>>> x[0] = 3
>>> x.append(6)
>>> print(y)
```

будет напечатано

```
[3,2,3,6]
```

Но есть тонкий момент: `y` ссылается на тот же объект, что и `x`, но не на сам `x`. Продолжим пример; после выполнения предложений

```
>>> x = [1,2,3]
>>> y = x
>>> x[0] = 3
>>> x = "Norwegian Blue"
>>> print(x, y)
```

будет напечатано

```
Norwegian Blue [3,2,3]
```

Некоторые типы Python неизменяемы, т. е. их значение в памяти нельзя изменить. К таким типам относятся int, float, decimal, bool, string и tuple. Если не считать tuple, то удивляться тут нечему. Рассмотрим следующий код:

```
>>> x = (1,2,3)
>>> y = x
>>> x[0] = 3
```

В результате будет выдана ошибка, потому что кортежи (в отличие от списков) нельзя изменять после создания.

A.11.2. Копирование

Иногда присваивание ссылки – не то, что требуется. Для таких ситуаций предназначен модуль Python `copy`, который позволяет перенести данные из одной переменной в другую. Он поддерживает поверхностное копирование методом `copy` и глубокое копирование методом `deepcopy`. Разница проявляется только для вложенных структур (например, словарей, содержащих списки). Метод `deepcopy` пытается создать новую копию всего, а метод `copy` создает новую копию только верхнего уровня, а для всего остального пытается создать ссылки. Код

```
>>> import copy
>>> x = [1,2,3]
>>> y = copy.deepcopy(x)
>>> x[0] = 3
>>> x.append(6)
>>> print(x,y)
```

напечатает

```
[3,2,3,6] [1,2,3]
```

В этом конкретном примере `copy.copy` и `copy.deepcopy` ведут себя одинаково.

A.12. Модули

Модулем называется любой файл, содержащий предложения на Python. Например, любой файл, содержащий «программу» на Python, в которой определены классы или функции, является модулем. Определения, находящиеся в одном модуле, могут быть сделаны доступными в другом модуле (или программе) с помощью команды `import`, которая может перечислить имена, подлежащие импорту, или сказать, что нужно импортировать все имена, воспользовавшись звездочкой.

Вместе с Python обычно устанавливается много стандартных модулей, например `types`. Команда `from types import *` приводит к импорту всех имен из модуля `types`.

Несколько файлов модулей, находящихся в одном каталоге, можно организовать в виде *пакета*, а пакеты могут содержать модули и подпакеты. При импорте из пакета можно указать имя пакета (т. е. имя каталога), точку и имя модуля. Например, команда

```
import pyomo.environ as pyo
```

импортирует модуль `environ` из пакета `pyomo` и делает имена из этого модуля доступными по локальному имени `pyo`. По аналогии с методом `__init__` в классе Python, в каталог можно включить файл `__init__.py`, тогда содержащийся в нем код будет исполняться на этапе импорта модуля.

A.13. Ресурсы, посвященные Python

- Домашняя страница Python, http://www.python.org.
- Сайт Stack Overflow, https://stackoverflow.com.

Литература

[1] AIMMS. Home page. http://www.aimms.com, 2017.
[2] AMPL. Home page. http://www.ampl.com, 2017.
[3] P. Anbalagan and M. 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 S. 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] B. Bequette. Process control: modeling, design, and simulation. Prentice Hall, 2003.
[8] BSD. Open Source Initiative (OSI) – the BSD license. http://www.opensource.org/licenses/bsd-license.php, 2009.
[9] COIN-OR. Home page. http://www.coin-or.org, 2017.
[10] COUENNE. Home page. http://www.coin-or.org/Couenne, 2017.
[11] CPLEX. http://www.cplex.com, July 2010.
[12] S. P. Dirkse and M. C. Ferris. MCPLIB: A collection of nonlinear mixed-complementarity problems. Optimization Methods and Software, 5 (4): 319–345, 1995.
[13] I. Dunning, J. Huchette, and M. Lubin. Jump: A modeling language for mathematical optimization. SIAM Review, 59 (2): 295–320, 2017. doi: 10.1137/15M1020575.
[14] M. C. Ferris and T. S. Munson. Complementarity problems in GAMS and the path solver. Journal of Economic Dynamics and Control, 24 (2): 165–188, 2000.
[15] M. C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. SIAM Review, 39 (4): 669–713, 1997.
[16] M. C. Ferris, R. Fourer, and D. M. Gay. Expressing complementarity problems in an algebraic modeling language and communicating them to solvers. SIAM J. Optimization, 9 (4): 991–1009, 1999.
[17] M. C. Ferris, S. 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.
[18] M. C. Ferris, S. P. Dirkse, J.-H. Jagla, and A. Meeraus. An extended mathematical programming framework. Computers and Chemical Engineering, 33 (12): 19731982, 2009.

[19] FLOPC++. Home page. https://projects.coin-or.org/FlopC++, 2017.
[20] J. Fortuny-Amat and B. McCarl. A representation and economic interpretation of a two-level programming problem. The Journal of the Operations Research Society, 32 (9): 783–792, 1981.
[21] 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.
[22] GAMS. Home page. http://www.gams.com, 2008.
[23] D. Gay. Hooking your solver to ampl. Numerical Analysis Manuscript, pages 93–10, 1993.
[24] D. Gay. Writing. nl files, 2005.
[25] GLPK. GLPK: GNU linear programming toolkit. http://www.gnu.org/software/glpk, 2009.
[26] H. J. Greenberg. A bibliography for the development of an intelligent mathematical programming system. ITORMS, 1 (1), 1996.
[27] J. L. Gross and J. Yellen. Graph Theory and Its Applications, 2nd Edition. Chapman & Hall/CRC, 2006.
[28] GUROBI. Gurobi optimization. http://www.gurobi.com, July 2010.
[29] 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.
[30] W. E. Hart, J.-P. Watson, and D. L. Woodruff. Pyomo: Modeling and solving mathematical programs in Python. Mathematical Programming Computation, 3: 219–260, 2011.
[31] K. K. Haugen, A. Lkketangen, and D. L. Woodruff. Progressive hedging as a meta-heuristic applied to stochastic lot-sizing. European Journal of Operational Research, 132 (1): 116 – 122, 2001.
[32] 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.
[33] J. Hu, J. E. Mitchell, J.-S. Pang, K. P. Bennett, and G. Kunapuli. On the global solution of linear programs with linear complementarity constraints. SIAM J. Optimization, 19 (1): 445–471, 2008.
[34] Ipopt. Home page. https://projects.coin-or.org/Ipopt, 2017.
[35] 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.
[36] J. J. Júdice. Algorithms for linear programming with linear complementarity constraints. TOP, 20 (1): 4–25, 2011.
[37] J. Kallrath. Modeling Languages in Mathematical Optimization. Kluwer Academic Publishers, 2004.
[38] S. Lee and I. E. Grossmann. New algorithms for nonlinear generalized disjunctive programming. Comp.Chem.Engng, 24 (9–10): 2125–2141, 2000.
[39] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In 2004 IEEE Intl Symp on Computer Aided Control Systems Design, 2004.
[40] Z.-Q. Lou, J.-S. Pang, and D. Ralph. Mathematical Programming with Equilibrium Constraints. Cambridge University Press, Cambridge, UK, 1996.

[41] MacMPEC. MacMPEC: AMPL collection of MPECs. https://wiki.mcs.anl.gov/leyffer/index.php/MacMPEC, 2000.
[42] MATLAB. User's Guide. The MathWorks, Inc., 1992.
[43] T. S. Munson. Algorithms and Environments for Complementarity. PhD thesis, University of Wisconsin, Madison, 2000.
[44] B. Nicholson, J. D. Siirola, J.-P. Watson, V. M. Zavala, and L. T. Biegler. pyomo.dae: A modeling and automatic discretization framework for optimization with differential and algebraic equations. Mathematical Programming Computation, 2018.
[45] J. Nocedal and S. Wright. Numerical optimization, series in operations research and financial engineering, 2006.
[46] OpenOpt. Home page. https://pypi.python.org/pypi/openopt, 2017.
[47] OptimJ. Wikipedia page. https://en.wikipedia.org/wiki/OptimJ, 2017.
[48] J. Outrata, M. Kocvara, and J. Zowe. Nonsmooth Approach to Optimization Problems with Equilibrium Constraints. Kluwer Academic Publishers, Dordrecht, 1998.
[49] PuLP. A Python linear programming modeler. https://pythonhosted.org/PuLP/, 2017.
[50] PyGlpk. PyGlpk: A Python module which encapsulates GLPK. http://www.tfinley.net/software/pyglpk, 2011.
[51] Pyipopt. Home page. https://github.com/xuy/pyipopt, 2017.
[52] pyomo-model-libraries. Models and examples for Pyomo. https://github.com/Pyomo/pyomo-model-libraries, 2015.
[53] Pyomo Software. Github site. https://github.com/Pyomo, 2017.
[54] R. Raman and I. E. Grossmann. Modelling and computational techniques for logic based integer programming. Comp.Chem.Engng, 18 (7): 563–578, 1994.
[55] N. Sawaya and I. E. Grossmann. Computational implementation of non-linear convex hull reformulation. Comp.Chem.Engng, 31 (7): 856–866, 2007.
[56] H. Schichl. Models and the history of modeling. In J. Kallrath, editor, Modeling Languages in Mathematical Optimization, Dordrecht, Netherlands, 2004. Kluwer Academic Publishers.
[57] TOMLAB. TOMLAB optimization environment. http://www.tomopt.com/tomlab, 2008.
[58] H. P. Williams. Model Building in Mathematical Programming. John Wiley & Sons, Ltd., fifth edition, 2013.
[59] Y. Zhou and J. Davis. Open source software reliability model: An empirical approach. ACM SIGSOFT Software Engineering Notes, 30: 1–6, 2005.

Предметный указатель

Н

О

П

Р

С

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Майкл Л. Бинум, Габриэль А. Хакебейл, Уильям Э. Харт,
Карл Д. Лэрд, Бетани Л. Николсон, Джон Д. Сиирола,
Жан-Поль Уотсон, Дэвид Л. Вудраф

Pyomo. Моделирование оптимизации на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 18,69. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Книга содержит полное руководство по Pyomo (Python Optimization Modeling Objects) — пакету с открытым исходным кодом, предназначенному для формулирования и решения крупномасштабных задач оптимизации. Pyomo включает классы Python для определения разреженных множеств, параметров и переменных, с помощью которых записываются алгебраические выражения, определяющие целевые функции и ограничения. Кроме того, программу можно использовать как из командной строки, так и из интерактивного окружения Python, что сильно упрощает создание моделей Pyomo, применение различных оптимизаторов и изучение решений.

Благодаря многочисленным примерам, иллюстрирующим различные способы формулирования моделей, книга прекрасно раскрывает широту средств моделирования, поддерживаемых Pyomo, и ее подходы к сложным практическим приложениям.

В числе рассматриваемых тем:

- нелинейное программирование в Pyomo;
- структурное моделирование с помощью блоков;
- повышение производительности моделей;
- абстрактные модели и их решение;
- расширения моделирования.

Издание предназначено для начинающих и опытных разработчиков моделей, в том числе студентов старших курсов и аспирантов, научных работников и инженеров-практиков.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

Springer

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-93700-230-3