



THE
POWER
TO KNOW.

SAS/OR[®] 9.3 User's Guide Mathematical Programming Examples



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2012. *SAS/OR® 9.3 User's Guide: Mathematical Programming Examples*. Cary, NC: SAS Institute Inc.

SAS/OR® 9.3 User's Guide: Mathematical Programming Examples

Copyright © 2012, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-61290-262-3

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, April 2012

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at support.sas.com/publishing or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

Introduction	i
Chapter 1. Food Manufacture 1: When to Buy and How to Blend	1
Chapter 2. Food Manufacture 2: Limiting the Number of Ingredients and Adding Extra Conditions	23
Chapter 3. Factory Planning 1: What to Make, On What Machines, and When	31
Chapter 4. Factory Planning 2: When Should Machines Be Down for Maintenance	41
Chapter 5. Manpower Planning: How to Recruit, Retrain, Make Redundant, or Overman	49
Chapter 6. Refinery Optimization: How to Run an Oil Refinery	63
Chapter 7. Mining: Which Pits to Work and When to Close Them Down	79
Chapter 8. Farm Planning: How Much to Grow and Rear	89
Chapter 9. Economic Planning: How Should an Economy Grow	103
Chapter 10. Decentralization: How to Disperse Offices from the Capital	119
Chapter 11. Curve Fitting: Fitting a Curve to a Set of Data Points	131
Chapter 12. Logical Design: Constructing an Electronic System with a Minimum Number of Components	147
Chapter 13. Market Sharing: Assigning Retailers to Company Divisions	157
Chapter 14. Opencast Mining: How Much to Excavate	169
Chapter 15. Tariff Rates (Power Generation): How to Determine Tariff Rates for the Sale of Electricity	177
Chapter 16. Hydro Power: How to Generate and Combine Hydro and Thermal Electricity Generation	189
Chapter 17. Three-Dimensional Noughts and Crosses: A Combinatorial Problem	199
Chapter 18. Optimizing a Constraint: Reconstructing an Integer Programming Constraint More Simply	207
Chapter 19. Distribution 1: Which Factories and Depots to Supply Which Customers	215
Chapter 20. Depot Location (Distribution 2): Where Should New Depots Be Built	231
Chapter 21. Agricultural Pricing: What Prices to Charge for Dairy Products	241
Chapter 22. Efficiency Analysis: How to Use Data Envelopment Analysis to Compare Efficiencies of Garages	251
Chapter 23. Milk Collection: How to Route and Assign Milk Collection Lorries to Farms	263
Chapter 24. Yield Management: What Quantities of Airline Tickets to Sell at What Prices and What Times	281

Subject Index	313
----------------------	------------

Syntax Index	315
---------------------	------------

Introduction

This book contains all 24 examples from the classic book *Model Building in Mathematical Programming* by H. Paul Williams. For each example, the problem statement is first repeated verbatim from Williams' book.¹ Then the problem is solved using the OPTMODEL procedure in SAS/OR software.

The examples cover linear programming, mixed integer linear programming, and quadratic programming. In most cases, the problem is solved with a single call to one of the mathematical programming solvers available in PROC OPTMODEL. The purpose of this book is to supplement the *SAS/OR User's Guide: Mathematical Programming* with additional examples that demonstrate best practices.

Each chapter contains five sections, described as follows.

- **Problem Statement:**
Repeats verbatim the problem description, including any tables and figures, from Williams' book.
- **Mathematical Programming Formulation:**
Describes the index sets, parameters, decision variables, objectives, and constraints for one formulation of the problem.
- **Input Data:**
Creates the input data sets and macro variables to be used by PROC OPTMODEL.
- **PROC OPTMODEL Statements and Output:**
Shows and discusses the PROC OPTMODEL statements that declare sets and parameters, read the input data, formulate the mathematical programming problem, solve the problem, and output the solution. Also shows the output created by PROC OPTMODEL and occasionally other SAS procedures.
- **Features Demonstrated:**
Lists the important PROC OPTMODEL features demonstrated in this example.

Although PROC OPTMODEL is case-insensitive, in the interest of clarity a few typographical conventions are observed regarding capitalization of names:

OPTMODEL Expression	Capitalization
Index set names	All uppercase
Index set member names	All lowercase
Parameter names	All lowercase
Variable names (including implicit variables)	First letter of each word
Objective names	First letter of each word
Constraint names	First letter

¹ Figures and tables are numbered differently so that they match the chapter organization of this book. To be consistent with the verbatim problem statement, all other sections use British spelling. However, for clarity, large numbers and decimals are punctuated in American style (for example, 10,000 instead of 10 000 and 0.5 instead of 0.5), words are occasionally added or changed (with the changes shown inside square brackets), and punctuation is occasionally changed.

The examples shown here are small and not computationally challenging. Throughout, a separation between data and model is maintained so that you can solve larger or more difficult instances without modifying the PROC OPTMODEL statements. A user who learns the techniques demonstrated in these examples will be well-prepared to use PROC OPTMODEL to tackle similar modeling challenges that arise in real-world problems.

Chapter 1

Food Manufacture 1: When to Buy and How to Blend

Contents

Problem Statement	1
Mathematical Programming Formulation	2
Input Data	4
PROC OPTMODEL Statements and Output	5
Features Demonstrated	21

Problem Statement

A food is manufactured by refining raw oils and blending them together.¹ The raw oils come in two categories:

vegetable oils	VEG 1
	VEG 2
non-vegetable oils	OIL 1
	OIL 2
	OIL 3

Each oil may be purchased for immediate delivery (January) or bought on the futures market for delivery in a subsequent month. Prices now and in the futures market are given below (in £/ton):

	VEG 1	VEG 2	OIL 1	OIL 2	OIL 3
January	110	120	130	110	115
February	130	130	110	90	115
March	110	140	130	100	95
April	120	110	120	120	125
May	100	120	150	110	105
June	90	100	140	80	135

The final product sells at £150 per ton.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 231–232).

Vegetable oils and non-vegetable oils require different production lines for refining. In any month it is not possible to refine more than 200 tons of vegetable oils and more than 250 tons of non-vegetable oils. There is no loss of weight in the refining process and the cost of refining may be ignored.

It is possible to store up to 1000 tons of each raw oil for use later. The cost of storage for vegetable and non-vegetable oil is £5 per ton per month. The final product cannot be stored, nor can refined oils be stored.

There is a technological restriction of hardness on the final product. In the units in which hardness is measured this must lie between 3 and 6. It is assumed that hardness blends linearly and that the hardnesses of the raw oils are

VEG 1 8.8

VEG 2 6.1

OIL 1 2.0

OIL 2 4.2

OIL 3 5.0

What buying and manufacturing policy should the company pursue in order to maximize profit?

At present there are 500 tons of each type of raw oil in storage. It is required that these stocks will also exist at the end of June.

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{oil} \in \text{OILS}$
- $\text{period} \in \text{PERIODS}$
- $\text{VEG} \subseteq \text{OILS}$: vegetable oils
- $\text{NONVEG} = \text{OILS} \setminus \text{VEG}$: non-vegetable oils

Parameters

Table 1.1 shows the parameters used in this example.

Table 1.1 Parameters

Parameter Name	Interpretation
<i>cost[oil,period]</i>	Cost of raw oil per period
<i>hardness[oil]</i>	Hardness of raw oil
<i>revenue_per_ton</i>	Revenue per ton of final product
<i>veg_ub</i>	Tons of vegetable oils that can be refined per period
<i>nonveg_ub</i>	Tons of non-vegetable oils that can be refined per period
<i>store_ub</i>	Tons of raw oil that can be stored per period
<i>storage_cost_per_ton</i>	Storage cost per ton of raw oil per period
<i>hardness_lb</i>	Lower bound on hardness of final product
<i>hardness_ub</i>	Upper bound on hardness of final product
<i>init_storage</i>	Initial tons of each type of raw oil in storage
<i>hardness_sol[period]</i>	Hardness of final product per period

Variables

Table 1.2 shows the variables used in this example.

Table 1.2 Variables

Variable Name	Interpretation
Buy[oil,period]	Tons of raw oil to buy per period
Use[oil,period]	Tons of raw oil to use per period
Manufacture[period]	Tons of final product to manufacture per period
Store[oil,period]	Tons of raw oil to store as inventory at the end of each period

Objective

The objective is to maximize the following profit function, where Revenue, RawCost, and StorageCost are linear functions of Manufacture, Buy, and Store, respectively:

$$\text{Profit} = \text{Revenue} - \text{RawCost} - \text{StorageCost}$$

Constraints

The following constraints are used in this example:

- bounds on decision variables
- for period \in PERIODS,

$$\text{Manufacture}[\text{period}] = \sum_{\text{oil} \in \text{OILS}} \text{Use}[\text{oil}, \text{period}]$$

- for period \in PERIODS,

$$\sum_{oil \in VEG} \text{Use}[oil, period] \leq veg_ub$$

- for period \in PERIODS,

$$\sum_{oil \in NONVEG} \text{Use}[oil, period] \leq nonveg_ub$$

- for oil \in OILS and period \in PERIODS,

$$\text{Store}[oil, period - 1] + \text{Buy}[oil, period] = \text{Use}[oil, period] + \text{Store}[oil, period]$$

- for period \in PERIODS,

$$hardness_lb \leq \frac{\sum_{oil \in OILS} hardness[oil] \cdot \text{Use}[oil, period]}{\text{Manufacture}[period]} \leq hardness_ub$$

Input Data

The following data sets contain the input data that are used in this example:

```
data cost_data;
  input veg1-veg2 oil1-oil3;
  datalines;
110 120 130 110 115
130 130 110 90 115
110 140 130 100 95
120 110 120 120 125
100 120 150 110 105
90 100 140 80 135
;

data hardness_data;
  input oil $ hardness;
  datalines;
veg1 8.8
veg2 6.1
oil1 2.0
oil2 4.2
oil3 5.0
;
```

It is possible to store the other (scalar) parameters in an additional data set that contains one observation, with one data set variable per parameter. But the SAS macro language is used instead, with one macro variable per parameter.

```

%let revenue_per_ton = 150;
%let veg_ub = 200;
%let nonveg_ub = 250;
%let store_ub = 1000;
%let storage_cost_per_ton = 5;
%let hardness_lb = 3;
%let hardness_ub = 6;
%let init_storage = 500;

```

PROC OPTMODEL Statements and Output

The first READ DATA statement populates the OILS index set and reads the one-dimensional hardness data:

```

proc optmodel;
  set <str> OILS;
  num hardness {OILS};
  read data hardness_data into OILS=[oil] hardness;
  print hardness;

```

The PRINT statement results in the first section of output, shown in [Figure 1.1](#).

Figure 1.1 hardness Parameter

The OPTMODEL Procedure	
[1]	hardness
oil1	2.0
oil2	4.2
oil3	5.0
veg1	8.8
veg2	6.1

The second READ DATA statement populates the PERIODS index set and uses the already-populated OILS index set to loop across data set variables when reading the two-dimensional cost data. The PERIODS index set is numeric and is populated by using the automatic variable `_N_` from the `cost_data` data set, rather than by using the month names.

```

set PERIODS;
num cost {OILS, PERIODS};
read data cost_data into PERIODS=[_N_] {oil in OILS} <cost[oil,_N_]=col(oil)>;
print cost;

```

The PRINT statement results in the second section of output, shown in [Figure 1.2](#).

Figure 1.2 cost Parameter

	cost					
	1	2	3	4	5	6
oil1	130	110	130	120	150	140
oil2	110	90	100	120	110	80
oil3	115	115	95	125	105	135
veg1	110	130	110	120	100	90
veg2	120	130	140	110	120	100

You can declare implicit variables with the IMPVAR statement, instead of defining explicit variables by using the VAR statement with an additional constraint. When you use the IMPVAR statement, PROC OPTMODEL performs an algebraic substitution, thereby reducing the number of variables and constraints passed to the solver.

```
var Buy {OILS, PERIODS} >= 0;
var Use {OILS, PERIODS} >= 0;
impvar Manufacture {period in PERIODS} = sum {oil in OILS} Use[oil,period];
```

The initial and terminal storage constraints for each raw oil are imposed by using the FIX statement to fix the values of the corresponding Store[oil,period] variables. An alternate approach is to use the CON statement to explicitly declare an equality constraint that contains exactly one variable.

```
num last_period = max {period in PERIODS} period;
var Store {OILS, PERIODS union {0}} >= 0 <= &store_ub;
for {oil in OILS} do;
    fix Store[oil,0] = &init_storage;
    fix Store[oil,last_period] = &init_storage;
end;
```

The following SET statement uses the SAS function SUBSTR together with the colon operator (:) to select the subset of oils whose name starts with 'veg':

```
set VEG = {oil in OILS: substr(oil,1,3) = 'veg'};
```

The following SET statement uses the DIFF operator to declare the non-vegetable oils to be the oils that do not appear in the set VEG:

```
set NONVEG = OILS diff VEG;
```

The following statements declare implicit variables, the objective, and constraints:

```
impvar Revenue =
    sum {period in PERIODS} &revenue_per_ton * Manufacture[period];
impvar RawCost =
    sum {oil in OILS, period in PERIODS} cost[oil,period] * Buy[oil,period];
impvar StorageCost =
    sum {oil in OILS, period in PERIODS} &storage_cost_per_ton * Store[oil,period];
max Profit = Revenue - RawCost - StorageCost;
```



```

con Veg_ub_con {period in PERIODS}:
    sum {oil in VEG} Use[oil,period] <= &veg_ub;

con Nonveg_ub_con {period in PERIODS}:
    sum {oil in NONVEG} Use[oil,period] <= &nonveg_ub;

con Flow_balance_con {oil in OILS, period in PERIODS}:
    Store[oil,period-1] + Buy[oil,period] = Use[oil,period] + Store[oil,period];

```

As expressed on page 4, the hardness of the final product is a ratio of linear functions of the decision variables. To increase algorithmic performance and reliability, the following two CON statements take advantage of the constant limits on hardness to linearize the nonlinear range constraint by clearing the denominator:

```

con Hardness_ub_con {period in PERIODS}:
    sum {oil in OILS} hardness[oil] * Use[oil,period]
    >= &hardness_lb * Manufacture[period];

con Hardness_lb_con {period in PERIODS}:
    sum {oil in OILS} hardness[oil] * Use[oil,period]
    <= &hardness_ub * Manufacture[period];

```

The following EXPAND statement displays the resulting model with all data populated, as shown in [Figure 1.3](#):

```

expand;

```

This optional statement is useful for debugging purposes, to make sure that the model created by PROC OPTMODEL is what you intended.

Figure 1.3 Output from EXPAND Statement

```

                                The OPTMODEL Procedure

Var Buy[veg1,1] >= 0
Var Buy[veg1,2] >= 0
Var Buy[veg1,3] >= 0
Var Buy[veg1,4] >= 0
Var Buy[veg1,5] >= 0
Var Buy[veg1,6] >= 0
Var Buy[veg2,1] >= 0
Var Buy[veg2,2] >= 0
Var Buy[veg2,3] >= 0
Var Buy[veg2,4] >= 0
Var Buy[veg2,5] >= 0
Var Buy[veg2,6] >= 0
Var Buy[oil1,1] >= 0
Var Buy[oil1,2] >= 0
Var Buy[oil1,3] >= 0
Var Buy[oil1,4] >= 0
Var Buy[oil1,5] >= 0
Var Buy[oil1,6] >= 0
Var Buy[oil2,1] >= 0
Var Buy[oil2,2] >= 0
Var Buy[oil2,3] >= 0
Var Buy[oil2,4] >= 0
Var Buy[oil2,5] >= 0
Var Buy[oil2,6] >= 0
Var Buy[oil3,1] >= 0
Var Buy[oil3,2] >= 0
Var Buy[oil3,3] >= 0
Var Buy[oil3,4] >= 0
Var Buy[oil3,5] >= 0
Var Buy[oil3,6] >= 0
Var Use[veg1,1] >= 0
Var Use[veg1,2] >= 0
Var Use[veg1,3] >= 0
Var Use[veg1,4] >= 0
Var Use[veg1,5] >= 0
Var Use[veg1,6] >= 0
Var Use[veg2,1] >= 0
Var Use[veg2,2] >= 0
Var Use[veg2,3] >= 0
Var Use[veg2,4] >= 0
Var Use[veg2,5] >= 0
Var Use[veg2,6] >= 0
Var Use[oil1,1] >= 0
Var Use[oil1,2] >= 0
Var Use[oil1,3] >= 0
Var Use[oil1,4] >= 0
Var Use[oil1,5] >= 0
Var Use[oil1,6] >= 0
Var Use[oil2,1] >= 0
Var Use[oil2,2] >= 0
Var Use[oil2,3] >= 0
Var Use[oil2,4] >= 0
Var Use[oil2,5] >= 0

```

Figure 1.3 continued

```

                                The OPTMODEL Procedure

Var Use[oil2,6] >= 0
Var Use[oil3,1] >= 0
Var Use[oil3,2] >= 0
Var Use[oil3,3] >= 0
Var Use[oil3,4] >= 0
Var Use[oil3,5] >= 0
Var Use[oil3,6] >= 0
Var Store[veg1,1] >= 0 <= 1000
Var Store[veg1,2] >= 0 <= 1000
Var Store[veg1,3] >= 0 <= 1000
Var Store[veg1,4] >= 0 <= 1000
Var Store[veg1,5] >= 0 <= 1000
Fix Store[veg1,6] = 500
Fix Store[veg1,0] = 500
Var Store[veg2,1] >= 0 <= 1000
Var Store[veg2,2] >= 0 <= 1000
Var Store[veg2,3] >= 0 <= 1000
Var Store[veg2,4] >= 0 <= 1000
Var Store[veg2,5] >= 0 <= 1000
Fix Store[veg2,6] = 500
Fix Store[veg2,0] = 500
Var Store[oil1,1] >= 0 <= 1000
Var Store[oil1,2] >= 0 <= 1000
Var Store[oil1,3] >= 0 <= 1000
Var Store[oil1,4] >= 0 <= 1000
Var Store[oil1,5] >= 0 <= 1000
Fix Store[oil1,6] = 500
Fix Store[oil1,0] = 500
Var Store[oil2,1] >= 0 <= 1000
Var Store[oil2,2] >= 0 <= 1000
Var Store[oil2,3] >= 0 <= 1000
Var Store[oil2,4] >= 0 <= 1000
Var Store[oil2,5] >= 0 <= 1000
Fix Store[oil2,6] = 500
Fix Store[oil2,0] = 500
Var Store[oil3,1] >= 0 <= 1000
Var Store[oil3,2] >= 0 <= 1000
Var Store[oil3,3] >= 0 <= 1000
Var Store[oil3,4] >= 0 <= 1000
Var Store[oil3,5] >= 0 <= 1000
Fix Store[oil3,6] = 500
Fix Store[oil3,0] = 500
Impvar Manufacture[1] = Use[veg1,1] + Use[veg2,1] + Use[oil1,1] + Use[oil2,1] +
Use[oil3,1]
Impvar Manufacture[2] = Use[veg1,2] + Use[veg2,2] + Use[oil1,2] + Use[oil2,2] +
Use[oil3,2]
Impvar Manufacture[3] = Use[veg1,3] + Use[veg2,3] + Use[oil1,3] + Use[oil2,3] +
Use[oil3,3]
Impvar Manufacture[4] = Use[veg1,4] + Use[veg2,4] + Use[oil1,4] + Use[oil2,4] +
Use[oil3,4]
Impvar Manufacture[5] = Use[veg1,5] + Use[veg2,5] + Use[oil1,5] + Use[oil2,5] +
Use[oil3,5]
Impvar Manufacture[6] = Use[veg1,6] + Use[veg2,6] + Use[oil1,6] + Use[oil2,6] +

```

Figure 1.3 continued

```

The OPTMODEL Procedure

Use[oil3,6]
Impvar Revenue = 150*Manufacture[1] + 150*Manufacture[2] + 150*Manufacture[3] +
150*Manufacture[4] + 150*Manufacture[5] + 150*Manufacture[6]
Impvar RawCost = 110*Buy[veg1,1] + 130*Buy[veg1,2] + 110*Buy[veg1,3] + 120*
Buy[veg1,4] + 100*Buy[veg1,5] + 90*Buy[veg1,6] + 120*Buy[veg2,1] + 130*
Buy[veg2,2] + 140*Buy[veg2,3] + 110*Buy[veg2,4] + 120*Buy[veg2,5] + 100*
Buy[veg2,6] + 130*Buy[oil1,1] + 110*Buy[oil1,2] + 130*Buy[oil1,3] + 120*
Buy[oil1,4] + 150*Buy[oil1,5] + 140*Buy[oil1,6] + 110*Buy[oil2,1] + 90*
Buy[oil2,2] + 100*Buy[oil2,3] + 120*Buy[oil2,4] + 110*Buy[oil2,5] + 80*
Buy[oil2,6] + 115*Buy[oil3,1] + 115*Buy[oil3,2] + 95*Buy[oil3,3] + 125*
Buy[oil3,4] + 105*Buy[oil3,5] + 135*Buy[oil3,6]
Impvar StorageCost = 5*Store[veg1,1] + 5*Store[veg1,2] + 5*Store[veg1,3] + 5*
Store[veg1,4] + 5*Store[veg1,5] + 5*Store[veg1,6] + 5*Store[veg2,1] + 5*
Store[veg2,2] + 5*Store[veg2,3] + 5*Store[veg2,4] + 5*Store[veg2,5] + 5*
Store[veg2,6] + 5*Store[oil1,1] + 5*Store[oil1,2] + 5*Store[oil1,3] + 5*
Store[oil1,4] + 5*Store[oil1,5] + 5*Store[oil1,6] + 5*Store[oil2,1] + 5*
Store[oil2,2] + 5*Store[oil2,3] + 5*Store[oil2,4] + 5*Store[oil2,5] + 5*
Store[oil2,6] + 5*Store[oil3,1] + 5*Store[oil3,2] + 5*Store[oil3,3] + 5*
Store[oil3,4] + 5*Store[oil3,5] + 5*Store[oil3,6]
Maximize Profit=- StorageCost - RawCost + Revenue
Constraint Veg_ub_con[1]: Use[veg1,1] + Use[veg2,1] <= 200
Constraint Veg_ub_con[2]: Use[veg1,2] + Use[veg2,2] <= 200
Constraint Veg_ub_con[3]: Use[veg1,3] + Use[veg2,3] <= 200
Constraint Veg_ub_con[4]: Use[veg1,4] + Use[veg2,4] <= 200
Constraint Veg_ub_con[5]: Use[veg1,5] + Use[veg2,5] <= 200
Constraint Veg_ub_con[6]: Use[veg1,6] + Use[veg2,6] <= 200
Constraint Nonveg_ub_con[1]: Use[oil1,1] + Use[oil2,1] + Use[oil3,1] <= 250
Constraint Nonveg_ub_con[2]: Use[oil1,2] + Use[oil2,2] + Use[oil3,2] <= 250
Constraint Nonveg_ub_con[3]: Use[oil1,3] + Use[oil2,3] + Use[oil3,3] <= 250
Constraint Nonveg_ub_con[4]: Use[oil1,4] + Use[oil2,4] + Use[oil3,4] <= 250
Constraint Nonveg_ub_con[5]: Use[oil1,5] + Use[oil2,5] + Use[oil3,5] <= 250
Constraint Nonveg_ub_con[6]: Use[oil1,6] + Use[oil2,6] + Use[oil3,6] <= 250
Constraint Flow_balance_con[veg1,1]: - Store[veg1,1] - Use[veg1,1] + Buy[veg1,1]
+ Store[veg1,0] = 0
Constraint Flow_balance_con[veg1,2]: - Store[veg1,2] - Use[veg1,2] + Buy[veg1,2]
+ Store[veg1,1] = 0
Constraint Flow_balance_con[veg1,3]: - Store[veg1,3] - Use[veg1,3] + Buy[veg1,3]
+ Store[veg1,2] = 0
Constraint Flow_balance_con[veg1,4]: - Store[veg1,4] - Use[veg1,4] + Buy[veg1,4]
+ Store[veg1,3] = 0
Constraint Flow_balance_con[veg1,5]: - Store[veg1,5] - Use[veg1,5] + Buy[veg1,5]
+ Store[veg1,4] = 0
Constraint Flow_balance_con[veg1,6]: - Store[veg1,6] - Use[veg1,6] + Buy[veg1,6]
+ Store[veg1,5] = 0
Constraint Flow_balance_con[veg2,1]: - Store[veg2,1] - Use[veg2,1] + Buy[veg2,1]
+ Store[veg2,0] = 0
Constraint Flow_balance_con[veg2,2]: - Store[veg2,2] - Use[veg2,2] + Buy[veg2,2]
+ Store[veg2,1] = 0
Constraint Flow_balance_con[veg2,3]: - Store[veg2,3] - Use[veg2,3] + Buy[veg2,3]
+ Store[veg2,2] = 0
Constraint Flow_balance_con[veg2,4]: - Store[veg2,4] - Use[veg2,4] + Buy[veg2,4]
+ Store[veg2,3] = 0
Constraint Flow_balance_con[veg2,5]: - Store[veg2,5] - Use[veg2,5] + Buy[veg2,5]

```

Figure 1.3 continued

```

The OPTMODEL Procedure

+ Store[veg2,4] = 0
Constraint Flow_balance_con[veg2,6]: - Store[veg2,6] - Use[veg2,6] + Buy[veg2,6]
+ Store[veg2,5] = 0
Constraint Flow_balance_con[oil1,1]: - Store[oil1,1] - Use[oil1,1] + Buy[oil1,1]
+ Store[oil1,0] = 0
Constraint Flow_balance_con[oil1,2]: - Store[oil1,2] - Use[oil1,2] + Buy[oil1,2]
+ Store[oil1,1] = 0
Constraint Flow_balance_con[oil1,3]: - Store[oil1,3] - Use[oil1,3] + Buy[oil1,3]
+ Store[oil1,2] = 0
Constraint Flow_balance_con[oil1,4]: - Store[oil1,4] - Use[oil1,4] + Buy[oil1,4]
+ Store[oil1,3] = 0
Constraint Flow_balance_con[oil1,5]: - Store[oil1,5] - Use[oil1,5] + Buy[oil1,5]
+ Store[oil1,4] = 0
Constraint Flow_balance_con[oil1,6]: - Store[oil1,6] - Use[oil1,6] + Buy[oil1,6]
+ Store[oil1,5] = 0
Constraint Flow_balance_con[oil2,1]: - Store[oil2,1] - Use[oil2,1] + Buy[oil2,1]
+ Store[oil2,0] = 0
Constraint Flow_balance_con[oil2,2]: - Store[oil2,2] - Use[oil2,2] + Buy[oil2,2]
+ Store[oil2,1] = 0
Constraint Flow_balance_con[oil2,3]: - Store[oil2,3] - Use[oil2,3] + Buy[oil2,3]
+ Store[oil2,2] = 0
Constraint Flow_balance_con[oil2,4]: - Store[oil2,4] - Use[oil2,4] + Buy[oil2,4]
+ Store[oil2,3] = 0
Constraint Flow_balance_con[oil2,5]: - Store[oil2,5] - Use[oil2,5] + Buy[oil2,5]
+ Store[oil2,4] = 0
Constraint Flow_balance_con[oil2,6]: - Store[oil2,6] - Use[oil2,6] + Buy[oil2,6]
+ Store[oil2,5] = 0
Constraint Flow_balance_con[oil3,1]: - Store[oil3,1] - Use[oil3,1] + Buy[oil3,1]
+ Store[oil3,0] = 0
Constraint Flow_balance_con[oil3,2]: - Store[oil3,2] - Use[oil3,2] + Buy[oil3,2]
+ Store[oil3,1] = 0
Constraint Flow_balance_con[oil3,3]: - Store[oil3,3] - Use[oil3,3] + Buy[oil3,3]
+ Store[oil3,2] = 0
Constraint Flow_balance_con[oil3,4]: - Store[oil3,4] - Use[oil3,4] + Buy[oil3,4]
+ Store[oil3,3] = 0
Constraint Flow_balance_con[oil3,5]: - Store[oil3,5] - Use[oil3,5] + Buy[oil3,5]
+ Store[oil3,4] = 0
Constraint Flow_balance_con[oil3,6]: - Store[oil3,6] - Use[oil3,6] + Buy[oil3,6]
+ Store[oil3,5] = 0
Constraint Hardness_ub_con[1]: - 3*Manufacture[1] + 8.8*Use[veg1,1] + 6.1*
Use[veg2,1] + 2*Use[oil1,1] + 4.2*Use[oil2,1] + 5*Use[oil3,1] >= 0
Constraint Hardness_ub_con[2]: - 3*Manufacture[2] + 8.8*Use[veg1,2] + 6.1*
Use[veg2,2] + 2*Use[oil1,2] + 4.2*Use[oil2,2] + 5*Use[oil3,2] >= 0
Constraint Hardness_ub_con[3]: - 3*Manufacture[3] + 8.8*Use[veg1,3] + 6.1*
Use[veg2,3] + 2*Use[oil1,3] + 4.2*Use[oil2,3] + 5*Use[oil3,3] >= 0
Constraint Hardness_ub_con[4]: - 3*Manufacture[4] + 8.8*Use[veg1,4] + 6.1*
Use[veg2,4] + 2*Use[oil1,4] + 4.2*Use[oil2,4] + 5*Use[oil3,4] >= 0
Constraint Hardness_ub_con[5]: - 3*Manufacture[5] + 8.8*Use[veg1,5] + 6.1*
Use[veg2,5] + 2*Use[oil1,5] + 4.2*Use[oil2,5] + 5*Use[oil3,5] >= 0
Constraint Hardness_ub_con[6]: - 3*Manufacture[6] + 8.8*Use[veg1,6] + 6.1*
Use[veg2,6] + 2*Use[oil1,6] + 4.2*Use[oil2,6] + 5*Use[oil3,6] >= 0
Constraint Hardness_lb_con[1]: - 6*Manufacture[1] + 8.8*Use[veg1,1] + 6.1*
Use[veg2,1] + 2*Use[oil1,1] + 4.2*Use[oil2,1] + 5*Use[oil3,1] <= 0

```

Figure 1.3 continued

```

The OPTMODEL Procedure

Constraint Hardness_lb_con[2]: - 6*Manufacture[2] + 8.8*Use[veg1,2] + 6.1*
Use[veg2,2] + 2*Use[oil1,2] + 4.2*Use[oil2,2] + 5*Use[oil3,2] <= 0
Constraint Hardness_lb_con[3]: - 6*Manufacture[3] + 8.8*Use[veg1,3] + 6.1*
Use[veg2,3] + 2*Use[oil1,3] + 4.2*Use[oil2,3] + 5*Use[oil3,3] <= 0
Constraint Hardness_lb_con[4]: - 6*Manufacture[4] + 8.8*Use[veg1,4] + 6.1*
Use[veg2,4] + 2*Use[oil1,4] + 4.2*Use[oil2,4] + 5*Use[oil3,4] <= 0
Constraint Hardness_lb_con[5]: - 6*Manufacture[5] + 8.8*Use[veg1,5] + 6.1*
Use[veg2,5] + 2*Use[oil1,5] + 4.2*Use[oil2,5] + 5*Use[oil3,5] <= 0
Constraint Hardness_lb_con[6]: - 6*Manufacture[6] + 8.8*Use[veg1,6] + 6.1*
Use[veg2,6] + 2*Use[oil1,6] + 4.2*Use[oil2,6] + 5*Use[oil3,6] <= 0

```

By using the `.sol` suffix, the numeric parameter `hardness_sol` computes hardness of the final product from the optimal decision variable values returned by the solver:

```

num hardness_sol {period in PERIODS} =
    (sum {oil in OILS} hardness[oil] * Use[oil,period].sol) / Manufacture[period].sol;

```

You can declare `hardness_sol` even before the solver is called. Because the declaration includes an equals sign, the values are automatically updated each time the right-hand side changes. The following statements call the solver and print the solution:

```

solve;
print Buy Use Store Manufacture hardness_sol;

```

Multiple CREATE DATA statements, with the variables of interest grouped according to their index sets, create multiple output data sets (not shown):

```

create data sol_data1 from [oil period] Buy Use Store;
create data sol_data2 from [period] Manufacture;

```

In this example, all variables are real, the objective function is linear, and all constraints are linear. So PROC OPTMODEL automatically recognizes that this model is a linear programming problem, and the first SOLVE statement calls the default linear programming solver, which is dual simplex. To invoke a non-default solver (such as primal simplex, interior point, or network simplex), you can use the SOLVER= option in the SOLVE statement:

```

solve with lp / solver=ps;
print Buy Use Store Manufacture hardness_sol;
solve with lp / solver=ii;
print Buy Use Store Manufacture hardness_sol;
solve with lp / solver=ns;
print Buy Use Store Manufacture hardness_sol;
quit;

```

Each solver returns an optimal solution with a profit of £107,843, although the optimal solutions differ from each other, as shown in Figure 1.4 through Figure 1.7.

Figure 1.4 shows the output when you use the (default) dual simplex solver.

Figure 1.4 Output from Dual Simplex Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	Profit
Objective Type	Linear
Number of Variables	95
Bounded Above	0
Bounded Below	60
Bounded Below and Above	25
Free	0
Fixed	10
Number of Constraints	54
Linear LE (\leq)	18
Linear EQ ($=$)	30
Linear GE (\geq)	6
Linear Range	0
Constraint Coefficients	210
Solution Summary	
Solver	Dual Simplex
Objective Function	Profit
Solution Status	Optimal
Objective Value	107842.59259
Iterations	65
Primal Infeasibility	4.440892E-14
Dual Infeasibility	4.964779E-15
Bound Infeasibility	0

Figure 1.4 continued

[1]	[2]	Buy	Use	Store
oil1	0			500.000
oil1	1	0.00	0.000	500.000
oil1	2	0.00	0.000	500.000
oil1	3	0.00	0.000	500.000
oil1	4	0.00	0.000	500.000
oil1	5	0.00	0.000	500.000
oil1	6	0.00	0.000	500.000
oil2	0			500.000
oil2	1	0.00	250.000	250.000
oil2	2	0.00	0.000	250.000
oil2	3	0.00	0.000	250.000
oil2	4	0.00	250.000	0.000
oil2	5	0.00	0.000	0.000
oil2	6	750.00	250.000	500.000
oil3	0			500.000
oil3	1	0.00	0.000	500.000
oil3	2	0.00	250.000	250.000
oil3	3	0.00	250.000	0.000
oil3	4	0.00	0.000	0.000
oil3	5	750.00	250.000	500.000
oil3	6	0.00	0.000	500.000
veg1	0			500.000
veg1	1	0.00	159.259	340.741
veg1	2	0.00	85.185	255.556
veg1	3	0.00	85.185	170.370
veg1	4	0.00	85.185	85.185
veg1	5	0.00	85.185	0.000
veg1	6	659.26	159.259	500.000
veg2	0			500.000
veg2	1	0.00	40.741	459.259
veg2	2	0.00	114.815	344.444
veg2	3	0.00	114.815	229.630
veg2	4	0.00	114.815	114.815
veg2	5	0.00	114.815	0.000
veg2	6	540.74	40.741	500.000

[1]	hardness_	
	Manufacture	sol
1	450	6.0000
2	450	6.0000
3	450	6.0000
4	450	5.5556
5	450	6.0000
6	450	6.0000

Figure 1.5 shows the output when you use the SOLVER=PS option to invoke the primal simplex solver.

Figure 1.5 Output from Primal Simplex Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	Profit
Objective Type	Linear
Number of Variables	95
Bounded Above	0
Bounded Below	60
Bounded Below and Above	25
Free	0
Fixed	10
Number of Constraints	54
Linear LE (<=)	18
Linear EQ (=)	30
Linear GE (>=)	6
Linear Range	0
Constraint Coefficients	210
Solution Summary	
Solver	Primal Simplex
Objective Function	Profit
Solution Status	Optimal
Objective Value	107842.59259
Iterations	49
Primal Infeasibility	5.684342E-14
Dual Infeasibility	4.964779E-15
Bound Infeasibility	5.947506E-14

Figure 1.5 continued

[1]	[2]	Buy	Use	Store
oil1	0			500.000
oil1	1	0.00	0.000	500.000
oil1	2	0.00	0.000	500.000
oil1	3	0.00	0.000	500.000
oil1	4	0.00	0.000	500.000
oil1	5	0.00	0.000	500.000
oil1	6	0.00	0.000	500.000
oil2	0			500.000
oil2	1	0.00	250.000	250.000
oil2	2	750.00	250.000	750.000
oil2	3	0.00	250.000	500.000
oil2	4	0.00	250.000	250.000
oil2	5	0.00	250.000	0.000
oil2	6	750.00	250.000	500.000
oil3	0			500.000
oil3	1	0.00	0.000	500.000
oil3	2	0.00	0.000	500.000
oil3	3	0.00	0.000	500.000
oil3	4	0.00	0.000	500.000
oil3	5	0.00	0.000	500.000
oil3	6	0.00	0.000	500.000
veg1	0			500.000
veg1	1	0.00	159.259	340.741
veg1	2	0.00	159.259	181.481
veg1	3	0.00	159.259	22.222
veg1	4	0.00	0.000	22.222
veg1	5	0.00	22.222	0.000
veg1	6	659.26	159.259	500.000
veg2	0			500.000
veg2	1	0.00	40.741	459.259
veg2	2	0.00	40.741	418.519
veg2	3	0.00	40.741	377.778
veg2	4	0.00	200.000	177.778
veg2	5	0.00	177.778	-0.000
veg2	6	540.74	40.741	500.000

[1]	hardness_	
	Manufacture	sol
1	450	6.0000
2	450	6.0000
3	450	6.0000
4	450	5.0444
5	450	5.1778
6	450	6.0000

Figure 1.6 shows the output when you use the SOLVER=II option to invoke the interior point solver.

Figure 1.6 Output from Interior Point Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	Profit
Objective Type	Linear
Number of Variables	95
Bounded Above	0
Bounded Below	60
Bounded Below and Above	25
Free	0
Fixed	10
Number of Constraints	54
Linear LE (\leq)	18
Linear EQ ($=$)	30
Linear GE (\geq)	6
Linear Range	0
Constraint Coefficients	210
Solution Summary	
Solver	Iterative Interior
Objective Function	Profit
Solution Status	Optimal
Objective Value	107842.59259
Iterations	10
Primal Infeasibility	5.387128E-17
Dual Infeasibility	3.043247E-17
Bound Infeasibility	2.758025E-17
Duality Gap	0

Figure 1.6 continued

[1]	[2]	Buy	Use	Store
oil1	0			500.00
oil1	1	0.00	0.000	500.00
oil1	2	0.00	0.000	500.00
oil1	3	0.00	0.000	500.00
oil1	4	0.00	0.000	500.00
oil1	5	0.00	0.000	500.00
oil1	6	0.00	0.000	500.00
oil2	0			500.00
oil2	1	0.00	167.102	332.90
oil2	2	338.22	188.085	483.03
oil2	3	0.00	178.491	304.54
oil2	4	0.00	159.310	145.23
oil2	5	0.00	145.232	0.00
oil2	6	750.00	250.000	500.00
oil3	0			500.00
oil3	1	0.00	82.898	417.10
oil3	2	0.00	61.915	355.19
oil3	3	207.56	71.509	491.24
oil3	4	0.00	90.690	400.55
oil3	5	204.22	104.768	500.00
oil3	6	0.00	0.000	500.00
veg1	0			500.00
veg1	1	0.00	94.427	405.57
veg1	2	0.00	96.819	308.75
veg1	3	0.00	103.891	204.86
veg1	4	0.00	103.633	101.23
veg1	5	0.00	101.230	0.00
veg1	6	659.26	159.259	500.00
veg2	0			500.00
veg2	1	0.00	105.573	394.43
veg2	2	0.00	103.181	291.25
veg2	3	0.00	96.109	195.14
veg2	4	0.00	96.367	98.77
veg2	5	0.00	98.770	0.00
veg2	6	540.74	40.741	500.00

[1]	Manufacture	hardness_
		sol
1	450	5.7584
2	450	5.7354
3	450	5.7949
4	450	5.8275
5	450	5.8381
6	450	6.0000

Figure 1.7 shows the output when you use the SOLVER=NS option to invoke the network simplex solver.

Figure 1.7 Output from Network Simplex Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	Profit
Objective Type	Linear
Number of Variables	95
Bounded Above	0
Bounded Below	60
Bounded Below and Above	25
Free	0
Fixed	10
Number of Constraints	54
Linear LE (<=)	18
Linear EQ (=)	30
Linear GE (>=)	6
Linear Range	0
Constraint Coefficients	210
Solution Summary	
Solver	Network Simplex
Objective Function	Profit
Solution Status	Optimal
Objective Value	107842.59259
Iterations	48
Iterations2	40
Primal Infeasibility	4.440892E-14
Dual Infeasibility	3.98119E-15
Bound Infeasibility	0

Figure 1.7 continued

[1]	[2]	Buy	Use	Store
oil1	0			500.000
oil1	1	0.00	0.000	500.000
oil1	2	0.00	0.000	500.000
oil1	3	0.00	0.000	500.000
oil1	4	0.00	0.000	500.000
oil1	5	0.00	0.000	500.000
oil1	6	0.00	0.000	500.000
oil2	0			500.000
oil2	1	0.00	0.000	500.000
oil2	2	0.00	250.000	250.000
oil2	3	0.00	0.000	250.000
oil2	4	0.00	0.000	250.000
oil2	5	0.00	250.000	0.000
oil2	6	750.00	250.000	500.000
oil3	0			500.000
oil3	1	0.00	250.000	250.000
oil3	2	0.00	0.000	250.000
oil3	3	250.00	250.000	250.000
oil3	4	0.00	250.000	0.000
oil3	5	500.00	0.000	500.000
oil3	6	0.00	0.000	500.000
veg1	0			500.000
veg1	1	0.00	85.185	414.815
veg1	2	0.00	159.259	255.556
veg1	3	0.00	85.185	170.370
veg1	4	0.00	11.111	159.259
veg1	5	0.00	159.259	0.000
veg1	6	659.26	159.259	500.000
veg2	0			500.000
veg2	1	0.00	114.815	385.185
veg2	2	0.00	40.741	344.444
veg2	3	0.00	114.815	229.630
veg2	4	0.00	188.889	40.741
veg2	5	0.00	40.741	0.000
veg2	6	540.74	40.741	500.000

[1]	hardness_	
	Manufacture	sol
1	450	6.0000
2	450	6.0000
3	450	6.0000
4	450	5.5556
5	450	6.0000
6	450	6.0000

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- numeric and string index sets
- reading dense two-dimensional data
- bounds in the VAR statement
- FIX statement
- IMPVAR statement
- MAX aggregation operator
- SUBSTR function
- using a colon (:) to select members of a set
- set operators DIFF and UNION
- linearizing a ratio constraint
- range constraint
- EXPAND statement
- using a variable suffix (such as `.sol`) in the declaration of a numeric parameter
- multiple input and output data sets
- SOLVER= option

Chapter 2

Food Manufacture 2: Limiting the Number of Ingredients and Adding Extra Conditions

Contents

Problem Statement	23
Mathematical Programming Formulation	23
Input Data	25
PROC OPTMODEL Statements and Output	25
Features Demonstrated	29

Problem Statement

It is wished to impose the following extra conditions on the food manufacture problem:¹

- (1) The food may never be made up of more than three oils in any month.
- (2) If an oil is used in a month at least 20 tons must be used.
- (3) If either of VEG 1 or VEG 2 is used in a month then OIL 3 must also be used.

Extend the food manufacture model to encompass these restrictions and find the new optimal solution.

Mathematical Programming Formulation

This formulation builds on the formulation used in [Chapter 1](#). This section includes only the new elements of the formulation.

Parameters

Table 2.1 shows the additional parameters used in this example.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, p. 232).

Table 2.1 Parameters

Parameter Name	Interpretation
<i>max_num_oils_used</i>	Maximum number of oils used per period
<i>min_oil_used_threshold</i>	Minimum tons of oil used per period if used in that period
<i>Use[oil,period].ub</i>	Upper bound on Use[oil,period]

Variables

Table 2.2 shows the additional variables used in this example.

Table 2.2 Variables

Variable Name	Interpretation
IsUsed[oil,period]	1 if Use[oil,period] is positive; 0 otherwise

Objective

The objective is the same as in Chapter 1.

Constraints

The following additional constraints are used in this example:

- for oil \in OILS and period \in PERIODS,

$$\text{Use}[\text{oil}, \text{period}] \leq \text{Use}[\text{oil}, \text{period}].ub \cdot \text{IsUsed}[\text{oil}, \text{period}]$$

- for period \in PERIODS,

$$\sum_{\text{oil} \in \text{OILS}} \text{IsUsed}[\text{oil}, \text{period}] \leq \text{max_num_oils_used}$$

- for oil \in OILS and period \in PERIODS,

$$\text{Use}[\text{oil}, \text{period}] \geq \text{min_oil_used_threshold} \cdot \text{IsUsed}[\text{oil}, \text{period}]$$

- for oil $\in \{ \text{'veg1'}, \text{'veg2'} \}$ and period \in PERIODS,

$$\text{IsUsed}[\text{oil}, \text{period}] \leq \text{IsUsed}[\text{'oil3'}, \text{period}]$$

Input Data

The following macro variables contain the additional input data that are used in this example:

```
%let max_num_oils_used = 3;
%let min_oil_used_threshold = 20;
```

PROC OPTMODEL Statements and Output

For completeness, all statements are shown. Statements that are new or changed from [Chapter 1](#) are indicated.

```
proc optmodel;
    set <str> OILS;
    num hardness {OILS};
    read data hardness_data into OILS=[oil] hardness;

    set PERIODS;
    num cost {OILS, PERIODS};
    read data cost_data into PERIODS=[_N_] {oil in OILS} <cost[oil,_N_]=col(oil)>;

    var Buy {OILS, PERIODS} >= 0;
    var Use {OILS, PERIODS} >= 0;
    impvar Manufacture {period in PERIODS} = sum {oil in OILS} Use[oil,period];

    num last_period = max {period in PERIODS} period;
    var Store {OILS, PERIODS union {0}} >= 0 <= &store_ub;
    for {oil in OILS} do;
        fix Store[oil,0] = &init_storage;
        fix Store[oil,last_period] = &init_storage;
    end;

    set VEG = {oil in OILS: substr(oil,1,3) = 'veg'};
    set NONVEG = OILS diff VEG;

    impvar Revenue =
        sum {period in PERIODS} &revenue_per_ton * Manufacture[period];
    impvar RawCost =
        sum {oil in OILS, period in PERIODS} cost[oil,period] * Buy[oil,period];
    impvar StorageCost =
        sum {oil in OILS, period in PERIODS} &storage_cost_per_ton * Store[oil,period];
    max Profit = Revenue - RawCost - StorageCost;

    con Veg_ub_con {period in PERIODS}:
        sum {oil in VEG} Use[oil,period] <= &veg_ub;

    con Nonveg_ub_con {period in PERIODS}:
        sum {oil in NONVEG} Use[oil,period] <= &nonveg_ub;
```

```

con Flow_balance_con {oil in OILS, period in PERIODS}:
    Store[oil,period-1] + Buy[oil,period] = Use[oil,period] + Store[oil,period];

con Hardness_ub_con {period in PERIODS}:
    sum {oil in OILS} hardness[oil] * Use[oil,period]
    >= &hardness_lb * Manufacture[period];

con Hardness_lb_con {period in PERIODS}:
    sum {oil in OILS} hardness[oil] * Use[oil,period]
    <= &hardness_ub * Manufacture[period];

```

The remaining statements are new in this example. The BINARY option in the following VAR statement declares IsUsed to be a binary variable:

```
var IsUsed {OILS, PERIODS} binary;
```

The .ub variable suffix imposes an upper bound on the Use variable, in preparation for the subsequent Link constraint. The validity of this upper bound follows from the Veg_ub_con and Nonveg_ub_con constraints.

```

for {period in PERIODS} do;
    for {oil in VEG}    Use[oil,period].ub = &veg_ub;
    for {oil in NONVEG} Use[oil,period].ub = &nonveg_ub;
end;

```

The following Link constraint enforces the rule that Use[oil,period] > 0 implies that IsUsed[oil,period] = 1:

```

con Link {oil in OILS, period in PERIODS}:
    Use[oil,period] <= Use[oil,period].ub * IsUsed[oil,period];

```

The following Logical1, Logical2, and Logical3 constraints correspond directly to the three extra conditions in the problem statement:

```

con Logical1 {period in PERIODS}:
    sum {oil in OILS} IsUsed[oil,period] <= &max_num_oils_used;

con Logical2 {oil in OILS, period in PERIODS}:
    Use[oil,period] >= &min_oil_used_threshold * IsUsed[oil,period];

con Logical3 {oil in {'veg1','veg2'}, period in PERIODS}:
    IsUsed[oil,period] <= IsUsed['oil3',period];

num hardness_sol {period in PERIODS} =
    (sum {oil in OILS} hardness[oil] * Use[oil,period].sol) / Manufacture[period].sol;

```

Because PROC OPTMODEL automatically recognizes that this model is a mixed integer linear programming problem, the following SOLVE statement calls the MILP solver, as shown in [Figure 2.1](#):

```
solve;
```

Figure 2.1 Summaries from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Maximization
Objective Function	Profit
Objective Type	Linear
Number of Variables	125
Bounded Above	0
Bounded Below	30
Bounded Below and Above	85
Free	0
Fixed	10
Binary	30
Integer	0
Number of Constraints	132
Linear LE (<=)	66
Linear EQ (=)	30
Linear GE (>=)	36
Linear Range	0
Constraint Coefficients	384
Solution Summary	
Solver	MILP
Objective Function	Profit
Solution Status	Optimal
Objective Value	100278.7037
Iterations	4295
Best Bound	.
Nodes	473
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	2.3763523E-9
Bound Infeasibility	1.496092E-10
Integer Infeasibility	1.188176E-11

The following PRINT statement creates the output shown in [Figure 2.2](#):

```
print Buy Use Store IsUsed Manufacture hardness_sol Logical1.body;
```

The `.body` constraint suffix accesses the left-hand side value of the `Logical1` constraint. For each period, the solution uses no more than three oils, as shown in [Figure 2.2](#). The following CREATE DATA statements create multiple output data sets, as in [Chapter 1](#):

```

create data sol_data1 from [oil period] Buy Use Store IsUsed;
create data sol_data2 from [period] Manufacture;
quit;

```

Figure 2.2 Output from Mixed Integer Linear Programming Solver

[1]	[2]	Buy	Use	Store	Is Used
oil1	0			500.00	
oil1	1	0.00	0.000	500.00	0
oil1	2	0.00	0.000	500.00	-0
oil1	3	0.00	0.000	500.00	0
oil1	4	0.00	0.000	500.00	0
oil1	5	0.00	0.000	500.00	0
oil1	6	0.00	0.000	500.00	0
oil2	0			500.00	
oil2	1	0.00	230.000	270.00	1
oil2	2	190.00	-0.000	460.00	0
oil2	3	0.00	0.000	460.00	0
oil2	4	0.00	230.000	230.00	1
oil2	5	0.00	230.000	-0.00	1
oil2	6	730.00	230.000	500.00	1
oil3	0			500.00	
oil3	1	0.00	20.000	480.00	1
oil3	2	0.00	250.000	230.00	1
oil3	3	40.00	250.000	20.00	1
oil3	4	0.00	20.000	0.00	1
oil3	5	540.00	20.000	520.00	1
oil3	6	0.00	20.000	500.00	1
veg1	0			500.00	
veg1	1	0.00	0.000	500.00	-0
veg1	2	0.00	85.185	414.81	1
veg1	3	0.00	85.185	329.63	1
veg1	4	0.00	155.000	174.63	1
veg1	5	0.00	155.000	19.63	1
veg1	6	480.37	0.000	500.00	0
veg2	0			500.00	
veg2	1	0.00	200.000	300.00	1
veg2	2	0.00	114.815	185.19	1
veg2	3	0.00	114.815	70.37	1
veg2	4	0.00	0.000	70.37	0
veg2	5	0.00	0.000	70.37	0
veg2	6	629.63	200.000	500.00	1

[1]	Manufacture	hardness_ sol	Logical1. BODY
1	450	5.08	3
2	450	6.00	3
3	450	6.00	3
4	405	6.00	3
5	405	6.00	3
6	450	5.08	3

Note that the maximum profit of £100,279 is smaller than in [Chapter 1](#). This result is expected because this model contains additional constraints.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- numeric and string index sets
- reading dense two-dimensional data
- bounds in the VAR statement
- FIX statement
- IMPVAR statement
- MAX aggregation operator
- SUBSTR function
- using a colon (:) to select members of a set
- set operators DIFF and UNION
- using a variable suffix (such as `.so1`) in the declaration of a numeric parameter
- multiple input and output data sets
- BINARY option
- `.ub` variable suffix
- `.body` constraint suffix

Chapter 3

Factory Planning 1: What to Make, On What Machines, and When

Contents

Problem Statement	31
Mathematical Programming Formulation	32
Input Data	34
PROC OPTMODEL Statements and Output	35
Features Demonstrated	40

Problem Statement

An engineering factory makes seven products (PROD 1 to PROD 7) on the following machines: four grinders, two vertical drills, three horizontal drills, one borer, and one planer.¹ Each product yields a certain contribution to profit (defined as £/unit selling price minus cost of raw materials). These quantities (in £/unit) together with the unit production times (hours) required on each process are given below. A dash indicates that a product does not require a process.

	PROD 1	PROD 2	PROD 3	PROD 4	PROD 5	PROD 6	PROD 7
Contribution to profit	10	6	8	4	11	9	3
Grinding	0.5	0.7	—	—	0.3	0.2	0.5
Vertical drilling	0.1	0.2	—	0.3	—	0.6	—
Horizontal drilling	0.2	—	0.8	—	—	—	0.6
Boring	0.05	0.03	—	0.07	0.1	—	0.08
Planing	—	—	0.01	—	0.05	—	0.05

In the present month (January) and the five subsequent months certain machines will be down for maintenance. These machines will be:

January	1 grinder
February	2 horizontal drills
March	1 borer
April	1 vertical drill
May	1 grinder and 1 vertical drill
June	1 planer and 1 horizontal drill

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 233–234).

There are marketing limitations on each product in each month. These are:

	1	2	3	4	5	6	7
January	500	1000	300	300	800	200	100
February	600	500	200	0	400	300	150
March	300	600	0	0	500	400	100
April	200	300	400	500	200	0	100
May	0	100	500	100	1000	300	0
June	500	500	100	300	1100	500	60

It is possible to store up to 100 of each product at a time at a cost of £0.5 per unit per month. There are no stocks at present but it is desired to have a stock of 50 of each type of product at the end of June.

The factory works a 6 day week with two shifts of 8 hours each day.

No sequencing problems need to be considered.

When and what should the factory make in order to maximize the total profit? Recommend any price increases and the value of acquiring any new machines.

N.B. It may be assumed that each month consists of only 24 working days.

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{product} \in \text{PRODUCTS}$
- $\text{machine_type} \in \text{MACHINE_TYPES}$
- $\text{period} \in \text{PERIODS}$

Parameters

Table 3.1 shows the parameters used in this example.

Table 3.1 Parameters

Parameter Name	Interpretation
<i>profit[product]</i>	Profit per unit of product
<i>demand[product,period]</i>	Demand for each product per period
<i>num_machines[machine_type]</i>	Total number of machines for each machine type
<i>num_machines_per_period[machine_type,period]</i>	For each machine type, the number of machines available per period
<i>num_machines_down_per_period[machine_type,period]</i>	For each machine type, the number of machines down per period
<i>production_time[product,machine_type]</i>	Production time per unit of product on each machine type
<i>store_ub</i>	Number of units that can be stored for each product per period
<i>storage_cost_per_unit</i>	Storage cost per unit per period
<i>final_storage</i>	Number of units of each product in storage at the end of the last period
<i>num_hours_per_period</i>	Number of working hours per period (month)

Variables

Table 3.2 shows the variables used in this example.

Table 3.2 Variables

Variable Name	Interpretation
<i>Make[product,period]</i>	Number of units of each product to make per period
<i>Sell[product,period]</i>	Number of units of each product to sell per period
<i>Store[product,period]</i>	Number of units of each product to store as inventory at the end of each period

Objective

The objective is to maximize the following function, where *StorageCost* is a linear function of *Store*:

$$\text{TotalProfit} = \sum_{\text{product} \in \text{PRODUCTS}} \sum_{\text{period} \in \text{PERIODS}} \text{profit}[\text{product}] \text{Sell}[\text{product,period}] - \text{StorageCost}$$

Constraints

The following constraints are used in this example:

- bounds on decision variables

- for machine_type ∈ MACHINE_TYPES and period ∈ PERIODS,

$$\sum_{\text{product} \in \text{PRODUCTS}} \text{production_time}[\text{product}, \text{machine_type}] \cdot \text{Make}[\text{product}, \text{period}] \\ \leq \text{num_hours_per_period} \cdot \text{num_machines_per_period}[\text{machine_type}, \text{period}]$$

- for product ∈ PRODUCTS and period ∈ PERIODS,

$$\begin{aligned} & (\text{if period} - 1 \in \text{PERIODS, then Store}[\text{product}, \text{period} - 1]; \text{ else } 0) \\ & + \text{Make}[\text{product}, \text{period}] \\ & = \text{Sell}[\text{product}, \text{period}] + \text{Store}[\text{product}, \text{period}] \end{aligned}$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data product_data;
    input product $ profit;
    datalines;
prod1 10
prod2 6
prod3 8
prod4 4
prod5 11
prod6 9
prod7 3
;

data demand_data;
    input prod1-prod7;
    datalines;
500 1000 300 300 800 200 100
600 500 200 0 400 300 150
300 600 0 0 500 400 100
200 300 400 500 200 0 100
0 100 500 100 1000 300 0
500 500 100 300 1100 500 60
;

data machine_type_data;
    input machine_type $ num_machines;
    datalines;
grinder 4
vdrill 2
hdrill 3
borer 1
```

```

planer 1
;

data machine_type_period_data;
    input machine_type $ period num_down;
    datalines;
grinder 1 1
hdrill 2 2
borer 3 1
vdrill 4 1
grinder 5 1
vdrill 5 1
planer 6 1
hdrill 6 1
;

data machine_type_product_data;
    input machine_type $ prod1-prod7;
    datalines;
grinder 0.5 0.7 0 0 0.3 0.2 0.5
vdrill 0.1 0.2 0 0.3 0 0.6 0
hdrill 0.2 0 0.8 0 0 0 0.6
borer 0.05 0.03 0 0.07 0.1 0 0.08
planer 0 0 0.01 0 0.05 0 0.05
;

%let store_ub = 100;
%let storage_cost_per_unit = 0.5;
%let final_storage = 50;
%let num_hours_per_period = 24 * 2 * 8;

```

PROC OPTMODEL Statements and Output

This example uses both one-dimensional and dense two-dimensional data, as in [Chapter 1](#) and [Chapter 2](#):

```

proc optmodel;
    set <str> PRODUCTS;
    num profit {PRODUCTS};
    read data product_data into PRODUCTS=[product] profit;

    set PERIODS;
    num demand {PRODUCTS, PERIODS};
    read data demand_data into PERIODS=[_N_]
        {product in PRODUCTS} <demand[product, _N_]=col (product)>;

    set <str> MACHINE_TYPES;
    num num_machines {MACHINE_TYPES};
    read data machine_type_data into MACHINE_TYPES=[machine_type] num_machines;

```

But this problem also has sparse two-dimensional data: for most (machine_type, period) pairs, the number of machines down is 0. In the following statements, the INIT option in the

second NUM statement initializes *num_machines_down_per_period* to 0. The read of the sparse data set *machine_type_period_data* populates only the nonzero values. The subsequent computation of *num_machines_per_period[machine_type,period]* then uses the initial value of *num_machines_down_per_period[machine_type,period]* when no other value has been supplied:

```
num num_machines_per_period {machine_type in MACHINE_TYPES, PERIODS}
  init num_machines[machine_type];
num num_machines_down_per_period {MACHINE_TYPES, PERIODS} init 0;
read data machine_type_period_data into [machine_type period]
  num_machines_down_per_period=num_down;
for {machine_type in MACHINE_TYPES, period in PERIODS}
  num_machines_per_period[machine_type,period] =
    num_machines_per_period[machine_type,period]
    - num_machines_down_per_period[machine_type,period];
print num_machines_per_period;
```

Figure 3.1 shows the resulting values of *num_machines_per_period*.

Figure 3.1 *num_machines_per_period* Parameter

The OPTMODEL Procedure						
num_machines_per_period						
	1	2	3	4	5	6
borer	1	1	0	1	1	1
grinder	3	4	4	4	3	4
hdrill	3	1	3	3	3	2
planer	1	1	1	1	1	0
vdrill	2	2	2	1	1	2

The following statements declare and read dense two-dimensional data:

```
num production_time {PRODUCTS, MACHINE_TYPES};
read data machine_type_product_data into [machine_type]
  {product in PRODUCTS} <production_time[product,machine_type]=col(product)>;
```

The following statements are straightforward:

```

var Make {PRODUCTS, PERIODS} >= 0;
var Sell {product in PRODUCTS, period in PERIODS} >= 0 <= demand[product,period];

num last_period = max {period in PERIODS} period;
var Store {PRODUCTS, PERIODS} >= 0 <= &store_ub;
for {product in PRODUCTS}
    fix Store[product,last_period] = &final_storage;

impvar StorageCost =
    sum {product in PRODUCTS, period in PERIODS}
        &storage_cost_per_unit * Store[product,period];
max TotalProfit =
    sum {product in PRODUCTS, period in PERIODS}
        profit[product] * Sell[product,period]
    - StorageCost;

con Machine_hours_con {machine_type in MACHINE_TYPES, period in PERIODS}:
    sum {product in PRODUCTS}
        production_time[product,machine_type] * Make[product,period]
    <= &num_hours_per_period * num_machines_per_period[machine_type,period];

```

The following Flow_balance_con constraint uses an IF-THEN/ELSE expression to handle the boundary conditions: if a previous period exists, the units in storage at the end of the previous period are available to be sold in the current period. (Because ELSE 0 is the default, you could use just an IF-THEN expression instead.)

```

con Flow_balance_con {product in PRODUCTS, period in PERIODS}:
    (if period - 1 in PERIODS then Store[product,period-1] else 0)
    + Make[product,period]
    = Sell[product,period] + Store[product,period];

solve;
print Make Sell Store;

```

The maximum total profit is £93,715, as shown in [Figure 3.2](#).

Figure 3.2 Output from Linear Programming Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	126
Bounded Above	0
Bounded Below	42
Bounded Below and Above	71
Free	0
Fixed	13
Number of Constraints	72
Linear LE (\leq)	30
Linear EQ ($=$)	42
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	281
Solution Summary	
Solver	Dual Simplex
Objective Function	TotalProfit
Solution Status	Optimal
Objective Value	93715.178571
Iterations	29
Primal Infeasibility	1.421085E-14
Dual Infeasibility	2.220446E-16
Bound Infeasibility	0

Figure 3.2 continued

	[1]	[2]	Make	Sell	Store
prod1	1		500.00	500.00	0.0
prod1	2		700.00	600.00	100.0
prod1	3		0.00	100.00	0.0
prod1	4		200.00	200.00	0.0
prod1	5		0.00	0.00	0.0
prod1	6		550.00	500.00	50.0
prod2	1		888.57	888.57	0.0
prod2	2		600.00	500.00	100.0
prod2	3		0.00	100.00	0.0
prod2	4		300.00	300.00	0.0
prod2	5		100.00	100.00	0.0
prod2	6		550.00	500.00	50.0
prod3	1		382.50	300.00	82.5
prod3	2		117.50	200.00	0.0
prod3	3		0.00	0.00	0.0
prod3	4		400.00	400.00	0.0
prod3	5		600.00	500.00	100.0
prod3	6		0.00	50.00	50.0
prod4	1		300.00	300.00	0.0
prod4	2		0.00	0.00	0.0
prod4	3		0.00	0.00	0.0
prod4	4		500.00	500.00	0.0
prod4	5		100.00	100.00	0.0
prod4	6		350.00	300.00	50.0
prod5	1		800.00	800.00	0.0
prod5	2		500.00	400.00	100.0
prod5	3		0.00	100.00	0.0
prod5	4		200.00	200.00	0.0
prod5	5		1100.00	1000.00	100.0
prod5	6		0.00	50.00	50.0
prod6	1		200.00	200.00	0.0
prod6	2		300.00	300.00	0.0
prod6	3		400.00	400.00	0.0
prod6	4		0.00	0.00	0.0
prod6	5		300.00	300.00	0.0
prod6	6		550.00	500.00	50.0
prod7	1		0.00	0.00	0.0
prod7	2		250.00	150.00	100.0
prod7	3		0.00	100.00	0.0
prod7	4		100.00	100.00	0.0
prod7	5		100.00	0.00	100.0
prod7	6		0.00	50.00	50.0

You can use the `.dual` constraint suffix to access the optimal dual variables returned by the solver:

```
print Machine_hours_con.dual;
create data sol_data1 from [product period] Make Sell Store;
quit;
```

These values, shown in Figure 3.3, suggest the change in optimal objective value if the factory acquires an additional machine in that period. For this test instance, it turns out that the positive dual variables all correspond to machines that are down.

Figure 3.3 Optimal Dual Variables

	Machine_hours_con.DUAL					
	1	2	3	4	5	6
borer	0.0000	0.0000	200.0000	0.0000	0.0000	0.0000
grinder	8.5714	0.0000	0.0000	0.0000	0.0000	0.0000
hdrill	0.0000	0.6250	0.0000	0.0000	0.0000	0.0000
planer	0.0000	0.0000	0.0000	0.0000	0.0000	800.0000
vdrill	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- numeric and string index sets
- reading dense two-dimensional data
- reading sparse two-dimensional data
- INIT option
- bounds in the VAR statement
- FIX statement
- IMPVAR statement
- MAX aggregation operator
- IF-THEN/ELSE expression
- `.dual` constraint suffix

Chapter 4

Factory Planning 2: When Should Machines Be Down for Maintenance

Contents

Problem Statement	41
Mathematical Programming Formulation	41
Input Data	42
PROC OPTMODEL Statements and Output	43
Features Demonstrated	47

Problem Statement

Instead of stipulating when each machine is down for maintenance in the factory planning problem, it is desired to find the best month for each machine to be down.¹

Each machine must be down for maintenance in one month of the six apart from the grinding machines, only two of which need be down in any six months.

Extend the model to allow it to make these extra decisions. How much is the extra flexibility of allowing down times to be chosen worth?

Mathematical Programming Formulation

This formulation builds on the formulation used in [Chapter 3](#). This section includes only the new elements of the formulation.

Parameters

Table 4.1 shows the additional parameters used in this example.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, p. 234).

Table 4.1 Parameters

Parameter Name	Interpretation
<i>num_machines_needing_maintenance[machine_type]</i>	For each machine type, the number of machines that need maintenance

Variables

Table 4.2 shows the additional variables used in this example.

Table 4.2 Variables

Variable Name	Interpretation
NumMachinesDown[machine_type,period]	For each machine type, the number of machines down per period

Objective

The objective is the same as in Chapter 3.

Constraints

The following additional constraints are used in this example:

- for machine_type \in MACHINE_TYPES and period \in PERIODS,

$$\sum_{\text{product} \in \text{PRODUCTS}} \text{production_time}[\text{product}, \text{machine_type}] \cdot \text{Make}[\text{product}, \text{period}] \leq \text{num_hours_per_period} \cdot (\text{num_machines}[\text{machine_type}] - \text{NumMachinesDown}[\text{machine_type}, \text{period}])$$

- for machine_type \in MACHINE_TYPES,

$$\sum_{\text{period} \in \text{PERIODS}} \text{NumMachinesDown}[\text{machine_type}, \text{period}] = \text{num_machines_needing_maintenance}[\text{machine_type}]$$

Input Data

Ignore machine_type_period_data from Chapter 3, and replace machine_type_data as follows:

```

data machine_type_data;
    input machine_type $ num_machines num_machines_needing_maintenance;
    datalines;
grinder 4 2
vdrill 2 2
hdrill 3 3
borer 1 1
planer 1 1
;

```

PROC OPTMODEL Statements and Output

For completeness, all statements are shown. Statements that are new or changed from [Chapter 3](#) are indicated.

```

proc optmodel;
    set <str> PRODUCTS;
    num profit {PRODUCTS};
    read data product_data into PRODUCTS=[product] profit;

    set PERIODS;
    num demand {PRODUCTS, PERIODS};
    read data demand_data into PERIODS=[_N_]
        {product in PRODUCTS} <demand[product,_N_]=col(product)>;

    set <str> MACHINE_TYPES;
    num num_machines {MACHINE_TYPES};

```

The following statements declare and populate the *num_machines_needing_maintenance* parameter:

```

num num_machines_needing_maintenance {MACHINE_TYPES};
read data machine_type_data into MACHINE_TYPES=[machine_type]
    num_machines num_machines_needing_maintenance;

```

The following statements are the same as in [Chapter 3](#):

```

num production_time {PRODUCTS, MACHINE_TYPES};
read data machine_type_product_data into [machine_type]
    {product in PRODUCTS} <production_time[product,machine_type]=col(product)>;

var Make {PRODUCTS, PERIODS} >= 0;
var Sell {product in PRODUCTS, period in PERIODS} >= 0 <= demand[product,period];

num last_period = max {period in PERIODS} period;
var Store {PRODUCTS, PERIODS} >= 0 <= &store_ub;
for {product in PRODUCTS}
    fix Store[product,last_period] = &final_storage;

impvar StorageCost =
    sum {product in PRODUCTS, period in PERIODS}

```

```

        &storage_cost_per_unit * Store[product,period];
max TotalProfit =
    sum {product in PRODUCTS, period in PERIODS}
        profit[product] * Sell[product,period]
    - StorageCost;

```

Most of the remaining statements are new or modified from [Chapter 3](#). The INTEGER option in the following VAR statement declares NumMachinesDown to be an integer variable:

```

var NumMachinesDown {MACHINE_TYPES, PERIODS} >= 0 integer;

con Machine_hours_con {machine_type in MACHINE_TYPES, period in PERIODS}:
    sum {product in PRODUCTS}
        production_time[product,machine_type] * Make[product,period]
<= &num_hours_per_period *
    (num_machines[machine_type] - NumMachinesDown[machine_type,period]);

con Maintenance_con {machine_type in MACHINE_TYPES}:
    sum {period in PERIODS} NumMachinesDown[machine_type,period]
    = num_machines_needing_maintenance[machine_type];

con Flow_balance_con {product in PRODUCTS, period in PERIODS}:
    (if period - 1 in PERIODS then Store[product,period-1] else 0)
    + Make[product,period]
    = Sell[product,period] + Store[product,period];

```

Because the problem contains integer variables, the SOLVE statement automatically invokes the MILP solver:

```

solve;
print Make Sell Store;
print NumMachinesDown;
create data sol_data1 from [product period] Make Sell Store;
create data sol_data2 from [machine_type period] NumMachinesDown;
quit;

```

The solver determines when machines should be down and obtains a total profit of £108,855, as shown in [Figure 4.1](#). This objective value represents an increase of £15,140 from the optimal objective in [Chapter 3](#).

Figure 4.1 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	156
Bounded Above	0
Bounded Below	72
Bounded Below and Above	71
Free	0
Fixed	13
Binary	0
Integer	30
Number of Constraints	77
Linear LE (\leq)	30
Linear EQ ($=$)	47
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	341
Solution Summary	
Solver	MILP
Objective Function	TotalProfit
Solution Status	Optimal
Objective Value	108855
Iterations	404
Best Bound	.
Nodes	1
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	1.915623E-11
Bound Infeasibility	3.538526E-10
Integer Infeasibility	2.498779E-13

Figure 4.1 continued

[1]	[2]	Make	Sell	Store		
prod1	1	500	500	0		
prod1	2	600	600	-0		
prod1	3	400	300	100		
prod1	4	0	100	0		
prod1	5	0	0	0		
prod1	6	550	500	50		
prod2	1	1000	1000	0		
prod2	2	500	500	-0		
prod2	3	700	600	100		
prod2	4	0	100	0		
prod2	5	100	100	0		
prod2	6	550	500	50		
prod3	1	300	300	0		
prod3	2	200	200	0		
prod3	3	100	0	100		
prod3	4	0	100	0		
prod3	5	500	500	0		
prod3	6	150	100	50		
prod4	1	300	300	0		
prod4	2	0	0	0		
prod4	3	100	0	100		
prod4	4	0	100	0		
prod4	5	100	100	0		
prod4	6	350	300	50		
prod5	1	800	800	0		
prod5	2	400	400	-0		
prod5	3	600	500	100		
prod5	4	0	100	0		
prod5	5	1000	1000	-0		
prod5	6	1150	1100	50		
prod6	1	200	200	0		
prod6	2	300	300	0		
prod6	3	400	400	0		
prod6	4	0	0	0		
prod6	5	300	300	0		
prod6	6	550	500	50		
prod7	1	100	100	0		
prod7	2	150	150	0		
prod7	3	200	100	100		
prod7	4	0	100	0		
prod7	5	0	0	0		
prod7	6	110	60	50		
NumMachinesDown						
	1	2	3	4	5	6
borer	0	0	-0	1	0	-0
grinder	0	0	0	2	0	0
hdrill	1	0	2	0	0	0
planer	0	-0	0	1	0	0
vdrill	0	1	0	1	0	0

As expected, the optimal numbers of machines down differ from the *num_machines_down_per_period* parameter values in [Chapter 3](#).

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- numeric and string index sets
- reading dense two-dimensional data
- bounds in the VAR statement
- FIX statement
- IMPVAR statement
- MAX aggregation operator
- IF-THEN/ELSE expression
- multiple input and output data sets
- INTEGER option

Chapter 5

Manpower Planning: How to Recruit, Retrain, Make Redundant, or Overman

Contents

Problem Statement	49
Mathematical Programming Formulation	51
Input Data	54
PROC OPTMODEL Statements and Output	54
Features Demonstrated	60

Problem Statement

A company is undergoing a number of changes which will affect its manpower requirements in future years.¹ Owing to the installation of new machinery, fewer unskilled but more skilled and semi-skilled workers will be required. In addition to this a downturn in trade is expected in the next year which will reduce the need for workers in all categories. The estimated manpower requirements for the next three years are as follows:

	Unskilled	Semi-skilled	Skilled
Current strength	2000	1500	1000
Year 1	1000	1400	1000
Year 2	500	2000	1500
Year 3	0	2500	2000

The company wishes to decide its policy with regard to the following over the next three years:

- (1) Recruitment
- (2) Retraining
- (3) Redundancy
- (4) Short-time working.

There is a natural wastage of labour. A fairly large number of workers leave during their first year. After this the rate is much smaller. Taking this into account, the wastage rates can be taken as below:

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 234–236).

	Unskilled	Semi-skilled	Skilled
Less than one year's service	25%	20%	10%
More than one year's service	10%	5%	5%

There has been no recent recruitment and all workers in the current labour force have been employed for more than one year.

Recruitment

It is possible to recruit a limited number of workers from outside. In any one year the numbers which can be recruited in each category are:

Unskilled	Semi-skilled	Skilled
500	800	500

Retraining

It is possible to retrain up to 200 unskilled workers per year to make them semi-skilled. This costs £400 per worker. The retraining of semi-skilled workers to make them skilled is limited to no more than one quarter of the skilled labour at the time as some training is done on the job. To retrain a semi-skilled worker in this way costs £500.

Downgrading of workers to a lower skill is possible but 50% of such workers leave, although it costs the company nothing. (This wastage is additional to the 'natural wastage' described above.)

Redundancy

The redundancy payment to an unskilled worker is £200 and to a semi-skilled or skilled worker £500.

Overmanning

It is possible to employ up to 150 more workers over the whole company than are needed but the extra costs per employee per year are:

Unskilled	Semi-skilled	Skilled
£1500	£2000	£3000

Short-time Working

Up to 50 workers in each category of skill can be put on short-time working. The cost of this (per employee per year) is:

Unskilled	Semi-skilled	Skilled
£500	£400	£400

An employee on short-time working meets the production requirements of half an employee.

The company's declared objective is to minimize redundancy. How should they operate in order to do this?

If their policy were to minimize costs, how much extra would this save? Deduce the cost of saving each type of job each year.

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{worker} \in \text{WORKERS}$
- $\text{period} \in \text{PERIODS0}$
- $\text{period} \in \text{PERIODS} = \text{PERIODS0} \setminus \{0\}$
- $(i, j) \in \text{RETRAIN_PAIRS}$: worker i retrained as worker j
- $(i, j) \in \text{DOWNGRADE_PAIRS}$: worker i downgraded to worker j

Parameters

Table 5.1 shows the parameters used in this example.

Table 5.1 Parameters

Parameter Name	Interpretation
<i>waste_new[worker]</i>	Fraction of workers who leave during their first year
<i>waste_old[worker]</i>	Fraction of workers who leave after their first year
<i>recruit_ub[worker]</i>	Upper bound on number of workers who are recruited from outside
<i>redundancy_cost[worker]</i>	Cost per worker made redundant
<i>overmanning_cost[worker]</i>	Cost per excess worker
<i>shorttime_ub[worker]</i>	Upper bound on number of short-time workers
<i>shorttime_cost[worker]</i>	Cost per short-time worker
<i>demand[worker,period]</i>	Manpower requirements
<i>retrain_ub[i,j]</i>	Upper bound on number of workers who can be retrained from <i>i</i> to <i>j</i> per year
<i>retrain_cost[i,j]</i>	Cost to retrain worker <i>i</i> as worker <i>j</i>
<i>semiskill_retrain_frac_ub</i>	Upper bound on fraction of semi-skilled workers who can be retrained to skilled
<i>downgrade_leave_frac</i>	Fraction of downgraded workers who leave
<i>overmanning_ub</i>	Upper bound on number of excess workers
<i>shorttime_frac</i>	Fraction of production requirements of a full-time employee that is met by each short-time employee

Variables

Table 5.2 shows the variables used in this example.

Table 5.2 Variables

Variable Name	Interpretation
<i>NumWorkers[worker,period]</i>	Number of workers per period
<i>NumRecruits[worker,period]</i>	Number of recruited workers per period
<i>NumRedundant[worker,period]</i>	Number of workers made redundant per period
<i>NumShortTime[worker,period]</i>	Number of short-time workers per period
<i>NumExcess[worker,period]</i>	Number of excess workers per period
<i>NumRetrain[i,j,period]</i>	Number of workers retrained from <i>i</i> to <i>j</i> per period
<i>NumDowngrade[i,j,period]</i>	Number of workers downgraded from <i>i</i> to <i>j</i> per period

Objectives

One objective is to minimize the following function:

$$\text{Redundancy} = \sum_{\text{worker} \in \text{WORKERS}} \sum_{\text{period} \in \text{PERIODS}} \text{NumRedundant}[\text{worker}, \text{period}]$$

A second objective is to minimize the following function:

$$\begin{aligned}
 \text{Cost} = & \sum \sum \text{redundancy_cost}[\text{worker}] \cdot \text{NumRedundant}[\text{worker}, \text{period}] \\
 & + \sum \sum \text{shorttime_cost}[\text{worker}] \cdot \text{NumShorttime}[\text{worker}, \text{period}] \\
 & + \sum \sum \text{overmanning_cost}[\text{worker}] \cdot \text{NumExcess}[\text{worker}, \text{period}] \\
 & + \sum_{(i,j) \in \text{RETRAIN_PAIRS}} \sum_{\text{period} \in \text{PERIODS}} \text{retrain_cost}[i,j] \cdot \text{NumRetrain}[i,j, \text{period}]
 \end{aligned}$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for worker \in WORKERS and period \in PERIODS,

$$\begin{aligned}
 & \text{NumWorkers}[\text{worker}, \text{period}] \\
 & - (1 - \text{shorttime_frac}) \cdot \text{NumShortTime}[\text{worker}, \text{period}] \\
 & - \text{NumExcess}[\text{worker}, \text{period}] \\
 & = \text{demand}[\text{worker}, \text{period}]
 \end{aligned}$$
- for worker \in WORKERS and period \in PERIODS,

$$\begin{aligned}
 & \text{NumWorkers}[\text{worker}, \text{period}] \\
 & = (1 - \text{waste_old}[\text{worker}]) \cdot \text{NumWorkers}[\text{worker}, \text{period} - 1] \\
 & + (1 - \text{waste_new}[\text{worker}]) \cdot \text{NumRecruits}[\text{worker}, \text{period}] \\
 & + (1 - \text{waste_old}[\text{worker}]) \cdot \sum_{(i, \text{worker}) \in \text{RETRAIN_PAIRS}} \text{NumRetrain}[i, \text{worker}, \text{period}] \\
 & + (1 - \text{downgrade_leave_frac}) \cdot \sum_{(i, \text{worker}) \in \text{DOWNGRADE_PAIRS}} \text{NumDowngrade}[i, \text{worker}, \text{period}] \\
 & - \sum_{(\text{worker}, j) \in \text{RETRAIN_PAIRS}} \text{NumRetrain}[\text{worker}, j, \text{period}] \\
 & - \sum_{(\text{worker}, j) \in \text{DOWNGRADE_PAIRS}} \text{NumDowngrade}[\text{worker}, j, \text{period}] \\
 & - \text{NumRedundant}[\text{worker}, \text{period}]
 \end{aligned}$$
- for period \in PERIODS,

$$\text{NumRetrain}[\text{'semiskilled'}, \text{'skilled'}, \text{period}] \leq \text{semiskill_retrain_frac_ub} \cdot \text{NumWorkers}[\text{'skilled'}, \text{period}]$$
- for period \in PERIODS,

$$\sum_{\text{worker} \in \text{WORKERS}} \text{NumExcess}[\text{worker}, \text{period}] \leq \text{overmanning_ub}$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data demand_data;
    input period unskilled semiskilled skilled;
    datalines;
0 2000 1500 1000
1 1000 1400 1000
2 500 2000 1500
3 0 2500 2000
;

data worker_data;
    input worker $12. waste_new waste_old recruit_ub redundancy_cost overmanning_cost
        shorttime_ub shorttime_cost;
    datalines;
unskilled 0.25 0.10 500 200 1500 50 500
semiskilled 0.20 0.05 800 500 2000 50 400
skilled 0.10 0.05 500 500 3000 50 400
;

data retrain_data;
    input worker1 $12. worker2 $12. retrain_ub retrain_cost;
    datalines;
unskilled semiskilled 200 400
semiskilled skilled . 500
;

data downgrade_data;
    input worker1 $12. worker2 $12.;
    datalines;
semiskilled unskilled
skilled semiskilled
skilled unskilled
;

%let semiskill_retrain_frac_ub = 0.25;
%let downgrade_leave_frac = 0.5;
%let overmanning_ub = 150;
%let shorttime_frac = 0.5;
```

PROC OPTMODEL Statements and Output

The first several index sets are one-dimensional, as in all the previous examples:


```

proc optmodel;
  set <str> WORKERS;
  num waste_new {WORKERS};
  num waste_old {WORKERS};
  num recruit_ub {WORKERS};
  num redundancy_cost {WORKERS};
  num overmanning_cost {WORKERS};
  num shorttime_ub {WORKERS};
  num shorttime_cost {WORKERS};
  read data worker_data into WORKERS=[worker]
    waste_new waste_old recruit_ub redundancy_cost overmanning_cost
    shorttime_ub shorttime_cost;

  set PERIODS0;
  num demand {WORKERS, PERIODS0};
  read data demand_data into PERIODS0=[period]
    {worker in WORKERS} <demand[worker,period]=col(worker)>;

  var NumWorkers {WORKERS, PERIODS0} >= 0;
  for {worker in WORKERS} fix NumWorkers[worker,0] = demand[worker,0];

  set PERIODS = PERIODS0 diff {0};
  var NumRecruits {worker in WORKERS, PERIODS} >= 0 <= recruit_ub[worker];
  var NumRedundant {WORKERS, PERIODS} >= 0;
  var NumShortTime {worker in WORKERS, PERIODS} >= 0 <= shorttime_ub[worker];
  var NumExcess {WORKERS, PERIODS} >= 0;

```

Both RETRAIN_PAIRS and DOWNGRADE_PAIRS are two-dimensional index sets, declared by using the optional <STR,STR> specification in the SET statement so that these sets contain pairs of strings. In general, a set can consist of tuples of any length and any combination of NUM and STR scalar-types.

```

  set <str,str> RETRAIN_PAIRS;
  num retrain_ub {RETRAIN_PAIRS};
  num retrain_cost {RETRAIN_PAIRS};
  read data retrain_data into RETRAIN_PAIRS=[worker1 worker2] retrain_ub retrain_cost;

  var NumRetrain {RETRAIN_PAIRS, PERIODS} >= 0;
  for {<i,j> in RETRAIN_PAIRS: retrain_ub[i,j] ne .}
    for {period in PERIODS} NumRetrain[i,j,period].ub = retrain_ub[i,j];

  set <str,str> DOWNGRADE_PAIRS;
  read data downgrade_data into DOWNGRADE_PAIRS=[worker1 worker2];
  var NumDowngrade {DOWNGRADE_PAIRS, PERIODS} >= 0;

  con Demand_con {worker in WORKERS, period in PERIODS}:
    NumWorkers[worker,period]
    - (1 - &shorttime_frac) * NumShortTime[worker,period]
    - NumExcess[worker,period]
    = demand[worker,period];

```

The following Flow_balance_con constraint uses an implicit slice to express a few of the summations compactly:

```

con Flow_balance_con {worker in WORKERS, period in PERIODS}:
    NumWorkers[worker,period]
= (1 - waste_old[worker]) * NumWorkers[worker,period-1]
+ (1 - waste_new[worker]) * NumRecruits[worker,period]
+ (1 - waste_old[worker]) *
    sum {<i,(worker)> in RETRAIN_PAIRS} NumRetrain[i,worker,period]
+ (1 - &downgrade_leave_frac) *
    sum {<i,(worker)> in DOWNGRADE_PAIRS} NumDowngrade[i,worker,period]
- sum {<(worker),j> in RETRAIN_PAIRS} NumRetrain[worker,j,period]
- sum {<(worker),j> in DOWNGRADE_PAIRS} NumDowngrade[worker,j,period]
- NumRedundant[worker,period];

```

For example,

```
<i,(worker)> in RETRAIN_PAIRS
```

is equivalent to

```
i in slice(<*,worker>,RETRAIN_PAIRS)
```

which is equivalent to

```
i in WORKERS: <i,worker> in RETRAIN_PAIRS
```

The remaining two constraints are straightforward:

```

con Semiskill_retrain_con {period in PERIODS}:
    NumRetrain['semiskilled','skilled',period]
<= &semiskill_retrain_frac_ub * NumWorkers['skilled',period];

con Overmanning_con {period in PERIODS}:
    sum {worker in WORKERS} NumExcess[worker,period] <= &overmanning_ub;

```

This example uses two objectives, Redundancy and Cost, declared in the following MIN statements:

```

min Redundancy =
    sum {worker in WORKERS, period in PERIODS} NumRedundant[worker,period];
min Cost =
    sum {worker in WORKERS, period in PERIODS} (
        redundancy_cost[worker] * NumRedundant[worker,period]
        + shorttime_cost[worker] * NumShorttime[worker,period]
        + overmanning_cost[worker] * NumExcess[worker,period])
    + sum {<i,j> in RETRAIN_PAIRS, period in PERIODS}
        retrain_cost[i,j] * NumRetrain[i,j,period];

```

The LP solver is called twice, and each SOLVE statement includes the OBJ option to specify which objective to optimize. The first PRINT statement after each SOLVE statement reports the values of both objectives even though only one objective is optimized at a time:

```

solve obj Redundancy;
print Redundancy Cost;
print NumWorkers NumRecruits NumRedundant NumShortTime NumExcess;
print NumRetrain;
print NumDowngrade;
create data sol_data1 from [worker period]

```

```

    NumWorkers NumRecruits NumRedundant NumShortTime NumExcess;
create data sol_data2 from [worker1 worker2 period] NumRetrain NumDowngrade;

solve obj Cost;
print Redundancy Cost;
print NumWorkers NumRecruits NumRedundant NumShortTime NumExcess;
print NumRetrain;
print NumDowngrade;
create data sol_data3 from [worker period]
    NumWorkers NumRecruits NumRedundant NumShortTime NumExcess;
create data sol_data4 from [worker1 worker2 period] NumRetrain NumDowngrade;
quit;

```

Figure 5.1 shows the output that results from the first SOLVE statement.

Figure 5.1 Output from First SOLVE Statement, Minimizing Redundancy

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	Redundancy
Objective Type	Linear
Number of Variables	63
Bounded Above	0
Bounded Below	39
Bounded Below and Above	21
Free	0
Fixed	3
Number of Constraints	24
Linear LE (<=)	6
Linear EQ (=)	18
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	108
Solution Summary	
Solver	Dual Simplex
Objective Function	Redundancy
Solution Status	Optimal
Objective Value	841.796875
Iterations	9
Primal Infeasibility	2.842171E-14
Dual Infeasibility	0
Bound Infeasibility	0
Redundancy	Cost
841.8	1441390

Figure 5.1 *continued*

[1]	[2]	Num Workers	Num Recruits	Num Redundant	Num Short Time	Num Excess
semiskilled	0	1500				
semiskilled	1	1443	0.00	0.00	50	17.969
semiskilled	2	2000	649.30	0.00	0	0.000
semiskilled	3	2500	676.97	0.00	0	0.000
skilled	0	1000				
skilled	1	1025	0.00	0.00	50	0.000
skilled	2	1500	500.00	0.00	0	0.000
skilled	3	2000	500.00	0.00	0	0.000
unskilled	0	2000				
unskilled	1	1157	0.00	442.97	50	132.031
unskilled	2	675	0.00	166.33	50	150.000
unskilled	3	175	0.00	232.50	50	150.000

[1]	[2]	[3]	Num Retrain
semiskilled	skilled	1	256.250
semiskilled	skilled	2	80.263
semiskilled	skilled	3	131.579
unskilled	semiskilled	1	200.000
unskilled	semiskilled	2	200.000
unskilled	semiskilled	3	200.000

[1]	[2]	[3]	Num Downgrade
semiskilled	unskilled	1	0.00
semiskilled	unskilled	2	0.00
semiskilled	unskilled	3	0.00
skilled	semiskilled	1	168.44
skilled	semiskilled	2	0.00
skilled	semiskilled	3	0.00
skilled	unskilled	1	0.00
skilled	unskilled	2	0.00
skilled	unskilled	3	0.00

Figure 5.2 shows the output that results from the second SOLVE statement.

Figure 5.2 Output from Second SOLVE Statement, Minimizing Cost

Problem Summary						
Objective Sense		Minimization				
Objective Function		Cost				
Objective Type		Linear				
Number of Variables		63				
Bounded Above		0				
Bounded Below		39				
Bounded Below and Above		21				
Free		0				
Fixed		3				
Number of Constraints		24				
Linear LE (\leq)		6				
Linear EQ ($=$)		18				
Linear GE (\geq)		0				
Linear Range		0				
Constraint Coefficients		108				
Solution Summary						
Solver		Dual Simplex				
Objective Function		Cost				
Solution Status		Optimal				
Objective Value		498677.28532				
Iterations		6				
Primal Infeasibility		2.842171E-14				
Dual Infeasibility		0				
Bound Infeasibility		0				
Redundancy		Cost				
1423.7		498677				
[1]	[2]	Num Workers	Num Recruits	Num Redundant	Num Short Time	Num Excess
semiskilled	0	1500				
semiskilled	1	1400	0.000	0.00	0	0
semiskilled	2	2000	800.000	0.00	0	0
semiskilled	3	2500	800.000	0.00	0	0
skilled	0	1000				
skilled	1	1000	55.556	0.00	0	0
skilled	2	1500	500.000	0.00	0	0
skilled	3	2000	500.000	0.00	0	0
unskilled	0	2000				
unskilled	1	1000	0.000	812.50	0	0
unskilled	2	500	0.000	257.62	0	0
unskilled	3	0	0.000	353.60	0	0

Figure 5.2 *continued*

[1]	[2]	[3]	Num Retrain
semiskilled	skilled	1	0.000
semiskilled	skilled	2	105.263
semiskilled	skilled	3	131.579
unskilled	semiskilled	1	0.000
unskilled	semiskilled	2	142.382
unskilled	semiskilled	3	96.399

[1]	[2]	[3]	Num Downgrade
semiskilled	unskilled	1	25
semiskilled	unskilled	2	0
semiskilled	unskilled	3	0
skilled	semiskilled	1	0
skilled	semiskilled	2	0
skilled	semiskilled	3	0
skilled	unskilled	1	0
skilled	unskilled	2	0
skilled	unskilled	3	0

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- numeric and string index sets
- reading dense two-dimensional data
- reading sparse two-dimensional data
- sets of tuples
- bounds in the VAR statement
- `.ub` variable suffix
- FIX statement
- using a colon (:) to select members of a set
- set operator DIFF
- SLICE expression

- implicit slice
- multiple objectives and the OBJ option
- multiple input and output data sets

Chapter 6

Refinery Optimization: How to Run an Oil Refinery

Contents

Problem Statement	63
Mathematical Programming Formulation	65
Input Data	69
PROC OPTMODEL Statements and Output	71
Features Demonstrated	76

Problem Statement

An oil refinery purchases two crude oils (crude 1 and crude 2).¹ These crude oils are put through four processes: distillation, reforming, cracking, and blending, to produce petrols and fuels which are sold.

Distillation

Distillation separates each crude oil into fractions known as *light naphtha*, *medium naphtha*, *heavy naphtha*, *light oil*, *heavy oil* and *residuum* according to their boiling points. Light, medium and heavy naphthas have octane numbers of 90, 80 and 70 respectively. The fractions into which one barrel of each type of crude splits are given in the table:

	Light naphtha	Medium naphtha	Heavy naphtha	Light oil	Heavy oil	Residuum
Crude 1	0.1	0.2	0.2	0.12	0.2	0.13
Crude 2	0.15	0.25	0.18	0.08	0.19	0.12

N.B. There is a small amount of wastage in distillation.

Reforming

The naphthas can be used immediately for blending into different grades of petrol or can go through a process known as reforming. Reforming produces a product known as reformed gasoline with an octane number of 115. The yields of reformed gasoline from each barrel of the different naphthas are given below:

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 236–238).

1 barrel of light naphtha yields 0.6 barrels of reformed gasoline;
1 barrel of medium naphtha yields 0.52 barrels of reformed gasoline;
1 barrel of heavy naphtha yields 0.45 barrels of reformed gasoline.

Cracking

The oils (light and heavy) can either be used directly for blending into *jet fuel* or *fuel oil* or be put through a process known as catalytic cracking. The catalytic cracker produces *cracked oil* and *cracked gasoline*. Cracked gasoline has an octane number of 105.

1 barrel of light oil yields 0.68 barrels of cracked oil and 0.28 barrels of cracked gasoline;
1 barrel of heavy oil yields 0.75 barrels of cracked oil and 0.2 barrels of cracked gasoline.

Cracked oil is used for blending *fuel oil* and *jet fuel*; cracked gasoline is used for blending *petrol*.

Residuum can be used for either producing *lube-oil* or blending into *jet fuel* and *fuel oil*:

1 barrel of residuum yields 0.5 barrels of lube-oil.

Blending

Petrols (Motor Fuel)

There are two sorts of petrol, *regular* and *premium*, obtained by blending the naphtha, reformed gasoline and cracked gasoline. The only stipulations concerning them are that regular must have an octane number of at least 84 and that premium must have an octane number of at least 94. It is assumed that octane numbers blend linearly by volume.

Jet Fuel

The stipulation concerning jet fuel is that its vapour pressure must not exceed 1 kilogram per square centimetre. The vapour pressures for light, heavy and cracked oils and residuum are 1.0, 0.6, 1.5 and 0.05 kilograms per square centimetre respectively. It may again be assumed that vapour pressures blend linearly by volume.

Fuel Oil

To produce fuel oil, light oil, cracked oil, heavy oil and residuum must be blended in the ratio 10:4:3:1.

There are availability and capacity limitations on the quantities and processes used:

- (a) The daily availability of crude 1 is 20,000 barrels.
- (b) The daily availability of crude 2 is 30,000 barrels.
- (c) At most 45,000 barrels of crude can be distilled per day.
- (d) At most 10,000 barrels of naphtha can be reformed per day.
- (e) At most 8000 barrels of oil can be cracked per day.
- (f) The daily production of lube oil must be between 500 and 1000 barrels.
- (g) Premium motor fuel production must be at least 40% of regular motor fuel production.

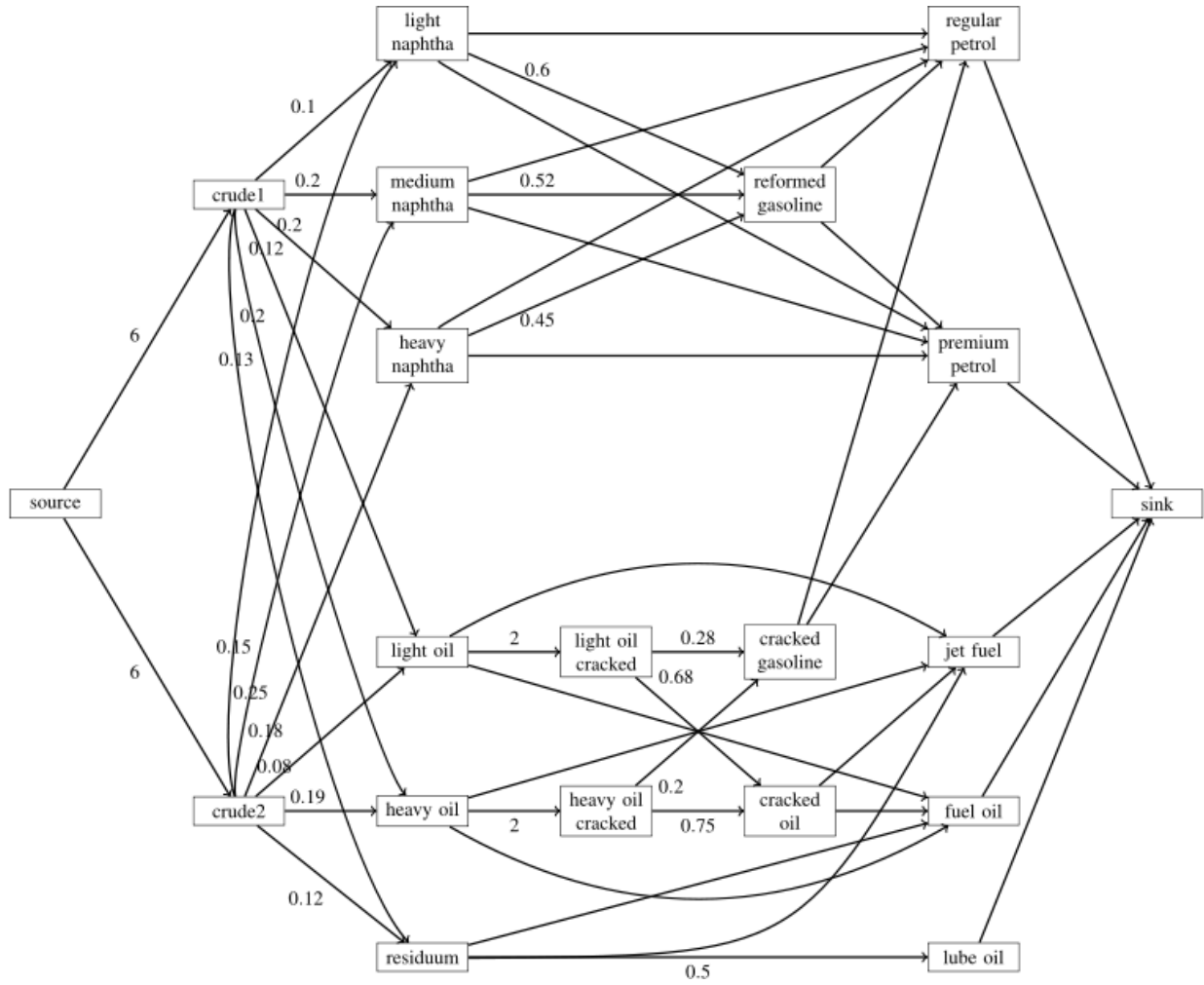
The profit contributions from the sale of the final products are (in pence per barrel)

Premium petrol	700
Regular petrol	600
Jet fuel	400
Fuel oil	350
Lube-oil	150

How should the operations of the refinery be planned in order to maximize total profit?

Mathematical Programming Formulation

The problem is represented as a generalized network flow problem with side constraints. Each node corresponds to a material, and each arc represents conversion of one material to another via one of the four processes, as shown in [Figure 6.1](#). The arc multiplier values 6 and 2, together with the Distillation and Cracking constraints specified later, are used to split the flow into equal parts at the head nodes of the corresponding arcs.

Figure 6.1 Generalized Network with Arc Multipliers

Index Sets and Their Members

The following index sets and their members are used in this example:

- $i \in \text{NODES}$
- $(i, j) \in \text{ARCS}$
- $\text{product} \in \text{FINAL_PRODUCTS}$
- $\text{crude} \in \text{CRUDES}$
- $\text{oil} \in \text{OILS}$
- $\text{oil} \in \text{CRACKED_OILS}$
- $\text{petrol} \in \text{PETROLS}$

Parameters

Table 6.1 shows the parameters used in this example.

Table 6.1 Parameters

Parameter Name	Interpretation
<i>arc_mult[arc]</i>	Arc multiplier (default value of 1)
<i>profit[product]</i>	Profit from sale of final product (in pounds per barrel)
<i>octane[i]</i>	Octane per node
<i>octane_lb[petrol]</i>	Lower bound on octane
<i>vapour_pressure[i]</i>	Vapour pressure per node
<i>fuel_oil_coefficient[i]</i>	Fuel oil blending coefficient per node
<i>sum_fuel_oil_coefficient</i>	$\sum_{(i, \text{'fuel_oil'}) \in \text{ARCS}} \text{fuel_oil_coefficient}[i]$
<i>vapour_pressure_ub</i>	Upper bound on vapour pressure
<i>crude_total_ub</i>	Upper bound on number of barrels of crude distilled per day
<i>naphtha_ub</i>	Upper bound on number of barrels of naphtha reformed per day
<i>cracked_oil_ub</i>	Upper bound on number of barrels of oil cracked per day
<i>lube_oil_lb</i>	Lower bound on number of barrels of lube oil produced per day
<i>lube_oil_ub</i>	Upper bound on number of barrels of lube oil produced per day
<i>premium_ratio</i>	Lower bound on ratio of premium motor fuel production to regular motor fuel production

Variables

Table 6.2 shows the variables used in this example.

Table 6.2 Variables

Variable Name	Interpretation
Flow[i,j]	Flow across arc (i, j)
CrudeDistilled[crude]	Barrels of crude oil distilled
OilCracked[oil]	Barrels of oil cracked

Objective

The objective is to maximize the following function:

$$\text{TotalProfit} = \sum_{i \in \text{FINAL_PRODUCTS}} \text{profit}[i] \cdot \text{Flow}[i, \text{'sink'}]$$

Constraints

The following constraints are used in this example:

- bounds on variables

- for $i \in \text{NODES} \setminus \{\text{'source'}, \text{'sink'}\}$,

$$\sum_{(i,j) \in \text{ARCS}} \text{Flow}[i,j] = \sum_{(j,i) \in \text{ARCS}} \text{arc_mult}[j,i] \cdot \text{Flow}[j,i]$$

- for $(i, j) \in \text{ARCS}$ such that $i \in \text{CRUDES}$,

$$\text{Flow}[i,j] = \text{CrudeDistilled}[i]$$

- for $(i, j) \in \text{ARCS}$ such that $i \in \text{CRACKED_OILS}$,

$$\text{Flow}[i,j] = \text{OilCracked}[i]$$

- for $\text{petrol} \in \text{PETROLS}$,

$$\frac{\sum_{(i,\text{petrol}) \in \text{ARCS}} \text{octane}[i] \cdot \text{arc_mult}[i,\text{petrol}] \cdot \text{Flow}[i,\text{petrol}]}{\sum_{(i,\text{petrol}) \in \text{ARCS}} \text{arc_mult}[i,\text{petrol}] \cdot \text{Flow}[i,\text{petrol}]} \geq \text{octane_lb}[\text{petrol}]$$

- $$\frac{\sum_{(i,\text{'jet_fuel'}) \in \text{ARCS}} \text{vapour_pressure}[i] \cdot \text{arc_mult}[i,\text{'jet_fuel'}] \cdot \text{Flow}[i,\text{'jet_fuel'}]}{\sum_{(i,\text{'jet_fuel'}) \in \text{ARCS}} \text{arc_mult}[i,\text{'jet_fuel'}] \cdot \text{Flow}[i,\text{'jet_fuel'}]} \leq \text{vapour_pressure_ub}$$

- for $(i, \text{'fuel_oil'}) \in \text{ARCS}$,

$$\frac{\text{Flow}[i,\text{'fuel_oil'}]}{\sum_{(j,\text{'fuel_oil'}) \in \text{ARCS}} \text{Flow}[j,\text{'fuel_oil'}]} = \frac{\text{fuel_oil_coefficient}[i]}{\text{sum_fuel_oil_coefficient}}$$

- $$\sum_{i \in \text{CRUDES}} \text{CrudeDistilled}[i] \leq \text{crude_total_ub}$$

- $$\sum_{\substack{(i,\text{'reformed_gasoline'}) \in \text{ARCS}: \\ \text{index}(i,\text{'naphtha'}) > 0}} \text{Flow}[i,\text{'reformed_gasoline'}] \leq \text{naphtha_ub}$$

- $$\sum_{(i,\text{'cracked_oil'}) \in \text{ARCS}} \text{Flow}[i,\text{'cracked_oil'}] \leq \text{cracked_oil_ub}$$

- $\text{lube_oil_lb} \leq \text{Flow}[\text{'lube_oil'}, \text{'sink'}] \leq \text{lube_oil_ub}$

- $$\frac{\sum_{(\text{'premium_petrol'}, j) \in \text{ARCS}} \text{Flow}[\text{'premium_petrol'}, j]}{\sum_{(\text{'regular_petrol'}, j) \in \text{ARCS}} \text{Flow}[\text{'regular_petrol'}, j]} \geq \text{premium_ratio}$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```

data crude_data;
    input crude $ crude_ub;
    datalines;
crude1 20000
crude2 30000
;

data arc_data;
    input i $18. j $18. multiplier;
    datalines;
source      crude1      6
source      crude2      6
crude1      light_naphtha 0.1
crude1      medium_naphtha 0.2
crude1      heavy_naphtha 0.2
crude1      light_oil     0.12
crude1      heavy_oil     0.2
crude1      residuum      0.13
crude2      light_naphtha 0.15
crude2      medium_naphtha 0.25
crude2      heavy_naphtha 0.18
crude2      light_oil     0.08
crude2      heavy_oil     0.19
crude2      residuum      0.12
light_naphtha regular_petrol .
light_naphtha premium_petrol .
medium_naphtha regular_petrol .
medium_naphtha premium_petrol .
heavy_naphtha regular_petrol .
heavy_naphtha premium_petrol .
light_naphtha reformed_gasoline 0.6
medium_naphtha reformed_gasoline 0.52
heavy_naphtha reformed_gasoline 0.45
light_oil      jet_fuel     .
light_oil      fuel_oil     .
heavy_oil      jet_fuel     .
heavy_oil      fuel_oil     .
light_oil      light_oil_cracked 2
light_oil_cracked cracked_oil 0.68
light_oil_cracked cracked_gasoline 0.28
heavy_oil      heavy_oil_cracked 2
heavy_oil_cracked cracked_oil 0.75
heavy_oil_cracked cracked_gasoline 0.2
cracked_oil     jet_fuel     .
cracked_oil     fuel_oil     .
reformed_gasoline regular_petrol .
reformed_gasoline premium_petrol .
cracked_gasoline regular_petrol .

```

```

cracked_gasoline premium_petrol .
residuum         lube_oil       0.5
residuum         jet_fuel       .
residuum         fuel_oil       .
;

data octane_data;
    input i $18. octane;
    datalines;
light_naphtha    90
medium_naphtha   80
heavy_naphtha    70
reformed_gasoline 115
cracked_gasoline 105
;

data petrol_data;
    input petrol $15. octane_lb;
    datalines;
regular_petrol  84
premium_petrol  94
;

data vapour_pressure_data;
    input oil $12. vapour_pressure;
    datalines;
light_oil      1.0
heavy_oil      0.6
cracked_oil    1.5
residuum       0.05
;

data fuel_oil_ratio_data;
    input oil $12. coefficient;
    datalines;
light_oil      10
cracked_oil    4
heavy_oil      3
residuum       1
;

data final_product_data;
    input product $15. profit;
    datalines;
premium_petrol 700
regular_petrol 600
jet_fuel       400
fuel_oil       350
lube_oil       150
;

%let vapour_pressure_ub = 1;
%let crude_total_ub = 45000;
%let naphtha_ub = 10000;

```



```
%let cracked_oil_ub = 8000;
%let lube_oil_lb = 500;
%let lube_oil_ub = 1000;
%let premium_ratio = 0.40;
```

PROC OPTMODEL Statements and Output

The NOMISS option in the first READ DATA statement ensures that only nonmissing values of the variable multiplier in the arc_data data set populate the *arc_mult* parameter. Because *arc_mult* is declared with an initial value of 1, parameters with no value default to 1.

```
proc optmodel;
  set <str,str> ARCS;
  num arc_mult {ARCS} init 1;
  read data arc_data nomiss into ARCS=[i j] arc_mult=multiplier;
  var Flow {ARCS} >= 0;

  set <str> FINAL_PRODUCTS;
  num profit {FINAL_PRODUCTS};
  read data final_product_data into FINAL_PRODUCTS=[product] profit;
```

The following FOR loop converts *profit* from pence per barrel to pounds per barrel, without altering the input data set:

```
for {product in FINAL_PRODUCTS} profit[product] = profit[product] / 100;
```

Most arcs appear in the arc_data data set, but the following assignment statement uses the set operators UNION and CROSS to augment the ARCS set with an arc from each final product to the sink node:

```
ARCS = ARCS union (FINAL_PRODUCTS cross {'sink'});

set NODES = union {<i,j> in ARCS} {i,j};

max TotalProfit = sum {i in FINAL_PRODUCTS} profit[i] * Flow[i,'sink'];

con Flow_balance {i in NODES diff {'source','sink'}}:
  sum {<(i),j> in ARCS} Flow[i,j] = sum {<j,(i)> in ARCS} arc_mult[j,i] * Flow[j,i];

set <str> CRUDES;
var CrudeDistilled {CRUDES} >= 0;
```

Because the decision variable CrudeDistilled automatically contains the .ub suffix, you can populate *CrudeDistilled.ub* directly by using the following READ DATA statement, without having to declare an additional parameter to store this upper bound:

```
read data crude_data into CRUDES=[crude] CrudeDistilled.ub=crude_ub;
con Distillation {<i,j> in ARCS: i in CRUDES}:
  Flow[i,j] = CrudeDistilled[i];
```

The SETOF operator, used together with the concatenation operator (||) in the following statements, enables you to create one index set from another:

```

set OILS = {'light_oil', 'heavy_oil'};
set CRACKED_OILS = setof {i in OILS} i || '_cracked';
var OilCracked {CRACKED_OILS} >= 0;
con Cracking {<i,j> in ARCS: i in CRACKED_OILS}:
    Flow[i,j] = OilCracked[i];

num octane {NODES} init .;
read data octane_data nomiss into [i] octane;

set <str> PETROLS;
num octane_lb {PETROLS};
read data petrol_data into PETROLS=[petrol] octane_lb;

num vapour_pressure {NODES} init .;
read data vapour_pressure_data nomiss into [oil] vapour_pressure;

```

As expressed on page 68, both octane numbers and vapour pressures of the blended fuels are ratios of linear functions of the decision variables. To increase algorithmic performance and reliability, the following two CON statements linearize the nonlinear ratio constraints by clearing the denominators:

```

con Blending_petrol {petrol in PETROLS}:
    sum {<i, (petrol)> in ARCS} octane[i] * arc_mult[i,petrol] * Flow[i,petrol]
    >= octane_lb[petrol] * sum {<i, (petrol)> in ARCS} arc_mult[i,petrol] * Flow[i,petrol];

con Blending_jet_fuel:
    sum {<i, 'jet_fuel'> in ARCS}
        vapour_pressure[i] * arc_mult[i, 'jet_fuel'] * Flow[i, 'jet_fuel']
    <= &vapour_pressure_ub *
        sum {<i, 'jet_fuel'> in ARCS} arc_mult[i, 'jet_fuel'] * Flow[i, 'jet_fuel'];

```

Similarly, the following CON statement linearizes the fuel oil ratio constraints by clearing the denominators:

```

num fuel_oil_coefficient {NODES} init 0;
read data fuel_oil_ratio_data nomiss into [oil] fuel_oil_coefficient=coefficient;
num sum_fuel_oil_coefficient = sum {<i, 'fuel_oil'> in ARCS} fuel_oil_coefficient[i];
con Blending_fuel_oil {<i, 'fuel_oil'> in ARCS}:
    sum_fuel_oil_coefficient * Flow[i, 'fuel_oil']
    = fuel_oil_coefficient[i] * sum {<j, 'fuel_oil'> in ARCS} Flow[j, 'fuel_oil'];

con Crude_total_ub_con:
    sum {i in CRUDES} CrudeDistilled[i] <= &crude_total_ub;

```

The following CON statement uses the SAS function INDEX together with the colon operator (:) to select the subset of arcs whose tail node contains 'naphtha' in its name:

```

con Naphtha_ub_con:
    sum {<i, 'reformed_gasoline'> in ARCS: index(i, 'naphtha') > 0}
        Flow[i, 'reformed_gasoline']
    <= &naphtha_ub;

```

```

con Cracked_oil_ub_con:
    sum {<i,'cracked_oil'> in ARCS} Flow[i,'cracked_oil'] <= &cracked_oil_ub;

con Lube_oil_range_con:
    &lube_oil_lb <= Flow['lube_oil','sink'] <= &lube_oil_ub;

```

As expressed on page 68, the premium ratio constraint involves a ratio of linear functions of the decision variables. The following CON statement linearizes the nonlinear ratio constraint by clearing the denominator:

```

con Premium_ratio_con:
    sum {<'premium_petrol',j> in ARCS} Flow['premium_petrol',j]
>= &premium_ratio * sum {<'regular_petrol',j> in ARCS} Flow['regular_petrol',j];

num octane_sol {petrol in PETROLS} =
    (sum {<i,(petrol)> in ARCS} octane[i] * arc_mult[i,petrol] * Flow[i,petrol].sol) /
    (sum {<i,(petrol)> in ARCS} arc_mult[i,petrol] * Flow[i,petrol].sol);

num vapour_pressure_sol =
    (sum {<i,'jet_fuel'> in ARCS}
        vapour_pressure[i] * arc_mult[i,'jet_fuel'] * Flow[i,'jet_fuel'].sol) /
    (sum {<i,'jet_fuel'> in ARCS} arc_mult[i,'jet_fuel'] * Flow[i,'jet_fuel'].sol);

num fuel_oil_ratio_sol {<i,'fuel_oil'> in ARCS} =
    (arc_mult[i,'fuel_oil'] * Flow[i,'fuel_oil'].sol) /
    (sum {<j,'fuel_oil'> in ARCS} arc_mult[j,'fuel_oil'] * Flow[j,'fuel_oil'].sol);

solve;
print CrudeDistilled;
print OilCracked Flow;
print octane_sol octane_lb;

```

Although the previous PRINT statements print all values of the given parameters, the following two PRINT statements use an index set to print a specified subset of the values:

```

print {<i,'jet_fuel'> in ARCS} vapour_pressure vapour_pressure_sol;
print {<i,'fuel_oil'> in ARCS} fuel_oil_coefficient fuel_oil_ratio_sol;
create data sol_data1 from [i j] Flow;
quit;

```

Figure 6.2 shows the output from the linear programming solver. For this test instance, it turns out that the optimal solution contains no fuel oil.

Figure 6.2 Output from Linear Programming Solver

```

The OPTMODEL Procedure

Problem Summary

Objective Sense           Maximization
Objective Function       TotalProfit
Objective Type            Linear

Number of Variables      51
Bounded Above            0
Bounded Below            49
Bounded Below and Above  2
Free                     0
Fixed                    0

Number of Constraints     46
Linear LE (<=)           4
Linear EQ (=)            38
Linear GE (>=)           3
Linear Range             1

Constraint Coefficients  158

Solution Summary

Solver                   Dual Simplex
Objective Function       TotalProfit
Solution Status          Optimal
Objective Value          211365.13477
Iterations               20

Primal Infeasibility     9.15179E-12
Dual Infeasibility       1.082467E-15
Bound Infeasibility      0

[1]      Crude
         Distilled

         crude1      15000
         crude2      30000

[1]      Oil
         Cracked

         heavy_oil_cracked      3800
         light_oil_cracked      4200

```

Figure 6.2 continued

[1]	[2]	Flow
cracked_gasoline	premium_petrol	411.63
cracked_gasoline	regular_petrol	1524.37
cracked_oil	fuel_oil	0.00
cracked_oil	jet_fuel	5706.00
crude1	heavy_naphtha	15000.00
crude1	heavy_oil	15000.00
crude1	light_naphtha	15000.00
crude1	light_oil	15000.00
crude1	medium_naphtha	15000.00
crude1	residuum	15000.00
crude2	heavy_naphtha	30000.00
crude2	heavy_oil	30000.00
crude2	light_naphtha	30000.00
crude2	light_oil	30000.00
crude2	medium_naphtha	30000.00
crude2	residuum	30000.00
fuel_oil	sink	0.00
heavy_naphtha	premium_petrol	0.00
heavy_naphtha	reformed_gasoline	5406.86
heavy_naphtha	regular_petrol	2993.14
heavy_oil	fuel_oil	0.00
heavy_oil	heavy_oil_cracked	3800.00
heavy_oil	jet_fuel	4900.00
heavy_oil_cracked	cracked_gasoline	3800.00
heavy_oil_cracked	cracked_oil	3800.00
jet_fuel	sink	15156.00
light_naphtha	premium_petrol	0.00
light_naphtha	reformed_gasoline	0.00
light_naphtha	regular_petrol	6000.00
light_oil	fuel_oil	0.00
light_oil	jet_fuel	0.00
light_oil	light_oil_cracked	4200.00
light_oil_cracked	cracked_gasoline	4200.00
light_oil_cracked	cracked_oil	4200.00
lube_oil	sink	500.00
medium_naphtha	premium_petrol	3973.06
medium_naphtha	reformed_gasoline	0.00
medium_naphtha	regular_petrol	6526.94
premium_petrol	sink	6817.78
reformed_gasoline	premium_petrol	2433.09
reformed_gasoline	regular_petrol	0.00
regular_petrol	sink	17044.45
residuum	fuel_oil	0.00
residuum	jet_fuel	4550.00
residuum	lube_oil	1000.00
source	crude1	15000.00
source	crude2	30000.00

[1]	octane_	
	sol	octane_lb
premium_petrol	94	94
regular_petrol	84	84

Figure 6.2 continued

	[1]	vapour_ pressure	
	cracked_oil	1.50	
	heavy_oil	0.60	
	light_oil	1.00	
	residuum	0.05	
		vapour_ pressure_ sol	
		0.77372	
	[1]	fuel_oil_ coefficient	fuel_oil_ ratio_sol
	cracked_oil	4	.
	heavy_oil	3	.
	light_oil	10	.
	residuum	1	.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming (generalized network flow with side constraints)
- numeric and string index sets
- reading sparse two-dimensional data
- NOMISS option
- sets of tuples
- bounds in the VAR statement
- `.ub` variable suffix
- using a colon (:) to select members of a set
- set operators UNION, DIFF, CROSS, and SETOF
- linearizing a ratio constraint
- range constraint

- INDEX function
- implicit slice
- using a variable suffix (such as `.s01`) in the declaration of a numeric parameter
- index set in the PRINT statement

Chapter 7

Mining: Which Pits to Work and When to Close Them Down

Contents

Problem Statement	79
Mathematical Programming Formulation	80
Input Data	82
PROC OPTMODEL Statements and Output	83
Features Demonstrated	87

Problem Statement

A mining company is going to continue operating in a certain area for the next five years.¹ There are four mines in this area but it can operate at most three in any one year. Although a mine may not operate in a certain year it is still necessary to keep it ‘open’, in the sense that royalties are payable, should it be operated in a future year. Clearly if a mine is not going to be worked again it can be closed down permanently and no more royalties need be paid. The yearly royalties payable on each mine kept ‘open’ are

Mine 1	£5 million
Mine 2	£4 million
Mine 3	£4 million
Mine 4	£5 million

There is an upper limit to the amount of ore which can be extracted from each mine in a year. These upper limits are:

Mine 1	2×10^6 tons
Mine 2	2.5×10^6 tons
Mine 3	1.3×10^6 tons
Mine 4	3×10^6 tons

The ore from the different mines is of varying quality. This quality is measured on a scale so that blending ores together results in a linear combination of the quality measurements, e.g. if equal quantities of two ores were combined the resultant ore would have a quality measurement half way between that of the ingredient ores. Measured in these units the qualities of the ores from the mines are given below:

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 238–239).

Mine 1	1.0
Mine 2	0.7
Mine 3	1.5
Mine 4	0.5

In each year it is necessary to combine the total outputs from each mine to produce a blended ore of exactly some stipulated quality. For each year these qualities are

Year 1	0.9
Year 2	0.8
Year 3	1.2
Year 4	0.6
Year 5	1.0

The final blended ore sells for £10 per ton each year. Revenue and expenditure for future years must be discounted at a rate of 10% per annum.

Which mines should be operated each year and how much should they produce?

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{mine} \in \text{MINES}$
- $\text{year} \in \text{YEARS}$

Parameters

Table 7.1 shows the parameters used in this example.

Table 7.1 Parameters

Parameter Name	Interpretation
<i>cost[mine]</i>	Yearly royalties payable on each mine kept open
<i>extract_ub[mine]</i>	Upper bound on amount of ore extracted from each mine in a year
<i>quality[mine]</i>	Quality of ore from each mine
<i>quality_required[year]</i>	Required quality of blended ore per year
<i>max_num_worked_per_year</i>	Maximum number of mines worked per year
<i>revenue_per_ton</i>	Revenue per ton of blended ore
<i>discount_rate</i>	Annual discount rate for future years
<i>Extract[mine,year].ub</i>	Upper bound on Extract[mine,year]
<i>quality_sol[year]</i>	Quality of blended ore per year

Variables

Table 7.2 shows the variables used in this example.

Table 7.2 Variables

Variable Name	Interpretation
<i>IsOpen[mine,year]</i>	1 if mine is open in that year; 0 otherwise
<i>IsWorked[mine,year]</i>	1 if mine is worked in that year; 0 otherwise
<i>Extract[mine,year]</i>	Amount of ore extracted per mine per year
<i>ExtractedPerYear[year]</i>	Amount of ore extracted per year

Objective

The objective is to maximize the following function, where *TotalRevenue* is a linear function of *ExtractedPerYear*, and *TotalCost* is a linear function of *IsOpen*:

$$\text{TotalProfit} = \text{TotalRevenue} - \text{TotalCost}$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $\text{year} \in \text{YEARS}$,

$$\text{ExtractedPerYear}[\text{year}] = \sum_{\text{mine} \in \text{MINES}} \text{Extract}[\text{mine}, \text{year}]$$

- for mine \in MINES and year \in YEARS,

$$\text{Extract}[\text{mine}, \text{year}] \leq \text{Extract}[\text{mine}, \text{year}].ub \cdot \text{IsWorked}[\text{mine}, \text{year}]$$

- for year \in YEARS,

$$\sum_{\text{mine} \in \text{MINES}} \text{IsWorked}[\text{mine}, \text{year}] \leq \text{max_num_worked_per_year}$$

- for mine \in MINES and year \in YEARS,

$$\text{IsWorked}[\text{mine}, \text{year}] \leq \text{IsOpen}[\text{mine}, \text{year}]$$

- for mine \in MINES and year $\in \text{YEARS} \setminus \{1\}$,

$$\text{IsOpen}[\text{mine}, \text{year}] \leq \text{IsOpen}[\text{mine}, \text{year} - 1]$$

- for year \in YEARS,

$$\frac{\sum_{\text{mine} \in \text{MINES}} \text{quality}[\text{mine}] \cdot \text{Extract}[\text{mine}, \text{year}]}{\text{ExtractedPerYear}[\text{year}]} = \text{quality_required}[\text{year}]$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data mine_data;
  input mine $ cost extract_ub quality;
  datalines;
mine1 5 2    1.0
mine2 4 2.5 0.7
mine3 4 1.3 1.5
mine4 5 3    0.5
;

data year_data;
  input year quality_required;
  datalines;
1 0.9
2 0.8
3 1.2
4 0.6
5 1.0
;

%let max_num_worked_per_year = 3;
%let revenue_per_ton = 10;
%let discount_rate = 0.10;
```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements are straightforward declarations of index sets, parameters, and variables:

```
proc optmodel;
  set <str> MINES;
  num cost {MINES};
  num extract_ub {MINES};
  num quality {MINES};
  read data mine_data into MINES=[mine] cost extract_ub quality;

  set YEARS;
  num quality_required {YEARS};
  read data year_data into YEARS=[year] quality_required;

  var IsOpen {MINES, YEARS} binary;
  var IsWorked {MINES, YEARS} binary;
  var Extract {mine in MINES, YEARS} >= 0 <= extract_ub[mine];
```

The following IMPVAR statement declares ExtractedPerYear as an implicit variable. Using the IMPVAR statement is an alternative to using a VAR statement and an additional constraint.

```
impvar ExtractedPerYear {year in YEARS} = sum {mine in MINES} Extract[mine,year];
```

The following NUM statement uses a formula to compute the discount (shown in [Figure 7.1](#)) that is used in the objective function:

```
num discount {year in YEARS} = 1 / (1 + &discount_rate)^(year - 1);
print discount;

impvar TotalRevenue =
  &revenue_per_ton * sum {year in YEARS} discount[year] * ExtractedPerYear[year];
impvar TotalCost =
  sum {mine in MINES, year in YEARS} discount[year] * cost[mine] * IsOpen[mine,year];
max TotalProfit = TotalRevenue - TotalCost;
```

Figure 7.1 *discount* Parameter

The OPTMODEL Procedure	
[1]	discount
1	1.00000
2	0.90909
3	0.82645
4	0.75131
5	0.68301

The following Link constraint enforces the rule that $\text{Extract}[\text{mine}, \text{year}] > 0$ implies that $\text{IsWorked}[\text{mine}, \text{year}] = 1$ (as in [Chapter 2](#)):

```

con Link {mine in MINES, year in YEARS}:
    Extract[mine,year] <= Extract[mine,year].ub * IsWorked[mine,year];

```

The following Cardinality constraint enforces the limit on number of mines worked per year:

```

con Cardinality {year in YEARS}:
    sum {mine in MINES} IsWorked[mine,year] <= &max_num_worked_per_year;

```

The following Worked_implies_open constraint enforces the rule that $\text{IsWorked}[\text{mine}, \text{year}] = 1$ implies that $\text{IsOpen}[\text{mine}, \text{year}] = 1$:

```

con Worked_implies_open {mine in MINES, year in YEARS}:
    IsWorked[mine,year] <= IsOpen[mine,year];

```

The following Continuity constraint enforces the rule that $\text{IsOpen}[\text{mine}, \text{year}] = 1$ implies that $\text{IsOpen}[\text{mine}, \text{year} - 1] = 1$:

```

con Continuity {mine in MINES, year in YEARS diff {1}}:
    IsOpen[mine,year] <= IsOpen[mine,year-1];

```

As expressed on page 82, the quality of the blended ore for each year is a ratio of linear functions of the decision variables. The following CON statement linearizes the nonlinear ratio constraint by clearing the denominator:

```

con Quality_con {year in YEARS}:
    sum {mine in MINES} quality[mine] * Extract[mine,year]
    = quality_required[year] * ExtractedPerYear[year];

```

By using the `.sol` suffix, the numeric parameter *quality_sol* computes the quality of the blended ore from the optimal decision variable values that are returned by the solver:

```

num quality_sol {year in YEARS} =
    (sum {mine in MINES} quality[mine] * Extract[mine,year].sol) /
    ExtractedPerYear[year].sol;

solve;
print IsOpen IsWorked Extract;
print ExtractedPerYear quality_sol quality_required;
create data sol_data1 from [mine year] IsOpen IsWorked Extract;
create data sol_data2 from [year] ExtractedPerYear;
quit;

```

Figure 7.2 shows the output from the mixed integer linear programming solver. The values of *quality_sol* and *quality_required* agree, as enforced by the *Quality_con* constraint.

Figure 7.2 Output from Mixed Integer Linear Programming Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	60
Bounded Above	0
Bounded Below	0
Bounded Below and Above	60
Free	0
Fixed	0
Binary	40
Integer	0
Number of Constraints	66
Linear LE (\leq)	61
Linear EQ ($=$)	5
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	151
Solution Summary	
Solver	MILP
Objective Function	TotalProfit
Solution Status	Optimal
Objective Value	146.86197436
Iterations	55
Best Bound	.
Nodes	1
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	5.008689E-15
Bound Infeasibility	2.637864E-15
Integer Infeasibility	2E-11

Figure 7.2 continued

[1]	[2]	Is Open	Is Worked	Extract
mine1	1	1	1	2.0000
mine1	2	1	0	0.0000
mine1	3	1	1	1.9500
mine1	4	1	1	0.1250
mine1	5	1	1	2.0000
mine2	1	1	-0	-0.0000
mine2	2	1	1	2.5000
mine2	3	1	0	0.0000
mine2	4	1	1	2.5000
mine2	5	1	1	2.1667
mine3	1	1	1	1.3000
mine3	2	1	1	1.3000
mine3	3	1	1	1.3000
mine3	4	1	0	0.0000
mine3	5	1	1	1.3000
mine4	1	1	1	2.4500
mine4	2	1	1	2.2000
mine4	3	1	0	0.0000
mine4	4	1	1	3.0000
mine4	5	0	0	0.0000

[1]	Extracted PerYear	quality_ sol	quality_ required
1	5.7500	0.9	0.9
2	6.0000	0.8	0.8
3	3.2500	1.2	1.2
4	5.6250	0.6	0.6
5	5.4667	1.0	1.0

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- numeric and string index sets
- calculated numeric parameter
- bounds in the VAR statement
- IMPVAR statement
- `.ub` variable suffix
- set operator DIFF
- linearizing a ratio constraint
- using a variable suffix (such as `.so1`) in the declaration of a numeric parameter
- multiple input and output data sets

Chapter 8

Farm Planning: How Much to Grow and Rear

Contents

Problem Statement	89
Mathematical Programming Formulation	90
Input Data	94
PROC OPTMODEL Statements and Output	95
Features Demonstrated	102

Problem Statement

A farmer wishes to plan production on his 200 acre farm over the next five years.¹

At present he has a herd of 120 cows. This is made up of 20 heifers and 100 milk-producing cows. Each heifer needs $\frac{2}{3}$ acre to support it and each dairy cow 1 acre. A dairy cow produces an average of 1.1 calves per year. Half of these calves will be bullocks which are sold almost immediately for an average of £30 each. The remaining heifers can be either sold almost immediately for £40 or reared to become milk-producing cows at two years old. It is intended that all dairy cows be sold at 12 years old for an average of £120 each, although there will probably be an annual loss of 5% per year among heifers and 2% among dairy cows. At present there are 10 cows of each age from newborn to 11 years old. The decision of how many heifers to sell in the current year has already been taken and implemented.

The milk from a cow yields an annual revenue of £370. A maximum of 130 cows can be housed at the present time. To provide accommodation for each cow beyond this number will entail a capital outlay of £200 per cow. Each milk-producing cow requires 0.6 tons of grain and 0.7 tons of sugar beet per year. Grain and sugar beet can both be grown on the farm. Each acre yields 1.5 tons of sugar beet. Only 80 acres are suitable for growing grain. They can be divided into four groups whose yields are as follows:

group 1	20 acres	1.1 tons per acre
group 2	30 acres	0.9 tons per acre
group 3	20 acres	0.8 tons per acre
group 4	10 acres	0.65 tons per acre

Grain can be bought for £90 per ton and sold for £75 per ton. Sugar beet can be bought for £70 per ton and sold for £58 per ton.

The labour requirements are:

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 239–240).

each heifer	10 hours per year
each milk-producing cow	42 hours per year
each acre put to grain	4 hours per year
each acre put to sugar beet	14 hours per year

Other costs are:

each heifer	£50 per year
each milk-producing cow	£100 per year
each acre put to grain	£15 per year
each acre put to sugar beet	£10 per year

Labour costs for the farm are at present £4000 per year and provide 5500 hours labour. Any labour needed above this will cost £1.20 per hour.

How should the farmer operate over the next five years to maximize profit? Any capital expenditure would be financed by a 10 year loan at 15% annual interest. The interest and capital repayment would be paid in 10 equally sized yearly instalments. In no year can the cash flow be negative. Lastly, the farmer would not wish to reduce the total number of dairy cows at the end of the five year period by more than 50% nor increase the number by more than 75%.

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{age} \in \text{AGES}$
- $\text{year} \in \text{YEARS}$
- $\text{year} \in \text{YEARS0} = \{0\} \cup \text{YEARS}$
- $\text{group} \in \text{GROUPS}$

Parameters

Table 8.1 shows the parameters used in this example.

Table 8.1 Parameters

Parameter Name	Interpretation
<i>init_num_cows[age]</i>	Initial number of cows of each age
<i>acres_needed[age]</i>	Number of acres needed to support a cow of each age
<i>annual_loss[age]</i>	Annual percentage loss per cow of each age
<i>bullock_yield[age]</i>	Number of bullocks yielded per year per cow of each age
<i>heifer_yield[age]</i>	Number of heifers yielded per year per cow of each age
<i>milk_revenue[age]</i>	Milk revenue per year per cow of each age
<i>grain_req[age]</i>	Tons of grain required per year per cow of each age
<i>sugar_beet_req[age]</i>	Tons of sugar beet required per year per cow of each age
<i>cow_labour_req[age]</i>	Number of labour hours required per year per cow of each age
<i>cow_other_costs[age]</i>	Other costs per year per cow of each age
<i>acres[group]</i>	Number of acres per group
<i>grain_yield[group]</i>	Tons of grain yielded per acre by each group
<i>yearly_loan_payment</i>	Amount of yearly loan payment per capital outlay
<i>num_years</i>	Number of years in planning horizon
<i>num_acres</i>	Number of acres in farm
<i>bullock_revenue</i>	Revenue for selling a bullock
<i>heifer_revenue</i>	Revenue for selling a heifer
<i>dairy_cow_selling_age</i>	Age by which a dairy cow is sold
<i>dairy_cow_selling_revenue</i>	Revenue for selling a dairy cow
<i>max_num_cows</i>	Maximum number of cows that can be housed at present
<i>sugar_beet_yield</i>	Tons of sugar beet yielded per acre
<i>grain_cost</i>	Cost per ton of grain bought
<i>grain_revenue</i>	Revenue per ton of grain sold
<i>grain_labour_req</i>	Number of labour hours required per year per acre of grain
<i>grain_other_costs</i>	Other costs per year per acre of grain
<i>sugar_beet_cost</i>	Cost per ton of sugar beet bought
<i>sugar_beet_revenue</i>	Revenue per ton of sugar beet sold
<i>sugar_beet_labour_req</i>	Number of labour hours required per year per acre of sugar beet
<i>sugar_beet_other_costs</i>	Other costs per year per acre of sugar beet
<i>nominal_labour_cost</i>	Present labour costs for farm per year
<i>nominal_labour_hours</i>	Maximum number of labour hours available without incurring additional labour costs
<i>excess_labour_cost</i>	Cost per hour of additional labour
<i>capital_outlay_unit</i>	Capital outlay per additional cow beyond <i>max_num_cows</i>
<i>num_loan_years</i>	Number of years for loan
<i>annual_interest_rate</i>	Annual interest rate for loan
<i>max_decrease_ratio</i>	Maximum relative decrease in total number of cows at end of planning horizon
<i>max_increase_ratio</i>	Maximum relative increase in total number of cows at end of planning horizon

Variables

Table 8.2 shows the variables used in this example.

Table 8.2 Variables

Variable Name	Interpretation
NumCows[age,year]	Number of cows of each age per year
NumBullocksSold[year]	Number of bullocks sold per year
NumHeifersSold[year]	Number of heifers sold per year
GrainAcres[group,year]	Acres of grain per group and year
GrainBought[year]	Tons of grain bought per year
GrainSold[year]	Tons of grain sold per year
SugarBeetAcres[year]	Acres of sugar beet per year
SugarBeetBought[year]	Tons of sugar beet bought per year
SugarBeetSold[year]	Tons of sugar beet sold per year
NumExcessLabourHours[year]	Number of additional labour hours per year
CapitalOutlay[year]	Number of capital outlays per year
GrainGrown[group,year]	Tons of grain grown per group and year
SugarBeetGrown[year]	Tons of sugar beet grown per year
Revenue[year]	Revenue per year
Cost[year]	Cost per year
Profit[year]	Profit per year

Objective

The objective is to maximize the following function, where $\text{Profit}[\text{year}] = \text{Revenue}[\text{year}] - \text{Cost}[\text{year}]$ and Revenue and Cost are linear functions of other variables:

TotalProfit =

$$\sum_{\text{year} \in \text{YEARS}} (\text{Profit}[\text{year}] - \text{yearly_loan_payment} \cdot (\text{num_years} - 1 + \text{year}) \cdot \text{CapitalOutlay}[\text{year}])$$

Constraints

The following constraints are used in this example:

- bounds on variables

- for $\text{year} \in \text{YEARS}$,

$$\begin{aligned} & \sum_{\text{age} \in \text{AGES}} \text{acres_needed}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}] \\ & + \sum_{\text{group} \in \text{GROUPS}} \text{GrainAcres}[\text{group}, \text{year}] + \text{SugarBeetAcres}[\text{year}] \\ & \leq \text{num_acres} \end{aligned}$$

- for $\text{age} \in \text{AGES} \setminus \{\text{dairy_cow_selling_age}\}$ and $\text{year} \in \text{YEARS0} \setminus \{\text{num_years}\}$,

$$\text{NumCows}[\text{age} + 1, \text{year} + 1] = (1 - \text{annual_loss}[\text{age}]) \cdot \text{NumCows}[\text{age}, \text{year}]$$

- for $\text{year} \in \text{YEARS}$,

$$\text{NumBullocksSold}[\text{year}] = \sum_{\text{age} \in \text{AGES}} \text{bullock_yield}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}]$$

- for $\text{year} \in \text{YEARS}$,

$$\text{NumCows}[0, \text{year}] = \sum_{\text{age} \in \text{AGES}} \text{heifer_yield}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}] - \text{NumHeifersSold}[\text{year}]$$

- for $\text{year} \in \text{YEARS}$,

$$\sum_{\text{age} \in \text{AGES}} \text{NumCows}[\text{age}, \text{year}] \leq \text{max_num_cows} + \sum_{\substack{\text{y} \in \text{YEARS}: \\ \text{y} \leq \text{year}}} \text{CapitalOutlay}[\text{y}]$$

- for $\text{group} \in \text{GROUPS}$ and $\text{year} \in \text{YEARS}$,

$$\text{GrainGrown}[\text{group}, \text{year}] = \text{grain_yield}[\text{group}] \cdot \text{GrainAcres}[\text{group}, \text{year}]$$

- for $\text{year} \in \text{YEARS}$,

$$\begin{aligned} & \sum_{\text{age} \in \text{AGES}} \text{grain_req}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}] \\ & \leq \sum_{\text{group} \in \text{GROUPS}} \text{GrainGrown}[\text{group}, \text{year}] + \text{GrainBought}[\text{year}] - \text{GrainSold}[\text{year}] \end{aligned}$$

- for $\text{year} \in \text{YEARS}$,

$$\text{SugarBeetGrown}[\text{year}] = \text{sugar_beet_yield} \cdot \text{SugarBeetAcres}[\text{year}]$$

- for $\text{year} \in \text{YEARS}$,

$$\begin{aligned} & \sum_{\text{age} \in \text{AGES}} \text{sugar_beet_req}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}] \\ & \leq \text{SugarBeetGrown}[\text{year}] + \text{SugarBeetBought}[\text{year}] - \text{SugarBeetSold}[\text{year}] \end{aligned}$$

- for year \in YEARS,

$$\begin{aligned}
 & \sum_{\text{age} \in \text{AGES}} \text{cow_labour_req}[\text{age}] \cdot \text{NumCows}[\text{age}, \text{year}] \\
 & + \sum_{\text{group} \in \text{GROUPS}} \text{grain_labour_req} \cdot \text{GrainAcres}[\text{group}, \text{year}] \\
 & + \text{sugar_beet_labour_req} \cdot \text{SugarBeetAcres}[\text{year}] \\
 & \leq \text{nominal_labour_hours} + \text{NumExcessLabourHours}[\text{year}]
 \end{aligned}$$

- for year \in YEARS,

$$\text{Profit}[\text{year}] \geq 0$$

- $1 - \text{max_decrease_ratio} \leq \frac{\sum_{\substack{\text{age} \in \text{AGES}: \\ \text{age} \geq 2}} \text{NumCows}[\text{age}, \text{num_years}]}{\sum_{\substack{\text{age} \in \text{AGES}: \\ \text{age} \geq 2}} \text{init_num_cows}[\text{age}]} \leq 1 + \text{max_increase_ratio}$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```

data cow_data;
  do age = 0 to 11;
    init_num_cows = 10;
    if age < 2 then do;
      acres_needed = 2/3;
      annual_loss = 0.05;
      bullock_yield = 0;
      heifer_yield = 0;
      milk_revenue = 0;
      grain_req = 0;
      sugar_beet_req = 0;
      labour_req = 10;
      other_costs = 50;
    end;
    else do;
      acres_needed = 1;
      annual_loss = 0.02;
      bullock_yield = 1.1/2;
      heifer_yield = 1.1/2;
      milk_revenue = 370;
      grain_req = 0.6;
      sugar_beet_req = 0.7;
      labour_req = 42;
      other_costs = 100;
    end;
  end;
output;

```



```

    end;
run;

data grain_data;
    input group $ acres yield;
    datalines;
group1 20 1.1
group2 30 0.9
group3 20 0.8
group4 10 0.65
;

%let num_years = 5;
%let num_acres = 200;
%let bullock_revenue = 30;
%let heifer_revenue = 40;
%let dairy_cow_selling_age = 12;
%let dairy_cow_selling_revenue = 120;
%let max_num_cows = 130;
%let sugar_beet_yield = 1.5;
%let grain_cost = 90;
%let grain_revenue = 75;
%let grain_labour_req = 4;
%let grain_other_costs = 15;
%let sugar_beet_cost = 70;
%let sugar_beet_revenue = 58;
%let sugar_beet_labour_req = 14;
%let sugar_beet_other_costs = 10;
%let nominal_labour_cost = 4000;
%let nominal_labour_hours = 5500;
%let excess_labour_cost = 1.2;
%let capital_outlay_unit = 200;
%let num_loan_years = 10;
%let annual_interest_rate = 0.15;
%let max_decrease_ratio = 0.50;
%let max_increase_ratio = 0.75;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements are straightforward declarations of index sets, parameters, and variables:

```

proc optmodel;
    set AGES;
    num init_num_cows {AGES};
    num acres_needed {AGES};
    num annual_loss {AGES};
    num bullock_yield {AGES};
    num heifer_yield {AGES};
    num milk_revenue {AGES};
    num grain_req {AGES};

```

```

num sugar_beet_req {AGES};
num cow_labour_req {AGES};
num cow_other_costs {AGES};
read data cow_data into AGES=[age]
    init_num_cows acres_needed annual_loss bullock_yield heifer_yield milk_revenue
    grain_req sugar_beet_req cow_labour_req=labour_req cow_other_costs=other_costs;

num num_years = &num_years;
set YEARS = 1..num_years;
set YEARS0 = {0} union YEARS;

var NumCows {AGES union {&dairy_cow_selling_age}, YEARS0} >= 0;
for {age in AGES} fix NumCows[age,0] = init_num_cows[age];
fix NumCows[&dairy_cow_selling_age,0] = 0;

var NumBullocksSold {YEARS} >= 0;
var NumHeifersSold {YEARS} >= 0;

set <str> GROUPS;
num acres {GROUPS};
num grain_yield {GROUPS};
var GrainAcres {GROUPS, YEARS} >= 0;

```

In [Chapter 6](#), the READ DATA statement was used to populate the `.ub` suffix for a one-dimensional variable. The following READ DATA statement populates `.ub` for the two-dimensional variable GrainAcres:

```

read data grain_data into GROUPS=[group]
    {year in YEARS} <GrainAcres[group,year].ub=acres>
    grain_yield=yield;
var GrainBought {YEARS} >= 0;
var GrainSold {YEARS} >= 0;

var SugarBeetAcres {YEARS} >= 0;
var SugarBeetBought {YEARS} >= 0;
var SugarBeetSold {YEARS} >= 0;

var NumExcessLabourHours {YEARS} >= 0;
var CapitalOutlay {YEARS} >= 0;

```

The following NUM statement uses the FINANCE function to calculate the yearly loan payment per capital outlay:

```

num yearly_loan_payment =
    -finance('pmt', &annual_interest_rate, &num_loan_years, &capital_outlay_unit);
print yearly_loan_payment;

```

The resulting value of *yearly_loan_payment* (shown in [Figure 8.1](#)) differs from the value given in Williams, perhaps due to rounding in intermediate calculations by Williams. The formula

$$\text{yearly_loan_payment} = \frac{\text{annual_interest_rate} \cdot \text{capital_outlay_unit}}{1 - (1 + \text{annual_interest_rate})^{-\text{num_loan_years}}}$$

yields the same value as the FINANCE function. For the given input data, it turns out that the optimal solution has no capital outlay and agrees with the solution reported in Williams.

Figure 8.1 *yearly_loan_payment* Parameter

The OPTMODEL Procedure	
yearly_	
loan_	
payment	
	39.85

The following IMPVAR statements declare Revenue, Cost, and Profit as linear functions of the decision variables:

```

impvar Revenue {year in YEARS} =
    &bullock_revenue * NumBullocksSold[year]
  + &heifer_revenue * NumHeifersSold[year]
  + &dairy_cow_selling_revenue * NumCows[&dairy_cow_selling_age,year]
  + sum {age in AGES} milk_revenue[age] * NumCows[age,year]
  + &grain_revenue * GrainSold[year]
  + &sugar_beet_revenue * SugarBeetSold[year]
;
impvar Cost {year in YEARS} =
    &grain_cost * GrainBought[year]
  + &sugar_beet_cost * SugarBeetBought[year]
  + &nominal_labour_cost
  + &excess_labour_cost * NumExcessLabourHours[year]
  + sum {age in AGES} cow_other_costs[age] * NumCows[age,year]
  + sum {group in GROUPS} &grain_other_costs * GrainAcres[group,year]
  + &sugar_beet_other_costs * SugarBeetAcres[year]
  + sum {y in YEARS: y <= year} yearly_loan_payment * CapitalOutlay[y]
;
impvar Profit {year in YEARS} = Revenue[year] - Cost[year];

```

The following objective declaration accounts for loan repayments beyond the planning horizon, as described in Williams:

```

max TotalProfit =
    sum {year in YEARS} (Profit[year]
        - yearly_loan_payment * (num_years - 1 + year) * CapitalOutlay[year]);

```

The following model declaration statements are straightforward:

```

con Num_acres_con {year in YEARS}:
    sum {age in AGES} acres_needed[age] * NumCows[age,year]
  + sum {group in GROUPS} GrainAcres[group,year]
  + SugarBeetAcres[year]
<= &num_acres;

con Aging {age in AGES diff {&dairy_cow_selling_age},
    year in YEARS0 diff {num_years}}:
    NumCows[age+1,year+1] = (1 - annual_loss[age]) * NumCows[age,year];

```

```

con NumBullocksSold_def {year in YEARS}:
    NumBullocksSold[year]
    = sum {age in AGES} bullock_yield[age] * NumCows[age,year];

con NumHeifersSold_def {year in YEARS}:
    NumCows[0,year]
    = sum {age in AGES} heifer_yield[age] * NumCows[age,year]
    - NumHeifersSold[year];

con Max_num_cows_def {year in YEARS}:
    sum {age in AGES} NumCows[age,year]
    <= &max_num_cows + sum {y in YEARS: y <= year} CapitalOutlay[y];

impvar GrainGrown {group in GROUPS, year in YEARS} =
    grain_yield[group] * GrainAcres[group,year];
con Grain_req_def {year in YEARS}:
    sum {age in AGES} grain_req[age] * NumCows[age,year]
    <= sum {group in GROUPS} GrainGrown[group,year]
    + GrainBought[year] - GrainSold[year];

impvar SugarBeetGrown {year in YEARS} =
    &sugar_beet_yield * SugarBeetAcres[year];
con Sugar_beet_req_def {year in YEARS}:
    sum {age in AGES} sugar_beet_req[age] * NumCows[age,year]
    <= SugarBeetGrown[year] + SugarBeetBought[year] - SugarBeetSold[year];

con Labour_req_def {year in YEARS}:
    sum {age in AGES} cow_labour_req[age] * NumCows[age,year]
    + sum {group in GROUPS} &grain_labour_req * GrainAcres[group,year]
    + &sugar_beet_labour_req * SugarBeetAcres[year]
    <= &nominal_labour_hours + NumExcessLabourHours[year];

con Cash_flow {year in YEARS}:
    Profit[year] >= 0;

```

The following CON statement declares a range constraint:

```

con Final_dairy_cows_range:
    1 - &max_decrease_ratio
    <= (sum {age in AGES: age >= 2} NumCows[age,num_years])
    / (sum {age in AGES: age >= 2} init_num_cows[age])
    <= 1 + &max_increase_ratio;

solve;

print NumCows NumBullocksSold NumHeifersSold CapitalOutlay
    NumExcessLabourHours Revenue Cost Profit;
print GrainAcres;
print GrainGrown;
print GrainBought GrainSold SugarBeetAcres SugarBeetGrown SugarBeetBought
    SugarBeetSold;
print Num_acres_con.body Max_num_cows_def.body Final_dairy_cows_range.body;

```

The following CREATE DATA statement writes dense two-dimensional data to a data set:

```
create data sol_data1 from [age]=AGES
  {year in YEARS} <col('NumCows_year' || year)=NumCows[age,year].sol>;
```

You could instead use the following CREATE DATA statement to interchange the roles of *age* and *year*:

```
create data sol_data1 from [year]=YEARS
  {age in AGES} <col('NumCows_age' || age)=NumCows[age,year].sol>;
```

The following CREATE DATA statement writes sparse two-dimensional data to a data set:

```
create data sol_data2 from [group year] GrainAcres GrainGrown;
```

The final CREATE DATA statement writes one-dimensional data to a data set, as in previous examples:

```
create data sol_data3 from [year]
  NumBullocksSold NumHeifersSold CapitalOutlay NumExcessLabourHours
  Revenue Cost Profit GrainBought GrainSold
  SugarBeetAcres SugarBeetGrown SugarBeetBought SugarBeetSold;
quit;
```

Figure 8.2 shows the output from the linear programming solver.

Figure 8.2 Output from Linear Programming Solver

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	143
Bounded Above	0
Bounded Below	110
Bounded Below and Above	20
Free	0
Fixed	13
Number of Constraints	101
Linear LE (<=)	25
Linear EQ (=)	70
Linear GE (>=)	5
Linear Range	1
Constraint Coefficients	780

Figure 8.2 continued

Solution Summary						
Solver	Dual Simplex					
Objective Function	TotalProfit					
Solution Status	Optimal					
Objective Value	121719.17286					
Iterations	32					
Primal Infeasibility	5.684342E-14					
Dual Infeasibility	0					
Bound Infeasibility	0					

NumCows						
	0	1	2	3	4	5
0	10.0000	22.8000	11.5844	0.0000	0.0000	0.0000
1	10.0000	9.5000	21.6600	11.0052	0.0000	0.0000
2	10.0000	9.5000	9.0250	20.5770	10.4549	0.0000
3	10.0000	9.8000	9.3100	8.8445	20.1655	10.2458
4	10.0000	9.8000	9.6040	9.1238	8.6676	19.7622
5	10.0000	9.8000	9.6040	9.4119	8.9413	8.4943
6	10.0000	9.8000	9.6040	9.4119	9.2237	8.7625
7	10.0000	9.8000	9.6040	9.4119	9.2237	9.0392
8	10.0000	9.8000	9.6040	9.4119	9.2237	9.0392
9	10.0000	9.8000	9.6040	9.4119	9.2237	9.0392
10	10.0000	9.8000	9.6040	9.4119	9.2237	9.0392
11	10.0000	9.8000	9.6040	9.4119	9.2237	9.0392
12	0.0000	9.8000	9.6040	9.4119	9.2237	9.0392

[1]	Num						
	Bullocks Sold	Heifers Sold	Capital Outlay	Excess Labour Hours	Revenue	Cost	Profit
1	53.735	30.935	0	0	41495	19588	21906
2	52.342	40.757	0	0	41153	19265	21889
3	57.436	57.436	0	0	45212	19396	25816
4	56.964	56.964	0	0	45860	19034	26826
5	50.853	50.853	0	0	42717	17434	25283

GrainAcres						
	1	2	3	4	5	
group1	20.0000	20.0000	20.0000	20.0000	20.0000	
group2	0.0000	0.0000	3.1342	0.0000	0.0000	
group3	0.0000	0.0000	0.0000	0.0000	0.0000	
group4	0.0000	0.0000	0.0000	0.0000	0.0000	

Figure 8.2 continued

GrainGrown						
	1	2	3	4	5	
group1	22.0000	22.0000	22.0000	22.0000	22.0000	
group2	0.0000	0.0000	2.8207	0.0000	0.0000	
group3	0.0000	0.0000	0.0000	0.0000	0.0000	
group4	0.0000	0.0000	0.0000	0.0000	0.0000	

[1]	Grain	Grain	Sugar	Sugar	Sugar	Sugar
	Bought	Sold	Beet	Beet	Beet	Beet
			Acres	Grown	Bought	Sold
1	36.620	0	60.767	91.150	0	22.760
2	35.100	0	62.670	94.005	0	27.388
3	37.837	0	65.100	97.650	0	24.550
4	40.143	0	76.429	114.643	0	42.143
5	33.476	0	87.539	131.309	0	66.586

[1]	Num_acres_	Max_num_
	con.BODY	cows_def.
		BODY
1	200	130.000
2	200	128.411
3	200	115.434
4	200	103.571
5	200	92.461

Final_dairy_
cows_range.
BODY
0.92461

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- numeric and string index sets
- bounds in the VAR statement
- FIX statement
- IMPVAR statement
- `.ub` variable suffix
- set operators UNION and DIFF
- using a colon (:) to select members of a set
- FINANCE function
- range constraint
- multiple input and output data sets
- writing dense two-dimensional data
- writing sparse two-dimensional data

Chapter 9

Economic Planning: How Should an Economy Grow

Contents

Problem Statement	103
Mathematical Programming Formulation	104
Input Data	107
PROC OPTMODEL Statements and Output	107
Features Demonstrated	117

Problem Statement

An economy consists of three industries: coal, steel and transport.¹ Each unit produced by one of the industries (a unit will be taken as £1's worth of value of production) requires inputs from possibly its own industry as well as other industries. The required inputs as well as the manpower requirements (also measured in £) are given in Table 9.1. There is a time lag in the economy so that output in year $t + 1$ requires an input in year t .

Output from an industry may also be used to build productive capacity for itself or other industries in future years. The inputs required to give unit increases (capacity for £1's worth of extra production) in productive capacity are given in Table 9.2. Input from an industry in year t results in a (permanent) increase in productive capacity in year $t + 2$.

Table 9.1

Inputs (year t)	Outputs (year $t + 1$), production		
	Coal	Steel	Transport
Coal	0.1	0.5	0.4
Steel	0.1	0.1	0.2
Transport	0.2	0.1	0.2
Manpower	0.6	0.3	0.2

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 240–241).

Table 9.2

Inputs (year t)	Outputs (year $t + 2$), productive capacity		
	Coal	Steel	Transport
Coal	0.0	0.7	0.9
Steel	0.1	0.1	0.2
Transport	0.2	0.1	0.2
Manpower	0.4	0.2	0.1

Table 9.3

	Year 0	
	Stocks	Productive capacity
Coal	150	300
Steel	80	350
Transport	100	280

Stocks of goods may be held from year to year. At present (year 0) the stocks and productive capacities (per year) are given in Table 9.3 (in £m). There is a limited yearly manpower capacity of £470m.

It is wished to investigate different possible growth patterns for the economy over the next five years. In particular it is desirable to know the growth patterns which would result from pursuing the following objectives:

- (i) Maximizing total productive capacity at the end of the five years while meeting an exogenous consumption requirement of £60m of coal, £60m of steel, and £30m of transport in every year (apart from year 0).
- (ii) Maximizing total production (rather than productive capacity) in the fourth and fifth years, but ignoring exogenous demand in each year.
- (iii) Maximizing the total manpower requirement (ignoring the manpower capacity limitation) over the period while meeting the yearly exogenous demands of (i).

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{year} \in \text{YEARS}$
- $\text{year} \in \text{YEARS0} = \{0\} \cup \text{YEARS}$

- $i, j \in \text{INDUSTRIES}$
- $i \in \text{INPUTS}$

Parameters

Table 9.4 shows the parameters used in this example.

Table 9.4 Parameters

Parameter Name	Interpretation
<i>num_years</i>	Number of years in planning horizon
<i>init_stocks[i]</i>	Initial stocks per industry (in £m)
<i>init_productive_capacity[i]</i>	Initial productive capacity per industry (in £m)
<i>demand[i]</i>	Yearly exogenous demand per industry (in £m) for years 1 through <i>num_years</i>
<i>final_demand[i]</i>	Yearly exogenous demand per industry (in £m) for years <i>num_years</i> + 1 and later
<i>production_coeff[i,j]</i>	Production coefficient for input <i>i</i> and industry <i>j</i>
<i>productive_capacity_coeff[i,j]</i>	Productive capacity coefficient for input <i>i</i> and industry <i>j</i>
<i>manpower_capacity</i>	Yearly manpower capacity (in £m)

Variables

Table 9.5 shows the variables used in this example.

Table 9.5 Variables

Variable Name	Interpretation
StaticProduction[i]	Static production of industry <i>i</i>
Production[i,year]	Production of industry <i>i</i> per year
Stock[i,year]	Stock level of industry <i>i</i> at beginning of each year
ExtraCapacity[i,year]	Incremental productive capacity of industry <i>i</i> available per year
ProductiveCapacity[i,year]	Permanent productive capacity of industry <i>i</i> available per year

Objectives

The first objective is to maximize the following function:

$$\text{TotalProductiveCapacity} = \sum_{i \in \text{INDUSTRIES}} \text{ProductiveCapacity}[i, \text{num_years}]$$

The second objective is to maximize the following function:

$$\text{TotalProduction} = \sum_{i \in \text{INDUSTRIES}} \sum_{\text{year}=4}^5 \text{Production}[i, \text{year}]$$

The third objective is to maximize the following function:

$$\begin{aligned} \text{TotalManpower} = & \sum_{i \in \text{INDUSTRIES}} \sum_{\text{year} \in \text{YEARS}} (\text{production_coeff}[\text{'manpower'}, i] \cdot \text{Production}[i, \text{year} + 1] \\ & + \text{productive_capacity_coeff}[\text{'manpower'}, i] \cdot \text{ExtraCapacity}[i, \text{year} + 2]) \end{aligned}$$

Constraints

The following constraints are used in this example:

- bounds on variables

- for $i \in \text{INDUSTRIES}$,

$$\text{StaticProduction}[i] = \text{demand}[i] + \sum_{j \in \text{INDUSTRIES}} \text{production_coeff}[i, j] \cdot \text{StaticProduction}[j]$$

- for $i \in \text{INDUSTRIES}$ and $\text{year} \in \{1, \dots, \text{num_years} + 1\}$,

$$\text{ProductiveCapacity}[i, \text{year}] = \text{init_productive_capacity}[i] + \sum_{y=2}^{\text{year}} \text{ExtraCapacity}[i, y]$$

- for $i \in \text{INDUSTRIES}$ and $\text{year} \in \text{YEARS0}$,

$$\begin{aligned} & \text{Stock}[i, \text{year}] + \text{Production}[i, \text{year}] \\ & = (\text{if } \text{year} \in \text{YEARS} \text{ then } \text{demand}[i] \text{ else } 0) \\ & + \sum_{j \in \text{INDUSTRIES}} \text{production_coeff}[i, j] \cdot \text{Production}[j, \text{year} + 1] \\ & + \sum_{j \in \text{INDUSTRIES}} \text{productive_capacity_coeff}[i, j] \cdot \text{ExtraCapacity}[j, \text{year} + 2] \\ & + \text{Stock}[i, \text{year} + 1] \end{aligned}$$

- for $\text{year} \in \{1, \dots, \text{num_years} + 1\}$,

$$\begin{aligned} & \sum_{j \in \text{INDUSTRIES}} \text{production_coeff}[\text{'manpower'}, j] \cdot \text{Production}[j, \text{year}] \\ & + \sum_{j \in \text{INDUSTRIES}} \text{productive_capacity_coeff}[\text{'manpower'}, j] \cdot \text{ExtraCapacity}[j, \text{year} + 1] \\ & \leq \text{manpower_capacity} \end{aligned}$$

- for $i \in \text{INDUSTRIES}$ and $\text{year} \in \{1, \dots, \text{num_years} + 1\}$,

$$\text{Production}[i, \text{year}] \leq \text{ProductiveCapacity}[i, \text{year}]$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data industry_data;
    input industry $9. init_stocks init_productive_capacity demand;
    datalines;
coal      150 300 60
steel     80 350 60
transport 100 280 30
;

data production_data;
    input input $9. coal steel transport;
    datalines;
coal      0.1 0.5 0.4
steel     0.1 0.1 0.2
transport 0.2 0.1 0.2
manpower  0.6 0.3 0.2
;

data productive_capacity_data;
    input input $9. coal steel transport;
    datalines;
coal      0.0 0.7 0.9
steel     0.1 0.1 0.2
transport 0.2 0.1 0.2
manpower  0.4 0.2 0.1
;

%let manpower_capacity = 470;
%let num_years = 5;
```

PROC OPTMODEL Statements and Output

The following PROC OPTMODEL statements declare index sets and parameters and then read input data:

```
proc optmodel;
    num num_years = &num_years;
    set YEARS = 1..num_years;
    set YEARS0 = {0} union YEARS;

    set <str> INDUSTRIES;
    num init_stocks {INDUSTRIES};
    num init_productive_capacity {INDUSTRIES};
    num demand {INDUSTRIES};
    read data industry_data into INDUSTRIES=[industry]
        init_stocks init_productive_capacity demand;
```

```

set <str> INPUTS;
num production_coeff {INPUTS, INDUSTRIES};
read data production_data into INPUTS=[input]
  {j in INDUSTRIES} <production_coeff[input,j]=col(j)>;

num productive_capacity_coeff {INPUTS, INDUSTRIES};
read data productive_capacity_data into INPUTS=[input]
  {j in INDUSTRIES} <productive_capacity_coeff[input,j]=col(j)>;

```

The following PROC OPTMODEL statements declare variables, a constant objective, and constraints for the static Leontief input-output model that is described on page 282 of Williams:

```

var StaticProduction {INDUSTRIES} >= 0;
min Zero = 0;
con Static_con {i in INDUSTRIES}:
  StaticProduction[i]
    = demand[i] + sum {j in INDUSTRIES} production_coeff[i,j] * StaticProduction[j];

```

The following SOLVE statement invokes the linear programming solver to solve this system of equations:

```

solve;
print StaticProduction;

```

Figure 9.1 shows the output from the linear programming solver for the static model.

Figure 9.1 Output from Linear Programming Solver for Static Leontief Input-Output Model

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	Zero
Objective Type	Constant
Number of Variables	3
Bounded Above	0
Bounded Below	3
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	3
Linear LE (<=)	0
Linear EQ (=)	3
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	9

Figure 9.1 continued

Solution Summary	
Solver	Dual Simplex
Objective Function	Zero
Solution Status	Optimal
Objective Value	0
Iterations	0
Primal Infeasibility	2.131628E-14
Dual Infeasibility	0
Bound Infeasibility	0
Static	
[1]	Production
coal	166.397
steel	105.668
transport	92.308

The following NUM and assignment statements declare the *final_demand* parameter and use the *.sol* variable suffix to populate *final_demand* with the solution from the static model:

```
num final_demand {INDUSTRIES};
for {i in INDUSTRIES} final_demand[i] = StaticProduction[i].sol;
```

The following statements declare variables, implicit variables, objectives, and constraints to be used in all three parts of the business problem:

```
var Production {INDUSTRIES, 0..num_years+1} >= 0;
var Stock {INDUSTRIES, 0..num_years+1} >= 0;
var ExtraCapacity {INDUSTRIES, 1..num_years+2} >= 0;
impvar ProductiveCapacity {i in INDUSTRIES, year in 1..num_years+1} =
    init_productive_capacity[i] + sum {y in 2..year} ExtraCapacity[i,y];
for {i in INDUSTRIES} do;
    Production[i,0].ub = 0;
    Stock[i,0].lb = init_stocks[i];
    Stock[i,0].ub = init_stocks[i];
end;

max TotalProductiveCapacity =
    sum {i in INDUSTRIES} ProductiveCapacity[i,num_years];
max TotalProduction =
    sum {i in INDUSTRIES, year in 4..5} Production[i,year];
max TotalManpower =
    sum {i in INDUSTRIES, year in YEARS} (
        production_coeff['manpower',i] * Production[i,year+1]
        + productive_capacity_coeff['manpower',i] * ExtraCapacity[i,year+2]);

con Continuity_con {i in INDUSTRIES, year in YEARS0}:
    Stock[i,year] + Production[i,year]
    = (if year in YEARS then demand[i] else 0)
```

```

+ sum {j in INDUSTRIES} (
    production_coeff[i,j] * Production[j,year+1]
+ productive_capacity_coeff[i,j] * ExtraCapacity[j,year+2])
+ Stock[i,year+1];

con Manpower_con {year in 1..num_years+1}:
    sum {j in INDUSTRIES} (
        production_coeff['manpower',j] * Production[j,year]
    + productive_capacity_coeff['manpower',j] * ExtraCapacity[j,year+1])
    <= &manpower_capacity;

con Capacity_con {i in INDUSTRIES, year in 1..num_years+1}:
    Production[i,year] <= ProductiveCapacity[i,year];

for {i in INDUSTRIES}
    Production[i,num_years+1].lb = final_demand[i];

for {i in INDUSTRIES, year in num_years+1..num_years+2}
    ExtraCapacity[i,year].ub = 0;

```

The following PROBLEM statement specifies the variables, objective, and constraints for Problem1:

```

problem Problem1 include
    Production Stock ExtraCapacity
    TotalProductiveCapacity
    Continuity_con Manpower_con Capacity_con;

```

Because Problem1 and Problem2 have the same variables and constraints, the following PROBLEM statement uses the FROM option to copy these common parts from Problem1 and uses the INCLUDE option to specify only the new objective:

```

problem Problem2 from Problem1 include
    TotalProduction;

```

The following PROBLEM statement specifies the variables, objective, and constraints for Problem3 (omitting the Manpower_con constraint):

```

problem Problem3 include
    Production Stock ExtraCapacity
    TotalManpower
    Continuity_con Capacity_con;

```

Note that implicit variables are not included in the PROBLEM statements. Including them would yield an ERROR message.

The following USE PROBLEM statement switches focus to the desired problem:

```

use problem Problem1;
solve;
print Production Stock ExtraCapacity ProductiveCapacity Manpower_con.body;

```

Figure 9.2 shows the output from the linear programming solver for Problem1.

Figure 9.2 Output from Linear Programming Solver for Problem1

Problem Summary		
Objective Sense	Maximization	
Objective Function	TotalProductiveCapacity	
Objective Type	Linear	
Number of Variables	63	
Bounded Above	0	
Bounded Below	51	
Bounded Below and Above	0	
Free	0	
Fixed	12	
Number of Constraints	42	
Linear LE (\leq)	24	
Linear EQ ($=$)	18	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	255	
Solution Summary		
Solver	Dual Simplex	
Objective Function	TotalProductiveCapacity	
Solution Status	Optimal	
Objective Value	2141.8751967	
Iterations	41	
Primal Infeasibility	5.684342E-14	
Dual Infeasibility	0	
Bound Infeasibility	7.105427E-15	

Figure 9.2 continued

[1]	[2]	Production	Stock	Extra Capacity	Productive Capacity
coal	0	0.000	150.0000		
coal	1	260.403	0.0000	0.0	300.0
coal	2	293.406	0.0000	0.0	300.0
coal	3	300.000	0.0000	0.0	300.0
coal	4	17.949	148.4480	189.2	489.2
coal	5	166.397	0.0000	1022.7	1511.9
coal	6	166.397	-0.0000	0.0	1511.9
coal	7			0.0	
steel	0	0.000	80.0000		
steel	1	135.342	12.2811	0.0	350.0
steel	2	181.660	0.0000	0.0	350.0
steel	3	193.090	0.0000	0.0	350.0
steel	4	105.668	0.0000	0.0	350.0
steel	5	105.668	0.0000	0.0	350.0
steel	6	105.668	-0.0000	0.0	350.0
steel	7			0.0	
transport	0	0.000	100.0000		
transport	1	140.722	6.2408	0.0	280.0
transport	2	200.580	0.0000	0.0	280.0
transport	3	267.152	0.0000	0.0	280.0
transport	4	92.308	0.0000	0.0	280.0
transport	5	92.308	0.0000	0.0	280.0
transport	6	92.308	0.0000	0.0	280.0
transport	7			0.0	
		Manpower_ con.BODY			
	[1]				
	1		224.99		
	2		270.66		
	3		367.04		
	4		470.00		
	5		150.00		
	6		150.00		

For Problem2, the right-hand side of each Continuity_con constraint changes to 0:

```

use problem Problem2;
for {i in INDUSTRIES, year in YEARS} do;
    Continuity_con[i,year].lb = 0;
    Continuity_con[i,year].ub = 0;
end;
solve;
print Production Stock ExtraCapacity ProductiveCapacity Manpower_con.body;

```

Figure 9.3 shows the output from the linear programming solver for Problem2.

Figure 9.3 Output from Linear Programming Solver for Problem2

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProduction
Objective Type	Linear
Number of Variables	63
Bounded Above	0
Bounded Below	51
Bounded Below and Above	0
Free	0
Fixed	12
Number of Constraints	42
Linear LE (\leq)	24
Linear EQ ($=$)	18
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	255
Solution Summary	
Solver	Dual Simplex
Objective Function	TotalProduction
Solution Status	Optimal
Objective Value	2618.5791147
Iterations	40
Primal Infeasibility	1.136868E-13
Dual Infeasibility	1.886512E-16
Bound Infeasibility	5.243478E-15

Figure 9.3 continued

[1]	[2]	Production	Stock	Extra Capacity	Productive Capacity
coal	0	0.000	150.000		
coal	1	300.000	20.110	0.0000	300.00
coal	2	315.323	131.554	15.3230	315.32
coal	3	430.505	0.000	115.1817	430.50
coal	4	430.505	0.000	-0.0000	430.50
coal	5	430.505	0.000	0.0000	430.50
coal	6	166.397	324.108	0.0000	430.50
coal	7			0.0000	
steel	0	0.000	80.000		
steel	1	86.730	11.532	0.0000	350.00
steel	2	155.337	0.000	0.0000	350.00
steel	3	182.867	0.000	0.0000	350.00
steel	4	359.402	0.000	9.4023	359.40
steel	5	359.402	176.535	0.0000	359.40
steel	6	105.668	490.269	0.0000	359.40
steel	7			0.0000	
transport	0	0.000	100.000		
transport	1	141.312	0.000	0.0000	280.00
transport	2	198.388	0.000	0.0000	280.00
transport	3	225.918	0.000	0.0000	280.00
transport	4	519.383	0.000	239.3826	519.38
transport	5	519.383	293.465	0.0000	519.38
transport	6	92.308	750.540	0.0000	519.38
transport	7			0.0000	

	Manpower_ con.BODY
[1]	
1	240.41
2	321.55
3	384.17
4	470.00
5	470.00
6	150.00

For Problem3, the right-hand side of each Continuity_con[i,year] constraint changes back to *demand[i]*:

```

use problem Problem3;
for {i in INDUSTRIES, year in YEARS} do;
    Continuity_con[i,year].lb = demand[i];
    Continuity_con[i,year].ub = demand[i];
end;
solve;
print Production Stock ExtraCapacity ProductiveCapacity Manpower_con.body;
quit;

```

Figure 9.4 shows the output from the linear programming solver for Problem3.

Figure 9.4 Output from Linear Programming Solver for Problem3

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalManpower
Objective Type	Linear
Number of Variables	63
Bounded Above	0
Bounded Below	51
Bounded Below and Above	0
Free	0
Fixed	12
Number of Constraints	36
Linear LE (\leq)	18
Linear EQ ($=$)	18
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	219
Solution Summary	
Solver	Dual Simplex
Objective Function	TotalManpower
Solution Status	Optimal
Objective Value	2450.0266228
Iterations	53
Primal Infeasibility	5.684342E-14
Dual Infeasibility	0
Bound Infeasibility	2.4869E-14

Figure 9.4 continued

[1]	[2]	Production	Stock	Extra Capacity	Productive Capacity
coal	0	0.00	150.0000		
coal	1	251.79	0.0000	0.0000	300.00
coal	2	316.02	0.0000	16.0152	316.02
coal	3	319.83	0.0000	3.8168	319.83
coal	4	366.35	0.0000	46.5177	366.35
coal	5	859.36	0.0000	493.0099	859.36
coal	6	859.36	460.2080	0.0000	859.36
coal	7			0.0000	
steel	0	0.00	80.0000		
steel	1	134.79	11.0280	0.0000	350.00
steel	2	175.04	0.0000	0.0000	350.00
steel	3	224.06	0.0000	0.0000	350.00
steel	4	223.14	0.0000	0.0000	350.00
steel	5	220.04	0.0000	0.0000	350.00
steel	6	350.00	0.0000	0.0000	350.00
steel	7			0.0000	
transport	0	0.00	100.0000		
transport	1	143.56	4.2472	0.0000	280.00
transport	2	181.68	0.0000	0.0000	280.00
transport	3	280.00	0.0000	0.0000	280.00
transport	4	279.07	0.0000	0.0000	280.00
transport	5	275.98	0.0000	0.0000	280.00
transport	6	195.54	-0.0000	0.0000	280.00
transport	7			0.0000	
		Manpower_ con.BODY			
		[1]			
		1	226.63		
		2	279.98		
		3	333.73		
		4	539.77		
		5	636.82		
		6	659.72		

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming (dynamic Leontief input-output model)
- numeric and string index sets
- set operator UNION
- multiple input data sets
- reading dense two-dimensional data
- IMPVAR statement
- **.lb** and **.ub** variable suffixes
- IF-THEN/ELSE expression
- using the **.lb** and **.ub** constraint suffixes to modify the right-hand side of a constraint
- multiple objectives
- PROBLEM and USE PROBLEM statements
- **.body** constraint suffix

Chapter 10

Decentralization: How to Disperse Offices from the Capital

Contents

Problem Statement	119
Mathematical Programming Formulation	120
Input Data	122
PROC OPTMODEL Statements and Output	123
Features Demonstrated	129

Problem Statement

A large company wishes to move some of its departments out of London.¹ There are benefits to be derived from doing this (cheaper housing, government incentives, easier recruitment, etc.) which have been costed. Also, however, there will be greater costs of communication between departments. These have also been costed for all possible locations of each department.

Where should each department be located so as to minimize overall yearly cost?

The company comprises five departments (A, B, C, D, E). The possible cities for relocation are Bristol and Brighton, or a department may be kept in London. None of these cities (including London) may be the location for more than three of the departments.

Benefits to be derived from each relocation are given below (in thousands of pounds per year):

	A	B	C	D	E
Bristol	10	15	10	20	5
Brighton	10	20	15	15	15

Communication costs are of the form $C_{ik}D_{jl}$ where C_{ik} is the quantity of communication between departments i and k per year and D_{jl} is the cost per unit of communication between cities j and l . C_{ik} and D_{jl} are given by the tables below:

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, p. 242).

Quantities of communication C_{ik} (in thousands of units)				
	A	B	C	D
A		0.0	1.0	1.5
B			1.4	1.2
C				0.0
D				

Costs per unit of communication D_{jl} (in £)			
	Bristol	Brighton	London
Bristol	5	14	13
Brighton		5	9
London			10

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- dept, $i, k \in \text{DEPTS}$
- city, $j, l \in \text{CITIES}$
- $(i, j, k, l) \in \text{IJKL} = \{i \in \text{DEPTS}, j \in \text{CITIES}, k \in \text{DEPTS}, l \in \text{CITIES} : i < k\}$

Parameters

Table 10.1 shows the parameters used in this example.

Table 10.1 Parameters

Parameter Name	Interpretation
$benefit[i, j]$	Yearly benefit derived from relocating department i to city j
$comm[i, k]$	Quantity of communication between departments i and k per year
$cost[j, l]$	Cost per unit of communication between cities j and l
max_num_depts	Upper bound on number of departments per city

Variables

Table 10.2 shows the variables used in this example.

Table 10.2 Variables

Variable Name	Interpretation
Assign[i,j]	1 if department i is assigned to city j ; 0 otherwise
Product[i,j,k,l]	Assign[i,j] · Assign[k,l]
TotalBenefit	Total yearly benefit of all relocations
TotalCost	Total yearly communication costs of all relocations

Objective

The objective is to maximize the following quadratic function:

$$\text{NetBenefit} = \sum_{i \in \text{DEPTS}} \sum_{j \in \text{CITIES}} \text{benefit}[i,j] \cdot \text{Assign}[i,j] - \sum_{(i,j,k,l) \in \text{IJKL}} \text{comm}[i,k] \cdot \text{cost}[j,l] \cdot \text{Product}[i,j,k,l]$$

where

$$\text{Product}[i,j,k,l] = \text{Assign}[i,j] \cdot \text{Assign}[k,l]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- $\text{TotalBenefit} = \sum_{i \in \text{DEPTS}} \sum_{j \in \text{CITIES}} \text{benefit}[i,j] \cdot \text{Assign}[i,j]$
- $\text{TotalCost} = \sum_{(i,j,k,l) \in \text{IJKL}} \text{comm}[i,k] \cdot \text{cost}[j,l] \cdot \text{Product}[i,j,k,l]$
- for $\text{dept} \in \text{DEPTS}$,

$$\sum_{\text{city} \in \text{CITIES}} \text{Assign}[\text{dept}, \text{city}] = 1$$

- for $\text{city} \in \text{CITIES}$,

$$\sum_{\text{dept} \in \text{DEPTS}} \text{Assign}[\text{dept}, \text{city}] \leq \text{max_num_depts}$$

- for $(i, j, k, l) \in \text{IJKL}$,

$$\text{Assign}[i,j] + \text{Assign}[k,l] - 1 \leq \text{Product}[i,j,k,l]$$
- for $(i, j, k, l) \in \text{IJKL}$,

$$\text{Product}[i,j,k,l] \leq \text{Assign}[i,j]$$
- for $(i, j, k, l) \in \text{IJKL}$,

$$\text{Product}[i,j,k,l] \leq \text{Assign}[k,l]$$
- for $i \in \text{DEPTS}$ and $k \in \text{DEPTS}$ and $l \in \text{CITIES}$ such that $i < k$,

$$\sum_{(i,j,k,l) \in \text{IJKL}} \text{Product}[i,j,k,l] = \text{Assign}[k,l]$$
- for $k \in \text{DEPTS}$ and $i \in \text{DEPTS}$ and $j \in \text{CITIES}$ such that $i < k$,

$$\sum_{(i,j,k,l) \in \text{IJKL}} \text{Product}[i,j,k,l] = \text{Assign}[i,j]$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data dept_data;
    input dept $ @@;
    datalines;
A B C D E
;

data city_data;
    input city $;
    datalines;
Bristol
Brighton
London
;

data benefit_data;
    input city $ A B C D E;
    datalines;
Bristol 10 15 10 20 5
Brighton 10 20 15 15 15
;

data comm_data;
    input i $ j $ comm;
    datalines;
```

```

A B 0.0
A C 1.0
A D 1.5
A E 0.0
B C 1.4
B D 1.2
B E 0.0
C D 0.0
C E 2.0
D E 0.7
;

data cost_data;
    input i $ j $ cost;
    datalines;
Bristol Bristol 5
Bristol Brighton 14
Bristol London 13
Brighton Brighton 5
Brighton London 9
London London 10
;

%let max_num_depts = 3;

```

PROC OPTMODEL Statements and Output

The first two READ DATA statements populate the DEPTS and CITIES index sets:

```

proc optmodel;
    set <str> DEPTS;
    read data dept_data into DEPTS=[dept];

    set <str> CITIES;
    read data city_data into CITIES=[city];

```

The following READ DATA statement reads dense two-dimensional data, as in previous examples, with an initial value of 0 for *benefit* to account for the possibility of staying in London:

```

num benefit {DEPTS, CITIES} init 0;
read data benefit_data into [city] {dept in DEPTS} <benefit[dept,city]=col(dept)>;
print benefit;

```

Note that this READ DATA statement does not repopulate CITIES using the following:

```
CITIES=[city]
```

Doing so would have removed London from the CITIES index set, because London does not appear in the benefit_data data set. [Figure 10.1](#) shows the resulting values of *benefit*.

Figure 10.1 *benefit* Parameter

The OPTMODEL Procedure			
benefit			
	Brighton	Bristol	London
A	10	10	0
B	20	15	0
C	15	10	0
D	15	20	0
E	15	5	0

The following NUM and assignment statements read an upper triangular matrix and use these values to populate the lower triangular part. The INIT option initializes the parameter *comm* to a missing value, and the body of the FOR loop replaces each missing value with the value obtained by reflection across the main diagonal.

```
num comm {DEPTS, DEPTS} init .;
read data comm_data into [i j] comm;
for {i in DEPTS, j in DEPTS} do;
  if i = j then comm[i,j] = 0;
  else if comm[i,j] = . then comm[i,j] = comm[j,i];
end;
print comm;
```

Figure 10.2 shows the resulting values of *comm*.

Figure 10.2 *comm* Parameter

comm					
	A	B	C	D	E
A	0.0	0.0	1.0	1.5	0.0
B	0.0	0.0	1.4	1.2	0.0
C	1.0	1.4	0.0	0.0	2.0
D	1.5	1.2	0.0	0.0	0.7
E	0.0	0.0	2.0	0.7	0.0

Similar statements are used to populate the *cost* parameter, but instead the main diagonal is read from the *cost_data* data set:

```
num cost {CITIES, CITIES} init .;
read data cost_data into [i j] cost;
for {i in CITIES, j in CITIES: cost[i,j] = .}
  cost[i,j] = cost[j,i];
print cost;
```

Figure 10.3 shows the resulting values of *cost*.

Figure 10.3 cost Parameter

	cost		
	Brighton	Bristol	London
Brighton	5	14	9
Bristol	14	5	13
London	9	13	10

The following declarations are straightforward:

```
var Assign {DEPTS, CITIES} binary;

set IJKL = {i in DEPTS, j in CITIES, k in DEPTS, l in CITIES: i < k};
var Product {IJKL} binary;

impvar TotalBenefit = sum {i in DEPTS, j in CITIES} benefit[i,j] * Assign[i,j];
impvar TotalCost = sum {<i,j,k,l> in IJKL} comm[i,k] * cost[j,l] * Product[i,j,k,l];
max NetBenefit = TotalBenefit - TotalCost;

con Assign_dept {dept in DEPTS}:
    sum {city in CITIES} Assign[dept,city] = 1;

con Cardinality {city in CITIES}:
    sum {dept in DEPTS} Assign[dept,city] <= &max_num_depts;
```

The following CON statement enforces the rule that $\text{Assign}[i,j] = 1$ and $\text{Assign}[k,l] = 1$ together imply $\text{Product}[i,j,k,l] = 1$:

```
con Product_def {<i,j,k,l> in IJKL}:
    Assign[i,j] + Assign[k,l] - 1 <= Product[i,j,k,l];
```

The following two CON statements enforce the converse rule that $\text{Product}[i,j,k,l] = 1$ implies both $\text{Assign}[i,j] = 1$ and $\text{Assign}[k,l] = 1$:

```
con Product_def2 {<i,j,k,l> in IJKL}:
    Product[i,j,k,l] <= Assign[i,j];
con Product_def3 {<i,j,k,l> in IJKL}:
    Product[i,j,k,l] <= Assign[k,l];
```

Because of the maximization objective in this problem, these two constraints could be omitted and would still be satisfied by an optimal solution. But including them can sometimes reduce the number of simplex iterations and branch-and-bound nodes.

```
solve;

print TotalBenefit TotalCost;
print Assign;
```

Figure 10.4 shows the output from the mixed integer linear programming solver.

Figure 10.4 Output from Mixed Integer Linear Programming Solver

Problem Summary			
Objective Sense	Maximization		
Objective Function	NetBenefit		
Objective Type	Linear		
Number of Variables	105		
Bounded Above	0		
Bounded Below	0		
Bounded Below and Above	105		
Free	0		
Fixed	0		
Binary	105		
Integer	0		
Number of Constraints	278		
Linear LE (<=)	273		
Linear EQ (=)	5		
Linear GE (>=)	0		
Linear Range	0		
Constraint Coefficients	660		
Solution Summary			
Solver	MILP		
Objective Function	NetBenefit		
Solution Status	Optimal		
Objective Value	14.9		
Iterations	143		
Best Bound	.		
Nodes	1		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	5.0005E-9		
Bound Infeasibility	5.0004999E-9		
Integer Infeasibility	5.0004999E-9		
Total Benefit	Total Cost		
80	65.1		
Assign			
	Brighton	Bristol	London
A	0	1	0
B	1	0	0
C	1	0	0
D	0	1	0
E	1	0	0

An alternative “compact linearization” formulation involves fewer constraints (Liberti 2007). The following DROP statement removes three families of constraints from the original formulation:

```
drop Product_def Product_def2 Product_def3;
```

The following two CON statements declare two new constraints that enforce the desired relationship between the Product and Assign variables:

```
con Product_def4 {i in DEPTS, k in DEPTS, l in CITIES: i < k}:
    sum {(i), j, (k), (l)> in IJKL} Product[i, j, k, l] = Assign[k, l];
con Product_def5 {k in DEPTS, i in DEPTS, j in CITIES: i < k}:
    sum {(i), (j), (k), l> in IJKL} Product[i, j, k, l] = Assign[i, j];

solve;

print TotalBenefit TotalCost;
print Assign;
quit;
```

Figure 10.5 shows the output from the mixed integer linear programming solver for the compact linearization formulation.

Figure 10.5 Output from Mixed Integer Linear Programming Solver (Compact Linearization)

Problem Summary	
Objective Sense	Maximization
Objective Function	NetBenefit
Objective Type	Linear
Number of Variables	105
Bounded Above	0
Bounded Below	0
Bounded Below and Above	105
Free	0
Fixed	0
Binary	105
Integer	0
Number of Constraints	68
Linear LE (<=)	3
Linear EQ (=)	65
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	270

Figure 10.5 *continued*

Solution Summary			
Solver	MILP		
Objective Function	NetBenefit		
Solution Status	Optimal		
Objective Value	14.90000015		
Iterations	49		
Best Bound	.		
Nodes	1		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	6.077471E-17		
Bound Infeasibility	5E-9		
Integer Infeasibility	9.9999999E-9		
Total	Total		
Benefit	Cost		
80	65.1		
Assign			
Brighton	Bristol	London	
A	0.0E+00	1.0E+00	0.0E+00
B	1.0E+00	0.0E+00	0.0E+00
C	1.0E+00	0.0E+00	0.0E+00
D	5.0E-09	1.0E+00	0.0E+00
E	1.0E+00	5.0E-09	0.0E+00

As expected, this solution agrees with the solution reported in Figure 10.4 for the original formulation.

In both formulations, the Product variable can be relaxed to be nonnegative instead of binary. The integrality of Assign, together with the various Product_def* constraints, automatically implies integrality of Product. For real-world problems, you should try both ways to determine which alternative performs better in specific cases.

Similarly, the compact formulation is weaker but contains fewer constraints than the original formulation. The net effect on total solve time is difficult to predict. For real-world problems, you should try both formulations.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (quadratic assignment problem)
- numeric and string index sets
- reading dense two-dimensional data
- reading sparse (upper triangular) two-dimensional data
- multiple input data sets
- set of tuples
- INIT option
- IMPVAR statement
- product of two binary variables
- DROP statement
- compact linearization

Chapter 11

Curve Fitting: Fitting a Curve to a Set of Data Points

Contents

Problem Statement	131
Mathematical Programming Formulation	132
Input Data	133
PROC OPTMODEL Statements and Output	134
Features Demonstrated	145

Problem Statement

A quantity y is known to depend upon another quantity x .¹ A set of corresponding values has been collected for x and y and is presented in Table 11.

x	0.0	0.5	1.0	1.5	1.9	2.5	3.0	3.5	4.0	4.5
y	1.0	0.9	0.7	1.5	2.0	2.4	3.2	2.0	2.7	3.5
x	5.0	5.5	6.0	6.6	7.0	7.6	8.5	9.0	10.0	
y	1.0	4.0	3.6	2.7	5.7	4.6	6.0	6.8	7.3	

- (1) Fit the ‘best’ straight line $y = bx + a$ to this set of data points. The objective is to minimize the sum of *absolute deviations* of each observed value of y from the value predicted by the linear relationship.
- (2) Fit the ‘best’ straight line where the objective is to minimize the *maximum deviation* of all the observed values of y from the value predicted by the linear relationship.
- (3) Fit the ‘best’ quadratic curve $y = cx^2 + bx + a$ to this set of data points using the same objectives as in (1) and (2).

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 242–243).

Mathematical Programming Formulation

Index Sets and Their Members

The following index set and its members are used in this example:

- $i \in \text{POINTS}$

Parameters

Table 11.1 shows the parameters used in this example.

Table 11.1 Parameters

Parameter Name	Interpretation
$x[i]$	x -coordinate of input data point i
$y[i]$	y -coordinate of input data point i
$order$	Order of polynomial regression curve
sum_abs_dev	Sum of absolute deviations between predicted and observed values
max_abs_dev	Maximum of absolute deviations between predicted and observed values

Variables

Table 11.2 shows the variables used in this example.

Table 11.2 Variables

Variable Name	Interpretation
$Beta[k]$	Regression coefficient of order k
$Estimate[i]$	Predicted value of y for data point i
$Surplus[i]$	$\max\{Estimate[i] - y[i], 0\}$
$Slack[i]$	$\max\{y[i] - Estimate[i], 0\}$
$MinMax$	$\max_{i \in \text{POINTS}} Estimate[i] - y[i] $

Objectives

The first objective is to minimize the following (nonlinear, nondifferentiable) L_1 norm function:

$$\text{Objective1} = \sum_{i \in \text{POINTS}} |Estimate[i] - y[i]|$$

The second objective is to minimize the following (nonlinear, nondifferentiable) L_∞ norm function:

$$\text{Objective2} = \max_{i \in \text{POINTS}} |\text{Estimate}[i] - y[i]|$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $i \in \text{POINTS}$,

$$\text{Estimate}[i] = \text{Beta}[0] + \sum_{k=1}^{\text{order}} \text{Beta}[k] \cdot x[i]^k$$

- for $i \in \text{POINTS}$,

$$\text{Estimate}[i] - \text{Surplus}[i] + \text{Slack}[i] = y[i]$$

- for $i \in \text{POINTS}$,

$$\text{MinMax} \geq \text{Surplus}[i] + \text{Slack}[i]$$

Input Data

The following data set contains the input data that are used in this example:

```
data xy_data;
  input x y;
  datalines;
0.0 1.0
0.5 0.9
1.0 0.7
1.5 1.5
1.9 2.0
2.5 2.4
3.0 3.2
3.5 2.0
4.0 2.7
4.5 3.5
5.0 1.0
5.5 4.0
6.0 3.6
6.6 2.7
7.0 5.7
7.6 4.6
```

```

8.5 6.0
9.0 6.8
10.0 7.3
;

```

PROC OPTMODEL Statements and Output

The following PROC OPTMODEL statements declare an index set and parameters and then read the input data:

```

proc optmodel;
  set POINTS;
  num x {POINTS};
  num y {POINTS};
  read data xy_data into POINTS=[_N_] x y;

```

The following NUM statement declares the *order* parameter, which is later set to 1 for Problems (1) and (2) and set to 2 for Problem (3):

```

num order;
var Beta {0..order};
impvar Estimate {i in POINTS} = Beta[0] + sum {k in 1..order} Beta[k] * x[i]^k;

```

The following statements encode the linearization of the L_1 norm:

```

var Surplus {POINTS} >= 0;
var Slack {POINTS} >= 0;
min Objective1 = sum {i in POINTS} (Surplus[i] + Slack[i]);
con Abs_dev_con {i in POINTS}:
  Estimate[i] - Surplus[i] + Slack[i] = y[i];

```

The following statements (which are not used but are shown here for comparison) encode an alternative linearization of the L_1 norm that requires half as many variables but twice as many constraints:

```

var AbsDeviation {POINTS} >= 0;
min Objective1 = sum {i in POINTS} AbsDeviation[i];
con Abs_dev_con1 {i in POINTS}:
  AbsDeviation[i] >= Estimate[i] - y[i];
con Abs_dev_con2 {i in POINTS}:
  AbsDeviation[i] >= y[i] - Estimate[i];

```

The following additional declarations encode the linearization of the L_∞ norm:

```

var MinMax;
min Objective2 = MinMax;
con MinMax_con {i in POINTS}:
  MinMax >= Surplus[i] + Slack[i];

```

The following statements (which are not used but match the formulation in Williams) encode an alternative linearization of the L_∞ norm that requires the same number of variables but twice as many constraints:


```

var MinMax;
min Objective2 = MinMax;
con MinMax_con1 {i in POINTS}:
    MinMax >= Surplus[i];
con MinMax_con2 {i in POINTS}:
    MinMax >= Slack[i];

```

The following NUM statements use the ABS function, the `.sol` variable suffix, and the MAX aggregation operator to compute the two norms from the optimal solution:

```

num sum_abs_dev = sum {i in POINTS} abs(Estimate[i].sol - y[i]);
num max_abs_dev = max {i in POINTS} abs(Estimate[i].sol - y[i]);

```

The following PROBLEM statement specifies the variables, objective, and constraints for L_1 minimization:

```

problem L1 include
    Beta Surplus Slack
    Objective1
    Abs_dev_con;

```

The following PROBLEM statement specifies the additional variables, objective, and constraints for L_∞ minimization:

```

problem Linf from L1 include
    MinMax
    Objective2
    MinMax_con;

```

The following statements specify a straight line fit, switch focus to problem L1, solve Problem (1), print the results, and store the y -values that are predicted by the optimal solution:

```

order = 1;
use problem L1;
solve;
print sum_abs_dev max_abs_dev;
print Beta;
print x y Estimate Surplus Slack;
create data sol_data1 from [POINTS] x y Estimate;

```

Figure 11.1 shows the output from the linear programming solver for Problem (1).

Figure 11.1 Output from Linear Programming Solver for Problem (1)

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	Objective1
Objective Type	Linear
Number of Variables	40
Bounded Above	0
Bounded Below	38
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	19
Linear LE (<=)	0
Linear EQ (=)	19
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	75
Solution Summary	
Solver	Dual Simplex
Objective Function	Objective1
Solution Status	Optimal
Objective Value	11.46625
Iterations	22
Primal Infeasibility	8.881784E-16
Dual Infeasibility	0
Bound Infeasibility	0
sum_	max_
abs_	abs_
dev	dev
11.466	2.7688
[1]	Beta
0	0.58125
1	0.63750

Figure 11.1 *continued*

	[1]	x	y	Estimate	Surplus	Slack
	1	0.0	1.0	0.58125	0.00000	0.41875
	2	0.5	0.9	0.90000	0.00000	0.00000
	3	1.0	0.7	1.21875	0.51875	0.00000
	4	1.5	1.5	1.53750	0.03750	0.00000
	5	1.9	2.0	1.79250	0.00000	0.20750
	6	2.5	2.4	2.17500	0.00000	0.22500
	7	3.0	3.2	2.49375	0.00000	0.70625
	8	3.5	2.0	2.81250	0.81250	0.00000
	9	4.0	2.7	3.13125	0.43125	0.00000
	10	4.5	3.5	3.45000	0.00000	0.05000
	11	5.0	1.0	3.76875	2.76875	0.00000
	12	5.5	4.0	4.08750	0.08750	0.00000
	13	6.0	3.6	4.40625	0.80625	0.00000
	14	6.6	2.7	4.78875	2.08875	0.00000
	15	7.0	5.7	5.04375	0.00000	0.65625
	16	7.6	4.6	5.42625	0.82625	0.00000
	17	8.5	6.0	6.00000	0.00000	0.00000
	18	9.0	6.8	6.31875	0.00000	0.48125
	19	10.0	7.3	6.95625	0.00000	0.34375

The following statements switch focus to problem Linf, solve Problem (2), print the results, and store the y -values that are predicted by the optimal solution:

```

use problem Linf;
solve;
print sum_abs_dev max_abs_dev;
print Beta;
print x y Estimate Surplus Slack;
create data sol_data2 from [POINTS] x y Estimate;

```

Figure 11.2 shows the output from the linear programming solver for Problem (2).

Figure 11.2 Output from Linear Programming Solver for Problem (2)

Problem Summary	
Objective Sense	Minimization
Objective Function	Objective2
Objective Type	Linear
Number of Variables	41
Bounded Above	0
Bounded Below	38
Bounded Below and Above	0
Free	3
Fixed	0
Number of Constraints	38
Linear LE (\leq)	0
Linear EQ ($=$)	19
Linear GE (\geq)	19
Linear Range	0
Constraint Coefficients	132
Solution Summary	
Solver	Dual Simplex
Objective Function	Objective2
Solution Status	Optimal
Objective Value	1.725
Iterations	23
Primal Infeasibility	8.881784E-16
Dual Infeasibility	0
Bound Infeasibility	0
sum_	max_
abs_	abs_
dev	dev
19.95	1.725
[1]	Beta
0	-0.400
1	0.625

Figure 11.2 *continued*

	[1]	x	y	Estimate	Surplus	Slack
	1	0.0	1.0	-0.4000	0.000	1.4000
	2	0.5	0.9	-0.0875	0.000	0.9875
	3	1.0	0.7	0.2250	0.000	0.4750
	4	1.5	1.5	0.5375	0.000	0.9625
	5	1.9	2.0	0.7875	0.000	1.2125
	6	2.5	2.4	1.1625	0.000	1.2375
	7	3.0	3.2	1.4750	0.000	1.7250
	8	3.5	2.0	1.7875	0.000	0.2125
	9	4.0	2.7	2.1000	0.000	0.6000
	10	4.5	3.5	2.4125	0.000	1.0875
	11	5.0	1.0	2.7250	1.725	0.0000
	12	5.5	4.0	3.0375	0.000	0.9625
	13	6.0	3.6	3.3500	0.000	0.2500
	14	6.6	2.7	3.7250	1.025	0.0000
	15	7.0	5.7	3.9750	0.000	1.7250
	16	7.6	4.6	4.3500	0.000	0.2500
	17	8.5	6.0	4.9125	0.000	1.0875
	18	9.0	6.8	5.2250	0.000	1.5750
	19	10.0	7.3	5.8500	0.000	1.4500

The following statements specify a quadratic curve fit, solve both parts of Problem (3), print the results, and store the y -values that are predicted by the optimal solutions:

```

order = 2;
use problem L1;
solve;
print sum_abs_dev max_abs_dev;
print Beta;
print x y Estimate Surplus Slack;
create data sol_data3 from [POINTS] x y Estimate;

use problem Linf;
solve;
print sum_abs_dev max_abs_dev;
print Beta;
print x y Estimate Surplus Slack;
create data sol_data4 from [POINTS] x y Estimate;
quit;

```

Figure 11.3 shows the output from the linear programming solver for the first part of Problem (3).

Figure 11.3 Output from Linear Programming Solver for First Part of Problem (3)

Problem Summary		
Objective Sense	Minimization	
Objective Function	Objective1	
Objective Type	Linear	
Number of Variables	41	
Bounded Above	0	
Bounded Below	38	
Bounded Below and Above	0	
Free	3	
Fixed	0	
Number of Constraints	19	
Linear LE (<=)	0	
Linear EQ (=)	19	
Linear GE (>=)	0	
Linear Range	0	
Constraint Coefficients	93	

Solution Summary		
Solver	Dual Simplex	
Objective Function	Objective1	
Solution Status	Optimal	
Objective Value	10.458964706	
Iterations	21	
Primal Infeasibility	8.881784E-16	
Dual Infeasibility	0	
Bound Infeasibility	0	

sum_	max_
abs_	abs_
dev	dev
10.459	2.298

[1]	Beta
0	0.982353
1	0.294510
2	0.033725

Figure 11.3 continued

[1]	x	y	Estimate	Surplus	Slack
1	0.0	1.0	0.98235	0.00000	0.017647
2	0.5	0.9	1.13804	0.23804	0.000000
3	1.0	0.7	1.31059	0.61059	0.000000
4	1.5	1.5	1.50000	0.00000	0.000000
5	1.9	2.0	1.66367	0.00000	0.336329
6	2.5	2.4	1.92941	0.00000	0.470588
7	3.0	3.2	2.16941	0.00000	1.030588
8	3.5	2.0	2.42627	0.42627	0.000000
9	4.0	2.7	2.70000	0.00000	0.000000
10	4.5	3.5	2.99059	0.00000	0.509412
11	5.0	1.0	3.29804	2.29804	0.000000
12	5.5	4.0	3.62235	0.00000	0.377647
13	6.0	3.6	3.96353	0.36353	0.000000
14	6.6	2.7	4.39520	1.69520	0.000000
15	7.0	5.7	4.69647	0.00000	1.003529
16	7.6	4.6	5.16861	0.56861	0.000000
17	8.5	6.0	5.92235	0.00000	0.077647
18	9.0	6.8	6.36471	0.00000	0.435294
19	10.0	7.3	7.30000	0.00000	0.000000

Figure 11.4 shows the output from the linear programming solver for the second part of Problem (3).

Figure 11.4 Output from Linear Programming Solver for Second Part of Problem (3)

Problem Summary	
Objective Sense	Minimization
Objective Function	Objective2
Objective Type	Linear
Number of Variables	42
Bounded Above	0
Bounded Below	38
Bounded Below and Above	0
Free	4
Fixed	0
Number of Constraints	38
Linear LE (<=)	0
Linear EQ (=)	19
Linear GE (>=)	19
Linear Range	0
Constraint Coefficients	150

Figure 11.4 *continued*

Solution Summary					
Solver		Dual Simplex			
Objective Function		Objective2			
Solution Status		Optimal			
Objective Value		1.475			
Iterations		25			
Primal Infeasibility		8.881784E-16			
Dual Infeasibility		0			
Bound Infeasibility		0			
sum_		max_			
abs_		abs_			
dev		dev			
16.757		1.475			
[1]		Beta			
0		2.475			
1		-0.625			
2		0.125			
[1]	x	y	Estimate	Surplus	Slack
1	0.0	1.0	2.4750	1.47500	0.00000
2	0.5	0.9	2.1938	1.29375	0.00000
3	1.0	0.7	1.9750	1.27500	0.00000
4	1.5	1.5	1.8188	0.31875	0.00000
5	1.9	2.0	1.7388	0.00000	0.26125
6	2.5	2.4	1.6938	0.00000	0.70625
7	3.0	3.2	1.7250	0.00000	1.47500
8	3.5	2.0	1.8188	0.00000	0.18125
9	4.0	2.7	1.9750	0.00000	0.72500
10	4.5	3.5	2.1938	0.00000	1.30625
11	5.0	1.0	2.4750	1.47500	0.00000
12	5.5	4.0	2.8188	0.00000	1.18125
13	6.0	3.6	3.2250	0.00000	0.37500
14	6.6	2.7	3.7950	1.09500	0.00000
15	7.0	5.7	4.2250	0.00000	1.47500
16	7.6	4.6	4.9450	0.34500	0.00000
17	8.5	6.0	6.1938	0.19375	0.00000
18	9.0	6.8	6.9750	0.17500	0.00000
19	10.0	7.3	8.7250	1.42500	0.00000

You can find a higher-order polynomial fit simply by increasing the value of the *order* parameter. The dimensions of the Beta and Estimate variables are automatically updated when *order* changes.

The following PROC SGPLOT statements use the output data sets created by PROC OPTMODEL to display the results from Problems (1) and (2) in one plot:


```

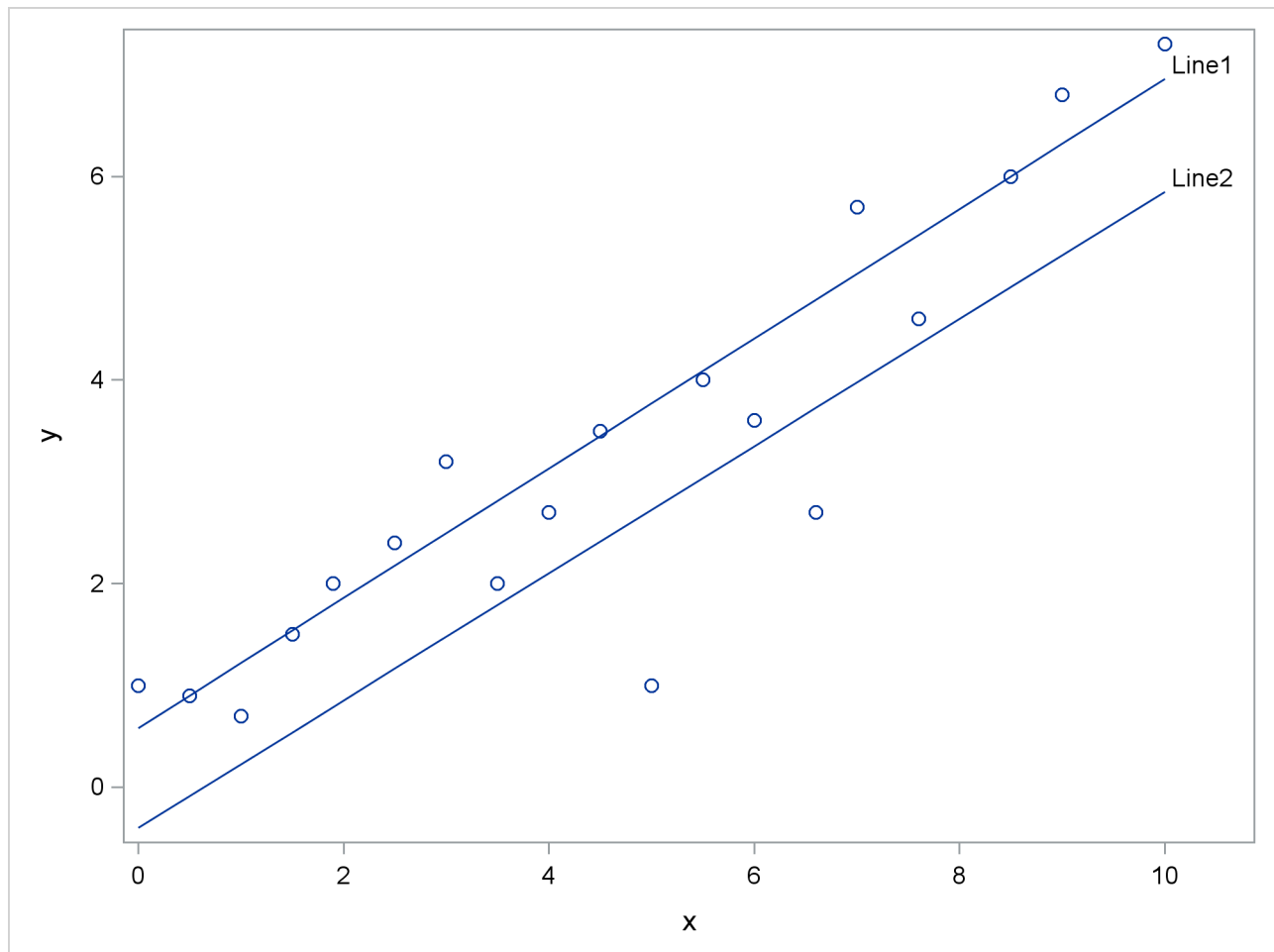
data plot1;
  merge sol_data1(rename=(Estimate=Line1)) sol_data2(rename=(Estimate=Line2));
run;

proc sgplot data=plot1;
  scatter x=x y=y;
  series x=x y=Line1 / curvelabel;
  series x=x y=Line2 / curvelabel;
run;

```

Figure 11.5 shows the regression lines for Problems (1) and (2), as on page 319 of Williams.

Figure 11.5 Regression Lines for Problems (1) and (2)



The following PROC SGPLOT statements use the output data sets created by PROC OPTMODEL to display the results from both parts of Problem (3) in one plot:

```

data plot2;
  merge sol_data3(rename=(Estimate=Curve1)) sol_data4(rename=(Estimate=Curve2));
run;

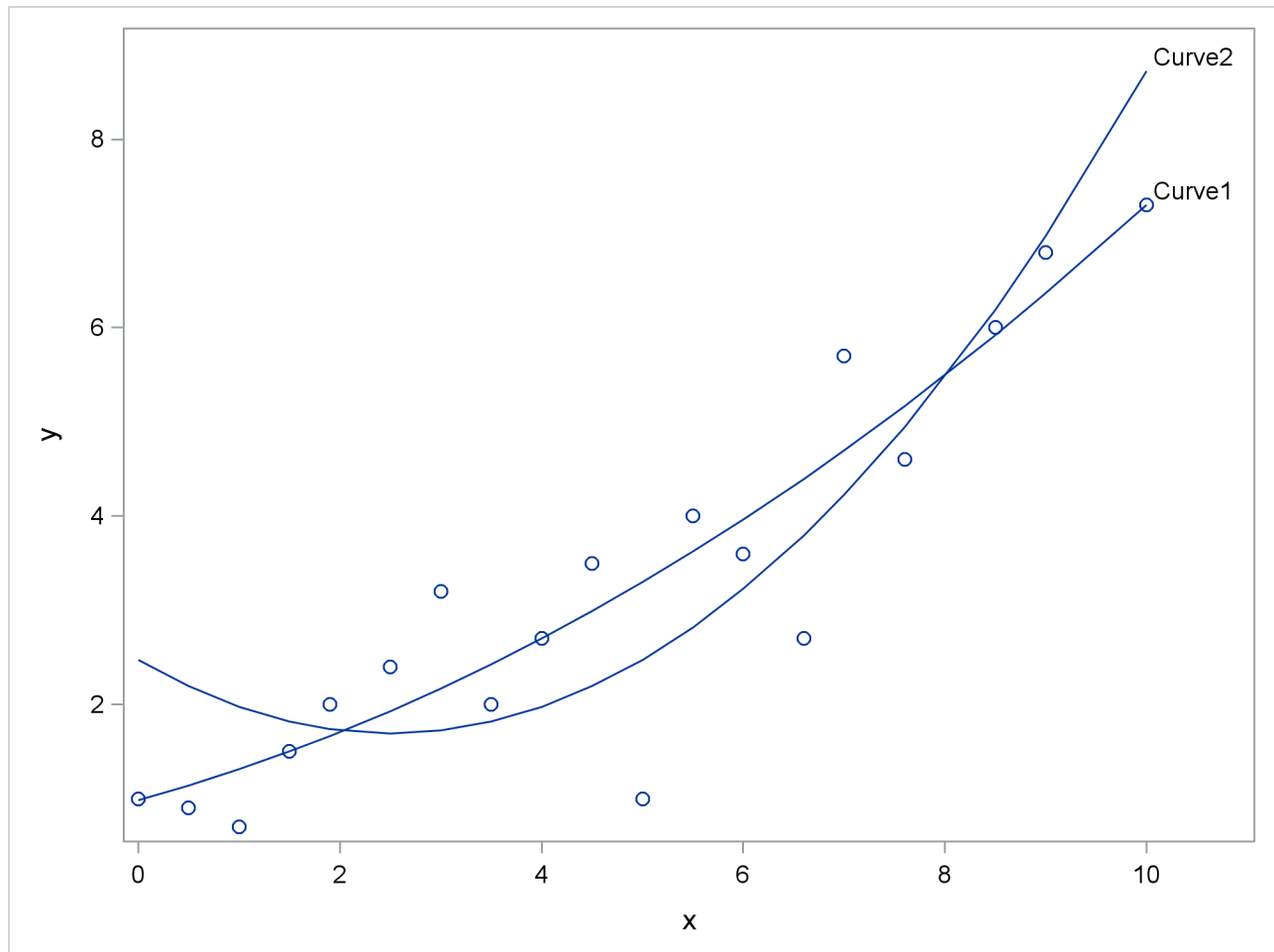
proc sgplot data=plot2;

```

```
scatter x=x y=y;  
series x=x y=Curve1 / curvelabel;  
series x=x y=Curve2 / curvelabel;  
run;
```

Figure 11.6 shows the regression curves for Problem (3), as on page 319 of Williams.

Figure 11.6 Regression Curves for Problem (3)



Features Demonstrated

The following features are demonstrated in this example:

- problem types: linear programming (polynomial regression with L_1 and L_∞ norms)
- numeric index set
- IMPVAR statement
- PROBLEM and USE PROBLEM statements
- multiple objectives
- using a variable suffix (such as `.sol`) in the declaration of a numeric parameter
- ABS function
- MAX aggregation operator
- SGPLOT procedure

Chapter 12

Logical Design: Constructing an Electronic System with a Minimum Number of Components

Contents

Problem Statement	147
Mathematical Programming Formulation	149
Input Data	151
PROC OPTMODEL Statements and Output	151
Features Demonstrated	155

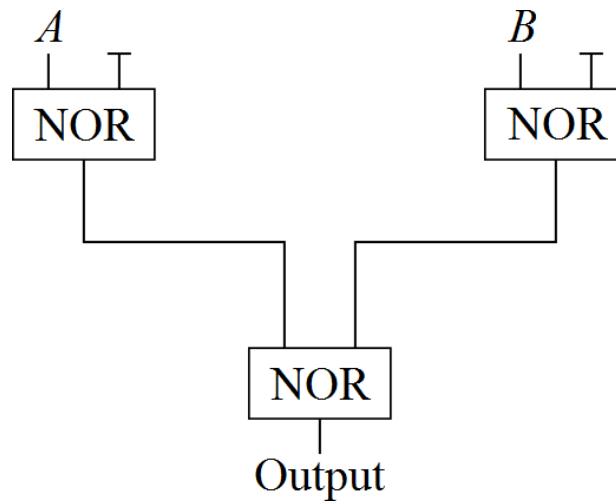
Problem Statement

Logical circuits have a given number of inputs and one output.¹ Impulses may be applied to the inputs of a given logical circuit and it will respond either by giving an output (signal 1) or by giving no output (signal 0). The input impulses are of the same kind as the outputs, i.e. 1 (positive input) or 0 (no input).

In this example a logical circuit is to be built up of NOR gates. A NOR gate is a device with two inputs and one output. It has the property that there is positive output (signal 1) if and only if *neither* input is positive, i.e. both inputs have value 0. By connecting such gates together with outputs from one gate possibly being inputs into another gate it is possible to construct a circuit to perform any desired logical function. For example the circuit illustrated in [Figure 12.1](#) will respond to the inputs *A* and *B* in the way indicated by the truth table in [Table 12.1](#).

The problem here is to construct a circuit using the *minimum number* of NOR gates which will perform the logical function specified by the truth table in [Table 12.2](#). This problem, together with further references to it, is discussed in Williams (1974).

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 243–244).

Figure 12.1**Table 12.1**

Inputs		Output
<i>A</i>	<i>B</i>	
0	0	0
0	1	0
1	0	0
1	1	1

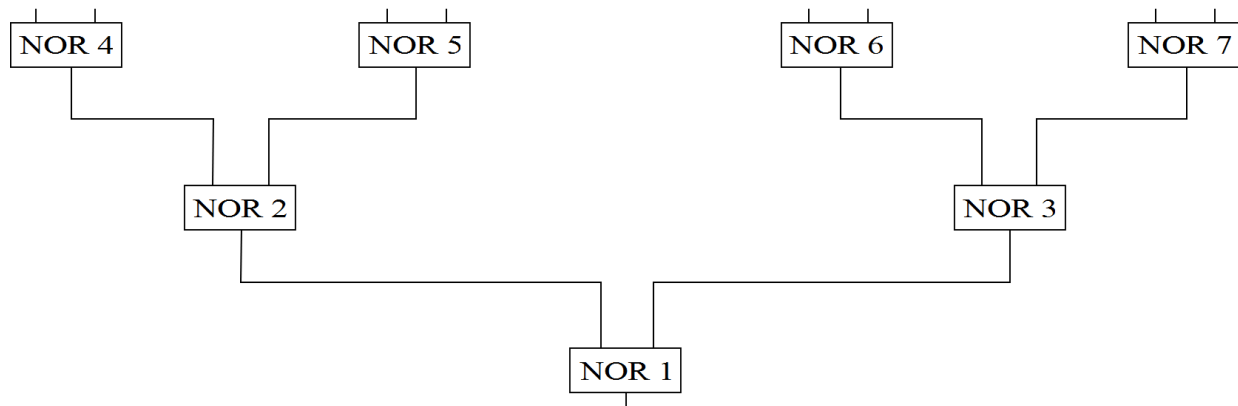
Table 12.2

Inputs		Output
<i>A</i>	<i>B</i>	
0	0	0
0	1	1
1	0	1
1	1	0

‘Fan-in’ and ‘fan-out’ are not permitted. That is, more than one output from a NOR gate cannot lead into one input, nor can one output lead into more than one input.

It may be assumed throughout that the optimal design is a ‘subnet’ of the ‘maximal’ net shown in [Figure 12.2](#).

Figure 12.2



Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $(i, j), (\text{pred}, \text{gate}) \in \text{ARCS}$
- $\text{pred}, \text{gate} \in \text{GATES}$
- $\text{row} \in \text{ROWS}$

Parameters

Table 12.3 shows the parameters used in this example.

Table 12.3 Parameters

Parameter Name	Interpretation
<i>inputA[row]</i>	Value of input <i>A</i> in each row of truth table
<i>inputB[row]</i>	Value of input <i>B</i> in each row of truth table
<i>target_output[row]</i>	Value of target output in each row of truth table

Variables

Table 12.4 shows the variables used in this example.

Table 12.4 Variables

Variable Name	Interpretation
UseGate[gate]	1 if gate is used; 0 otherwise
AssignAGate[gate]	1 if input <i>A</i> is assigned to gate; 0 otherwise
AssignBGate[gate]	1 if input <i>B</i> is assigned to gate; 0 otherwise
Output[gate,row]	Output value for each gate and each row of truth table

Objective

The objective is to minimize the following function:

$$\text{NumGatesUsed} = \sum_{\text{gate} \in \text{GATES}} \text{UseGate}[\text{gate}]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $\text{gate} \in \text{GATES}$,

$$\text{AssignAGate}[\text{gate}] \leq \text{UseGate}[\text{gate}]$$

- for $\text{gate} \in \text{GATES}$,

$$\text{AssignBGate}[\text{gate}] \leq \text{UseGate}[\text{gate}]$$

- for $\text{gate} \in \text{GATES}$,

$$\sum_{(\text{pred}, \text{gate}) \in \text{ARCS}} \text{UseGate}[\text{pred}] + \text{AssignAGate}[\text{gate}] + \text{AssignBGate}[\text{gate}] \leq 2$$

- for $\text{row} \in \text{ROWS}$,

$$\text{Output}[1, \text{row}] = \text{target_output}[\text{row}]$$

- for $\text{gate} \in \text{GATES}$ and $\text{row} \in \text{ROWS}$,

$$\text{Output}[\text{gate}, \text{row}] \leq \text{UseGate}[\text{gate}]$$

- for $\text{gate} \in \text{GATES}$ and $\text{row} \in \text{ROWS}$,

$$\text{inputA}[\text{row}] \cdot \text{AssignAGate}[\text{gate}] \leq 1 - \text{Output}[\text{gate}, \text{row}]$$

- for $\text{gate} \in \text{GATES}$ and $\text{row} \in \text{ROWS}$,

$$\text{inputB}[\text{row}] \cdot \text{AssignBGate}[\text{gate}] \leq 1 - \text{Output}[\text{gate}, \text{row}]$$

- for $(\text{pred}, \text{gate}) \in \text{ARCS}$ and $\text{row} \in \text{ROWS}$,

$$\text{Output}[\text{pred}, \text{row}] \leq 1 - \text{Output}[\text{gate}, \text{row}]$$

- for $\text{gate} \in \text{GATES}$ and $\text{row} \in \text{ROWS}$,

$$\begin{aligned} & \text{inputA}[\text{row}] \cdot \text{AssignAGate}[\text{gate}] + \text{inputB}[\text{row}] \cdot \text{AssignBGate}[\text{gate}] \\ & + \sum_{(\text{pred}, \text{gate}) \in \text{ARCS}} \text{Output}[\text{pred}, \text{row}] \\ & \geq \text{UseGate}[\text{gate}] - \text{Output}[\text{gate}, \text{row}] \end{aligned}$$

Input Data

The following data sets contain the input data that are used in this example:

```
data arc_data;
  input i j;
  datalines;
4 2
5 2
6 3
7 3
2 1
3 1
;

/* truth table (for XOR) */
data truth_data;
  input A B output;
  datalines;
0 0 0
0 1 1
1 0 1
1 1 0
;
```

PROC OPTMODEL Statements and Output

The following SET and READ statements declare the ARCS index set and then populate it by reading the arc_data data set:

```

proc optmodel;
  set <num,num> ARCS;
  read data arc_data into ARCS={i j};

```

Each arc corresponds to a connection from one NOR gate to another in Figure 12.2, and the following SET statement declares and populates the index set GATES by taking a union over ARCS, as in previous examples:

```

  set GATES = union {<i,j> in ARCS} {i,j};

  var UseGate {GATES} binary;
  min NumGatesUsed = sum {gate in GATES} UseGate[gate];

  var AssignAGate {GATES} binary;
  var AssignBGate {GATES} binary;

```

The following CON statements declare the constraints that if input *A* or *B* is assigned to a gate, that gate must be used:

```

  con AssignAGate_def {gate in GATES}:
    AssignAGate[gate] <= UseGate[gate];
  con AssignBGate_def {gate in GATES}:
    AssignBGate[gate] <= UseGate[gate];

```

The following CON statement declares the constraint that each gate has at most two inputs:

```

  con At_most_two_inputs {gate in GATES}:
    sum {<pred,(gate)> in ARCS} UseGate[pred]
    + AssignAGate[gate] + AssignBGate[gate]
    <= 2;

```

The following statements declare the ROWS index set and several parameters and read the truth_data data set that contains the target output:

```

  set ROWS;
  num inputA {ROWS};
  num inputB {ROWS};
  num target_output {ROWS};
  read data truth_data into ROWS={_N_} inputA=A inputB=B target_output=output;

```

The following VAR and FIX statements declare the Output variable and fix the output of Gate 1 to the desired values specified in *target_output*:

```

  var Output {GATES, ROWS} binary;
  for {row in ROWS} fix Output[1,row] = target_output[row];

```

The following CON statement declares the constraint that if any row has a positive Output, that gate must be used.

```

  con Output_link {gate in GATES, row in ROWS}:
    Output[gate,row] <= UseGate[gate];

```

The following CON statements declare the constraints that define each NOR gate:

```

/* if inputA[row] = 1 and AssignAGate[gate] = 1, then Output[gate,row] = 0 */
con NOR_def1 {gate in GATES, row in ROWS}:
    inputA[row] * AssignAGate[gate] <= 1 - Output[gate,row];

/* if inputB[row] = 1 and AssignBGate[gate] = 1, then Output[gate,row] = 0 */
con NOR_def2 {gate in GATES, row in ROWS}:
    inputB[row] * AssignBGate[gate] <= 1 - Output[gate,row];

/* if Output[pred,row] = 1, then Output[gate,row] = 0 */
con NOR_def3 {<pred, gate> in ARCS, row in ROWS}:
    Output[pred,row] <= 1 - Output[gate,row];

/* if UseGate[gate] = 1 and Output[gate,row] = 0, then
   (inputA[row] = 1 and AssignAGate[gate] = 1)
   or (inputB[row] = 1 and AssignBGate[gate] = 1)
   or sum {<pred, (gate)> in ARCS} Output[pred,row] >= 1 */
con NOR_def4 {gate in GATES, row in ROWS}:
    inputA[row] * AssignAGate[gate]
    + inputB[row] * AssignBGate[gate]
    + sum {<pred, (gate)> in ARCS} Output[pred,row]
    >= UseGate[gate] - Output[gate,row];

solve;
print UseGate AssignAGate AssignBGate;
print Output;
create data sol_data1 from [gate] UseGate AssignAGate AssignBGate;
create data sol_data2 from [gate row] Output;
quit;

```

Figure 12.3 shows the output from the mixed integer linear programming solver. In this case, five gates are used in the optimal solution.

Figure 12.3 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure			
Problem Summary			
Objective Sense	Minimization		
Objective Function	NumGatesUsed		
Objective Type	Linear		
Number of Variables	49		
Bounded Above	0		
Bounded Below	0		
Bounded Below and Above	45		
Free	0		
Fixed	4		
Binary	45		
Integer	4		
Number of Constraints	157		
Linear LE (\leq)	129		
Linear EQ ($=$)	0		
Linear GE (\geq)	28		
Linear Range	0		
Constraint Coefficients	344		
Solution Summary			
Solver	MILP		
Objective Function	NumGatesUsed		
Solution Status	Optimal		
Objective Value	5		
Iterations	103		
Best Bound	.		
Nodes	1		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	0		
Bound Infeasibility	0		
Integer Infeasibility	0		
[1]	Use Gate	Assign AGate	Assign BGate
1	1	0	0
2	1	1	1
3	1	0	0
4	0	0	0
5	0	0	0
6	1	0	1
7	1	1	0

Figure 12.3 *continued*

	Output			
	1	2	3	4
1	0	1	1	0
2	1	0	0	0
3	0	0	0	1
4	0	0	0	0
5	0	0	0	0
6	1	0	1	0
7	1	1	0	0

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- numeric and string index sets
- set of tuples
- set operator UNION
- implicit slice
- FIX statement
- modeling if-then constraints using binary variables

Chapter 13

Market Sharing: Assigning Retailers to Company Divisions

Contents

Problem Statement	157
Mathematical Programming Formulation	158
Input Data	161
PROC OPTMODEL Statements and Output	162
Features Demonstrated	168

Problem Statement

A large company has two divisions D1 and D2.¹ The company supplies retailers with oil and spirit.

It is desired to allocate each retailer to either division D1 or division D2. This division will be the retailer's supplier. As far as possible this division must be made so that D1 controls 40% of the market and D2 the remaining 60%. The retailers are listed below as M1 to M23. Each retailer has an estimated market for oil and spirit. Retailers M1 to M8 are in region 1; retailers M9 to M18 are in region 2; retailers M19 to M23 are in region 3. Certain retailers are considered to have good growth prospects and categorized as group A and the others are in group B. Each retailer has a certain number of delivery points as given below. It is desired to make the 40/60 split between D1 and D2 in each of the following respects:

- (1) Total number of delivery points
- (2) Control of spirit market
- (3) Control of oil market in region 1
- (4) Control of oil market in region 2
- (5) Control of oil market in region 3
- (6) Number of retailers in group A
- (7) Number of retailers in group B

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 244–245).

Table 13.1

	Retailer	Oil market (10 ⁶ gallons)	Delivery points	Spirit market (10 ⁶ gallons)	Growth category
Region 1	M1	9	11	34	A
	M2	13	47	411	A
	M3	14	44	82	A
	M4	17	25	157	B
	M5	18	10	5	A
	M6	19	26	183	A
	M7	23	26	14	B
	M8	21	54	215	B
Region 2	M9	9	18	102	B
	M10	11	51	21	A
	M11	17	20	54	B
	M12	18	105	0	B
	M13	18	7	6	B
	M14	17	16	96	B
	M15	22	34	118	A
	M16	24	100	112	B
	M17	36	50	535	B
	M18	43	21	8	B
Region 3	M19	6	11	53	B
	M20	15	19	28	A
	M21	15	14	69	B
	M22	25	10	65	B
	M23	39	11	27	B

There is a certain flexibility in that any share may vary by $\pm 5\%$. That is, the share can vary between the limits 35/65 and 45/55.

The primary aim is to find a feasible solution. If, however, there is some choice, then possible objectives are (i) to minimize the sum of the percentage deviations from the 40/60 split and (ii) to minimize the maximum such deviation.

Build a model to see if the problem has a feasible solution and if so find the optimal solutions.

The numerical data are given in Table 13.1.

Mathematical Programming Formulation

Williams uses the fact that the company has only two divisions to reduce the number of decision variables and constraints. The formulation shown here generalizes to any number of divisions.

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{retailer} \in \text{RETAILERS}$
- $r, \text{reg} \in \text{REGIONS}$
- $g, \text{group} \in \text{GROUPS}$
- $\text{retailer} \in \text{RETAILERS_region}[\text{region}]$
- $\text{retailer} \in \text{RETAILERS_group}[\text{group}]$
- $\text{division} \in \text{DIVISIONS}$
- $\text{category} \in \text{CATEGORIES}$

Parameters

Table 13.2 shows the parameters used in this example.

Table 13.2 Parameters

Parameter Name	Interpretation
<i>region[retailer]</i>	Region per retailer
<i>oil[retailer]</i>	Estimated market for oil per retailer
<i>delivery[retailer]</i>	Number of delivery points per retailer
<i>spirit[retailer]</i>	Estimated market for spirit per retailer
<i>growth[retailer]</i>	Growth category per retailer
<i>target[division]</i>	Market share target per division
<i>tolerance</i>	Amount by which market share can differ from target
<i>sum_abs_dev</i>	Sum of absolute deviations between market share and target
<i>max_abs_dev</i>	Maximum of absolute deviations between market share and target

Variables

Table 13.3 shows the variables used in this example.

Table 13.3 Variables

Variable Name	Interpretation
Assign[retailer,division]	1 if retailer is assigned to division; 0 otherwise
MarketShare[category,division]	Market share for category and division
Surplus[category,division]	$\max\{\text{MarketShare}[\text{category},\text{division}] - \text{target}[\text{division}], 0\}$
Slack[category,division]	$\max\{\text{target}[\text{division}] - \text{MarketShare}[\text{category},\text{division}], 0\}$
MinMax	$\max_{\substack{\text{category} \in \text{CATEGORIES}, \\ \text{division} \in \text{DIVISIONS}}} \text{MarketShare}[\text{category},\text{division}] - \text{target}[\text{division}] $

Objectives

The first objective is to minimize the following (nonlinear, nondifferentiable) L_1 norm function:

$$\text{Objective1} = \sum_{\substack{\text{category} \in \text{CATEGORIES}, \\ \text{division} \in \text{DIVISIONS}}} |\text{MarketShare}[\text{category},\text{division}] - \text{target}[\text{division}]|$$

The second objective is to minimize the following (nonlinear, nondifferentiable) L_∞ norm function:

$$\text{Objective2} = \max_{\substack{\text{category} \in \text{CATEGORIES}, \\ \text{division} \in \text{DIVISIONS}}} |\text{MarketShare}[\text{category},\text{division}] - \text{target}[\text{division}]|$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for retailer \in RETAILERS,

$$\sum_{\text{division} \in \text{DIVISIONS}} \text{Assign}[\text{retailer},\text{division}] = 1$$

- for division \in DIVISIONS,

$$\text{MarketShare}[\text{'delivery'},\text{division}] = \frac{\sum_{\text{retailer} \in \text{RETAILERS}} \text{delivery}[\text{retailer}] \cdot \text{Assign}[\text{retailer},\text{division}]}{\sum_{\text{retailer} \in \text{RETAILERS}} \text{delivery}[\text{retailer}]}$$

- for division \in DIVISIONS,

$$\text{MarketShare}[\text{'spirit'},\text{division}] = \frac{\sum_{\text{retailer} \in \text{RETAILERS}} \text{spirit}[\text{retailer}] \cdot \text{Assign}[\text{retailer},\text{division}]}{\sum_{\text{retailer} \in \text{RETAILERS}} \text{spirit}[\text{retailer}]}$$

- for $\text{reg} \in \text{REGIONS}$ and $\text{division} \in \text{DIVISIONS}$,

$$\text{MarketShare}[\text{'oil'}||\text{reg},\text{division}] = \frac{\sum_{\text{retailer} \in \text{RETAILERS_region}[\text{reg}]} \text{oil}[\text{retailer}] \cdot \text{Assign}[\text{retailer},\text{division}]}{\sum_{\text{retailer} \in \text{RETAILERS_region}[\text{reg}]} \text{oil}[\text{retailer}]}$$

- for $\text{group} \in \text{GROUPS}$ and $\text{division} \in \text{DIVISIONS}$,

$$\text{MarketShare}[\text{'growth'}||\text{group},\text{division}] = \frac{\sum_{\text{retailer} \in \text{RETAILERS_group}[\text{group}]} \text{Assign}[\text{retailer},\text{division}]}{|\text{RETAILERS_group}[\text{group}]|}$$

- for $\text{category} \in \text{CATEGORIES}$ and $\text{division} \in \text{DIVISIONS}$,

$$\text{MarketShare}[\text{category},\text{division}] - \text{Surplus}[\text{category},\text{division}] + \text{Slack}[\text{category},\text{division}] = \text{target}[\text{division}]$$

- for $\text{category} \in \text{CATEGORIES}$ and $\text{division} \in \text{DIVISIONS}$,

$$\text{MinMax} \geq \text{Surplus}[\text{category},\text{division}] + \text{Slack}[\text{category},\text{division}]$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data retailer_data;
  input region oil delivery spirit growth $;
  datalines;
1  9  11  34 A
1 13  47 411 A
1 14  44  82 A
1 17  25 157 B
1 18  10   5 A
1 19  26 183 A
1 23  26  14 B
1 21  54 215 B
2  9  18 102 B
2 11  51  21 A
2 17  20  54 B
2 18 105   0 B
2 18   7   6 B
2 17  16  96 B
2 22  34 118 A
2 24 100 112 B
2 36  50 535 B
2 43  21   8 B
3  6  11  53 B
3 15  19  28 A
3 15  14  69 B
3 25  10  65 B
```

```

3 39 11 27 B
;

data division_data;
    input target;
    datalines;
0.40
0.60
;

%let tolerance = 0.05;

```

PROC OPTMODEL Statements and Output

The following PROC OPTMODEL statements declare an index set and parameters and then read the input data:

```

proc optmodel;
    set RETAILERS;
    num region {RETAILERS};
    num oil {RETAILERS};
    num delivery {RETAILERS};
    num spirit {RETAILERS};
    str growth {RETAILERS};
    read data retailer_data into RETAILERS=[_N_] region oil delivery spirit growth;

```

The following statements declare parameters and index sets, which are initialized to be empty and then populated within a FOR loop. Note that both RETAILERS_region and RETAILERS_group are sets that are indexed by other sets:

```

    set REGIONS init {};
    set RETAILERS_region {REGIONS} init {};
    num r;
    set <str> GROUPS init {};
    set RETAILERS_group {GROUPS} init {};
    str g;
    for {retailer in RETAILERS} do;
        r = region[retailer];
        REGIONS = REGIONS union {r};
        RETAILERS_region[r] = RETAILERS_region[r] union {retailer};
        g = growth[retailer];
        GROUPS = GROUPS union {g};
        RETAILERS_group[g] = RETAILERS_group[g] union {retailer};
    end;

    set DIVISIONS;
    num target {DIVISIONS};
    read data division_data into DIVISIONS=[_N_] target;

    num tolerance = &tolerance;

```

The following declarations are straightforward:

```

var Assign {RETAILERS, DIVISIONS} binary;

con Assign_con {retailer in RETAILERS}:
    sum {division in DIVISIONS} Assign[retailer,division] = 1;

set CATEGORIES = {'delivery','spirit'}
    union (setof {reg in REGIONS} 'oil' || reg)
    union (setof {group in GROUPS} 'growth' || group);
var MarketShare {CATEGORIES, DIVISIONS};
var Surplus      {CATEGORIES, DIVISIONS} >= 0 <= tolerance;
var Slack        {CATEGORIES, DIVISIONS} >= 0 <= tolerance;

min Objective1 =
    sum {category in CATEGORIES, division in DIVISIONS}
        (Surplus[category,division] + Slack[category,division]);

con Delivery_con {division in DIVISIONS}:
    MarketShare['delivery',division]
    = (sum {retailer in RETAILERS} delivery[retailer] * Assign[retailer,division])
      / (sum {retailer in RETAILERS} delivery[retailer]);

con Spirit_con {division in DIVISIONS}:
    MarketShare['spirit',division]
    = (sum {retailer in RETAILERS} spirit[retailer] * Assign[retailer,division])
      / (sum {retailer in RETAILERS} spirit[retailer]);

con Oil_con {reg in REGIONS, division in DIVISIONS}:
    MarketShare['oil' || reg,division]
    = (sum {retailer in RETAILERS_region[reg]}
        oil[retailer] * Assign[retailer,division])
      / (sum {retailer in RETAILERS_region[reg]} oil[retailer]);

con Growth_con {group in GROUPS, division in DIVISIONS}:
    MarketShare['growth' || group,division]
    = (sum {retailer in RETAILERS_group[group]} Assign[retailer,division])
      / card(RETAILERS_group[group]);

con Abs_dev_con {category in CATEGORIES, division in DIVISIONS}:
    MarketShare[category,division]
    - Surplus[category,division] + Slack[category,division]
    = target[division];

```

The following NUM statements use the ABS function, the .sol variable suffix, and the MAX aggregation operator to compute the two norms from the optimal solution:

```

num sum_abs_dev =
    sum {category in CATEGORIES, division in DIVISIONS}
        abs(MarketShare[category,division].sol - target[division]);
num max_abs_dev =
    max {category in CATEGORIES, division in DIVISIONS}
        abs(MarketShare[category,division].sol - target[division]);

```

The MILP solver is called twice, and each SOLVE statement includes the OBJ option to specify which objective to optimize. The first PRINT statement after each SOLVE statement reports the values of both objectives even though only one objective is optimized at a time:

```
solve obj Objective1;
print sum_abs_dev max_abs_dev;
print Assign;
print MarketShare Surplus Slack;
```

Figure 13.1 shows the output that results from the first SOLVE statement. The optimal objective value disagrees with the value 0.0453 presented in Williams, even after you account for the fact that the formulation here counts each deviation twice (once per division).

Figure 13.1 Output from First SOLVE Statement, Minimizing Sum of Deviations

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	Objective1
Objective Type	Linear
Number of Variables	88
Bounded Above	0
Bounded Below	0
Bounded Below and Above	74
Free	14
Fixed	0
Binary	46
Integer	0
Number of Constraints	51
Linear LE (<=)	0
Linear EQ (=)	51
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	284
Solution Summary	
Solver	MILP
Objective Function	Objective1
Solution Status	Optimal
Objective Value	0.1059523594
Iterations	4302
Best Bound	.
Nodes	511
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	2.220446E-16
Bound Infeasibility	4.440892E-16
Integer Infeasibility	4.440892E-16

Figure 13.1 continued

		sum_ abs_dev	max_ abs_ dev	
		0.10595	0.025	
Assign				
		1	2	
	1	1	0	
	2	1	0	
	3	1	0	
	4	1	0	
	5	0	1	
	6	0	1	
	7	0	1	
	8	0	1	
	9	0	1	
	10	0	1	
	11	1	0	
	12	0	1	
	13	-0	1	
	14	0	1	
	15	0	1	
	16	1	0	
	17	0	1	
	18	1	0	
	19	0	1	
	20	0	1	
	21	1	-0	
	22	1	0	
	23	0	1	
Market				
[1]	[2]	Share	Surplus	Slack
delivery	1	0.40000	0.0000000	-0.0000000
delivery	2	0.60000	0.0000000	0.0000000
growthA	1	0.37500	0.0000000	0.0250000
growthA	2	0.62500	0.0250000	0.0000000
growthB	1	0.40000	-0.0000000	0.0000000
growthB	2	0.60000	0.0000000	-0.0000000
oil1	1	0.39552	0.0000000	0.0044776
oil1	2	0.60448	0.0044776	0.0000000
oil2	1	0.39070	0.0000000	0.0093023
oil2	2	0.60930	0.0093023	0.0000000
oil3	1	0.40000	0.0000000	0.0000000
oil3	2	0.60000	-0.0000000	0.0000000
spirit	1	0.41420	0.0141962	0.0000000
spirit	2	0.58580	0.0000000	0.0141962

The following statements declare the additional variable, objective, and constraints for problem (ii), with the tighter linearization of the L_∞ norm as in [Chapter 11](#):

```

var MinMax >= 0 init max_abs_dev;
min Objective2 = MinMax;
con MinMax_con {category in CATEGORIES, division in DIVISIONS}:
    MinMax >= Surplus[category,division] + Slack[category,division];

```

The following statements call the MILP solver to solve problem (ii). The PRIMALIN option is used to initialize the MILP solver with the optimal solution that was obtained for problem (i):

```

solve obj Objective2 with MILP / primalin;
print sum_abs_dev max_abs_dev;
print Assign;
print MarketShare Surplus Slack;
quit;

```

Figure 13.2 shows the output that results from the second SOLVE statement.

Figure 13.2 Output from Second SOLVE Statement, Minimizing Maximum Deviation

Problem Summary	
Objective Sense	Minimization
Objective Function	Objective2
Objective Type	Linear
Number of Variables	89
Bounded Above	0
Bounded Below	1
Bounded Below and Above	74
Free	14
Fixed	0
Binary	46
Integer	0
Number of Constraints	65
Linear LE (<=)	0
Linear EQ (=)	51
Linear GE (>=)	14
Linear Range	0
Constraint Coefficients	326

Figure 13.2 continued

Solution Summary		
Solver	MILP	
Objective Function	Objective2	
Solution Status	Optimal	
Objective Value	0.025	
Iterations	46	
Best Bound	.	
Nodes	1	
Relative Gap	0	
Absolute Gap	0	
Primal Infeasibility	2.220446E-16	
Bound Infeasibility	7.043926E-17	
Integer Infeasibility	0	
	sum_	max_
	abs_dev	abs_dev
	0.10595	0.025
Assign		
	1	2
1	1	0
2	1	0
3	1	0
4	1	0
5	0	1
6	0	1
7	0	1
8	0	1
9	0	1
10	0	1
11	1	0
12	0	1
13	0	1
14	0	1
15	0	1
16	1	0
17	0	1
18	1	0
19	0	1
20	0	1
21	1	0
22	1	0
23	0	1

Figure 13.2 *continued*

[1]	[2]	Market Share	Surplus	Slack
delivery	1	0.40000	0.0000000	-0.0000000
delivery	2	0.60000	0.0000000	0.0000000
growthA	1	0.37500	0.0000000	0.0250000
growthA	2	0.62500	0.0250000	0.0000000
growthB	1	0.40000	-0.0000000	0.0000000
growthB	2	0.60000	0.0000000	-0.0000000
oil1	1	0.39552	0.0000000	0.0044776
oil1	2	0.60448	0.0044776	0.0000000
oil2	1	0.39070	0.0000000	0.0093023
oil2	2	0.60930	0.0093023	0.0000000
oil3	1	0.40000	0.0000000	0.0000000
oil3	2	0.60000	-0.0000000	0.0000000
spirit	1	0.41420	0.0141962	0.0000000
spirit	2	0.58580	0.0000000	0.0141962

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (L_1 and L_∞ norms)
- numeric and string index sets
- multiple input data sets
- set operators UNION and SETOF
- string concatenation
- sets indexed by other sets
- multiple objectives and the OBJ option
- using a variable suffix (such as `.sol`) in the declaration of a numeric parameter
- CARD function
- ABS function
- MAX aggregation operator
- MILP solver option PRIMALIN

Chapter 14

Opencast Mining: How Much to Excavate

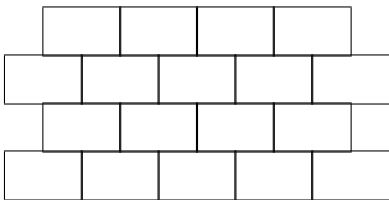
Contents

Problem Statement	169
Mathematical Programming Formulation	170
Input Data	171
PROC OPTMODEL Statements and Output	172
Features Demonstrated	175

Problem Statement

A company has obtained permission to opencast mine within a square plot 200 ft \times 200 ft.¹ The angle of slip of the soil is such that it is not possible for the sides of the excavation to be steeper than 45°. The company has obtained estimates for the value of the ore in various places at various depths. Bearing in mind the restrictions imposed by the angle of slip, the company decides to consider the problem as one of the extracting of rectangular blocks. Each block has horizontal dimensions 50 ft \times 50 ft and a vertical dimension of 25 ft. If the blocks are chosen to lie above one another, as illustrated in vertical section in Figure 14.1, then it is only possible to excavate blocks forming an upturned pyramid. (In a three-dimensional representation Figure 14.1 would show four blocks lying above each lower block.)

Figure 14.1



If the estimates of ore value are applied to give values (in percentage of pure metal) for each block in the maximum pyramid which can be extracted then the following values are obtained:

Level 1 (surface)	1.5	1.5	1.5	0.75	Level 2 (25 ft depth)	4.0	4.0	2.0
	1.5	2.0	1.5	0.75		3.0	3.0	1.0
	1.0	1.0	0.75	0.5		2.0	2.0	0.5
	0.75	0.75	0.5	0.25				
Level 3 (50 ft depth)	12.0		6.0		Level 4 (75 ft depth)	6.0		
	5.0		4.0					

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 245–247).

The cost of extraction increases with depth. At successive levels the cost of extracting a block is:

Level 1	£3000
Level 2	£6000
Level 3	£8000
Level 4	£10,000

The revenue obtained from a ‘100% value block’ would be £200,000. For each block here the revenue is proportional to ore value.

Build a model to help decide the best blocks to extract. The objective is to maximize revenue – cost.

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{block}, i, j \in \text{BLOCKS}$
- $\text{level} \in \text{LEVELS}$

Parameters

Table 14.1 shows the parameters used in this example.

Table 14.1 Parameters

Parameter Name	Interpretation
<i>level[block]</i>	Level of block
<i>row[block]</i>	Row of block
<i>column[block]</i>	Column of block
<i>revenue[block]</i>	Revenue obtained from extracting block
<i>cost[level]</i>	Cost of extracting block per level
<i>full_value</i>	Revenue obtained from a 100% value block
<i>profit[block]</i>	Profit obtained from extracting block

Variables

Table 14.2 shows the variables used in this example.

Table 14.2 Variables

Variable Name	Interpretation
Extract[block]	1 if block is extracted; 0 otherwise

Objective

The objective is to maximize the following function:

$$\text{TotalProfit} = \sum_{\text{block} \in \text{BLOCKS}} \text{profit}[\text{block}] \cdot \text{Extract}[\text{block}]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $i \in \text{BLOCKS}$ and $j \in \text{BLOCKS}$ such that $\text{level}[j] = \text{level}[i] - 1$ and $\text{row}[j] \in \{\text{row}[i], \text{row}[i] + 1\}$ and $\text{column}[j] \in \{\text{column}[i], \text{column}[i] + 1\}$,

$$\text{Extract}[i] \leq \text{Extract}[j]$$

Input Data

The following data sets and macro variable contain the input data that are used in this example:

```
data block_data;
  input level row column percent;
  datalines;
1 1 1 1.5
1 1 2 1.5
1 1 3 1.5
1 1 4 0.75
1 2 1 1.5
1 2 2 2.0
1 2 3 1.5
1 2 4 0.75
1 3 1 1.0
1 3 2 1.0
1 3 3 0.75
1 3 4 0.5
1 4 1 0.75
1 4 2 0.75
1 4 3 0.5
```

```

1 4 4 0.25
2 1 1 4.0
2 1 2 4.0
2 1 3 2.0
2 2 1 3.0
2 2 2 3.0
2 2 3 1.0
2 3 1 2.0
2 3 2 2.0
2 3 3 0.5
3 1 1 12.0
3 1 2 6.0
3 2 1 5.0
3 2 2 4.0
4 1 1 6.0
;

data level_data;
    input cost;
    datalines;
3000
6000
8000
10000
;

%let full_value = 200000;

```

PROC OPTMODEL Statements and Output

The following PROC OPTMODEL statements declare index sets and parameters and then read data sets to populate them:

```

proc optmodel;
    set BLOCKS;
    num level {BLOCKS};
    num row {BLOCKS};
    num column {BLOCKS};
    num revenue {BLOCKS};
    read data block_data into BLOCKS=[_N_] level row column revenue=percent;
    for {block in BLOCKS} revenue[block] = &full_value * revenue[block] / 100;

    set LEVELS;
    num cost {LEVELS};
    read data level_data into LEVELS=[_N_] cost;

```

The following statements declare binary decision variables and the objective:

```

var Extract {BLOCKS} binary;
num profit {block in BLOCKS} = revenue[block] - cost[level[block]];
max TotalProfit = sum {block in BLOCKS} profit[block] * Extract[block];

```

The following CON statement uses the colon operator (:) to enforce the rule that if you extract block i , then you must also extract each block j that lies above it:

```

con Precedence_con {i in BLOCKS, j in BLOCKS:
    level[j] = level[i] - 1
    and row[j] in {row[i], row[i]+1}
    and column[j] in {column[i], column[i]+1}
}:
    Extract[i] <= Extract[j];

```

The following SOLVE statement calls the MILP solver. However, since the constraint matrix is totally unimodular, the optimal solution of the LP relaxation is automatically integral, and hence no branching is required.

```

solve;
print Extract profit;

```

The following CREATE DATA statement uses the colon operator (:) to output only the blocks that are used in the optimal solution:

```

create data sol_data from [block]={block in BLOCKS: Extract[block].sol > 0.5}
    level row column revenue cost[level[block]] profit;
quit;

```

Figure 14.2 shows the output from the mixed integer linear programming solver.

Figure 14.2 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure		
Problem Summary		
Objective Sense	Maximization	
Objective Function	TotalProfit	
Objective Type	Linear	
Number of Variables	30	
Bounded Above	0	
Bounded Below	0	
Bounded Below and Above	30	
Free	0	
Fixed	0	
Binary	30	
Integer	0	
Number of Constraints	56	
Linear LE (\leq)	56	
Linear EQ ($=$)	0	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	112	
Solution Summary		
Solver	MILP	
Objective Function	TotalProfit	
Solution Status	Optimal	
Objective Value	17500	
Iterations	7	
Best Bound	.	
Nodes	1	
Relative Gap	0	
Absolute Gap	0	
Primal Infeasibility	0	
Bound Infeasibility	0	
Integer Infeasibility	0	

Figure 14.2 *continued*

[1]	Extract	profit
1	1	0
2	1	0
3	1	0
4	0	-1500
5	1	0
6	1	1000
7	1	0
8	0	-1500
9	1	-1000
10	1	-1000
11	1	-1500
12	0	-2000
13	0	-1500
14	0	-1500
15	0	-2000
16	0	-2500
17	1	2000
18	1	2000
19	0	-2000
20	1	0
21	1	0
22	0	-4000
23	0	-2000
24	0	-2000
25	0	-5000
26	1	16000
27	0	4000
28	0	2000
29	0	0
30	0	2000

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (totally unimodular)
- reading multiple input data sets
- IN expression
- logical operator AND
- using a colon (:) to select members of a set
- modeling if-then constraints using binary variables
- creating an output data set from a subset of an index set

Chapter 15

Tariff Rates (Power Generation): How to Determine Tariff Rates for the Sale of Electricity

Contents

Problem Statement	177
Mathematical Programming Formulation	178
Input Data	180
PROC OPTMODEL Statements and Output	180
Features Demonstrated	187

Problem Statement

A number of power stations are committed to meeting the following electricity load demands over a day:¹

12	[a.m.]	to	6	a.m.	15,000 megawatts
6	a.m.	to	9	a.m.	30,000 megawatts
9	a.m.	to	3	p.m.	25,000 megawatts
3	p.m.	to	6	p.m.	40,000 megawatts
6	p.m.	to	12	[a.m.]	27,000 megawatts

There are three types of generating unit available: 12 of type 1, 10 of type 2, and five of type 3. Each generator has to work between a minimum and a maximum level. There is an hourly cost of running each generator at minimum level. In addition there is an extra hourly cost for each megawatt at which a unit is operated above minimum level. To start up a generator also involves a cost. All this information is given in Table 15.1 (with costs in £).

In addition to meeting the estimated load demands there must be sufficient generators working at any time to make it possible to meet an increase in load of up to 15%. This increase would have to be accomplished by adjusting the output of generators already operating within their permitted limits.

Which generators should be working in which periods of the day to minimize total cost?

What is the marginal cost of production of electricity in each period of the day; i.e. what tariffs should be charged?

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, p. 247).

What would be the saving of lowering the 15% reserve output guarantee; i.e. what does this security of supply guarantee cost?

Table 15.1

	Minimum level	Maximum level	Cost per hour at minimum	Cost per hour per megawatt above minimum	[Start-up] cost
Type 1	850 MW	2000 MW	1000	2	2000
Type 2	1250 MW	1750 MW	2600	1.30	1000
Type 3	1500 MW	4000 MW	3000	3	500

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{period} \in \text{PERIODS}$
- $\text{type} \in \text{TYPES}$

Parameters

Table 15.2 shows the parameters used in this example.

Table 15.2 Parameters

Parameter Name	Interpretation
<i>length[period]</i>	Length per period (in hours)
<i>demand[period]</i>	Demand per period (in megawatts)
<i>num_avail[type]</i>	Number of units available per generator
<i>min_level[type]</i>	Minimum level (in megawatts) per generator
<i>max_level[type]</i>	Maximum level (in megawatts) per generator
<i>unit_cost[type]</i>	Hourly cost (in pounds) of running each generator at minimum level
<i>excess_cost[type]</i>	Extra hourly cost (in pounds) per megawatt at which a unit is operated above minimum level
<i>startup_cost[type]</i>	Cost (in pounds) of starting up each generator
<i>reserve</i>	Additional fraction of estimated load required to be met by generators already in operation

Variables

Table 15.3 shows the variables used in this example.

Table 15.3 Variables

Variable Name	Interpretation
NumWorking[period,type]	Number of units per type of generator operating per period
Excess[period,type]	Number of megawatt-hours generated above minimum level
NumStartup[period,type]	Number of units per type of generator starting up per period
Output[period,type]	Number of megawatt-hours generated

Objective

The objective is to minimize the following function:

$$\begin{aligned}
 \text{TotalCost} = & \sum_{\substack{\text{period} \in \text{PERIODS}, \\ \text{type} \in \text{TYPES}}} (unit_cost[type] \cdot length[period] \cdot \text{NumWorking}[\text{period}, \text{type}] \\
 & + excess_cost[type] \cdot length[period] \cdot \text{Excess}[\text{period}, \text{type}] \\
 & + startup_cost[type] \cdot \text{NumStartup}[\text{period}, \text{type}])
 \end{aligned}$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for period \in PERIODS and type \in TYPES,

$$\text{Output}[\text{period}, \text{type}] = min_level[type] \cdot \text{NumWorking}[\text{period}, \text{type}] + \text{Excess}[\text{period}, \text{type}]$$

- for period \in PERIODS,

$$\sum_{\text{type} \in \text{TYPES}} \text{Output}[\text{period}, \text{type}] \geq demand[\text{period}]$$

- for period \in PERIODS,

$$\sum_{\text{type} \in \text{TYPES}} max_level[type] \cdot \text{NumWorking}[\text{period}, \text{type}] \geq (1 + reserve) \cdot demand[\text{period}]$$

- for period \in PERIODS and type \in TYPES,

$$\text{Excess}[\text{period}, \text{type}] \leq (max_level[type] - min_level[type]) \cdot \text{NumWorking}[\text{period}, \text{type}]$$

- for period \in PERIODS and type \in TYPES,

$$\text{NumStartup}[\text{period}, \text{type}] \geq \text{NumWorking}[\text{period}, \text{type}]$$

– (if period $- 1 \in$ PERIODS, then NumWorking[period -1 ,type];
 else NumWorking[|PERIODS|,type])

Input Data

The following data sets and macro variable contain the input data that are used in this example:

```
data period_data;
  input length demand;
  datalines;
6 15000
3 30000
6 25000
3 40000
6 27000
;

data type_data;
  input num_avail min_level max_level unit_cost excess_cost startup_cost;
  datalines;
12 850 2000 1000 2 2000
10 1250 1750 2600 1.30 1000
5 1500 4000 3000 3 500
;

%let reserve = 0.15;
```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read the input data:

```
proc optmodel;
  set PERIODS;
  num length {PERIODS};
  num demand {PERIODS};
  read data period_data into PERIODS=[_N_] length demand;

  set TYPES;
  num num_avail {TYPES};
  num min_level {TYPES};
  num max_level {TYPES};
  num unit_cost {TYPES};
```

```

num excess_cost {TYPES};
num startup_cost {TYPES};
read data type_data into TYPES=[_N_]
    num_avail min_level max_level unit_cost excess_cost startup_cost;

```

The following VAR statements declare decision variables, with bounds that depend on type:

```

var NumWorking {PERIODS, type in TYPES} >= 0 <= num_avail[type] integer;
var Excess {PERIODS, TYPES} >= 0;
var NumStartup {PERIODS, type in TYPES} >= 0 <= num_avail[type] integer;

```

The following IMPVAR statement declares Output as an implicit variable:

```

impvar Output {period in PERIODS, type in TYPES} =
    min_level[type] * NumWorking[period,type] + Excess[period,type];

```

The following MIN statement declares the objective:

```

min TotalCost =
    sum {period in PERIODS, type in TYPES} (
        unit_cost[type] * length[period] * NumWorking[period,type]
        + excess_cost[type] * length[period] * Excess[period,type]
        + startup_cost[type] * NumStartup[period,type]);

```

The following CON statements declare the constraints, with an IF-THEN/ELSE expression in the last constraint to account for a boundary condition in the first period:

```

con Demand_con {period in PERIODS}:
    sum {type in TYPES} Output[period,type]
    >= demand[period];

con Reserve_con {period in PERIODS}:
    sum {type in TYPES} max_level[type] * NumWorking[period,type]
    >= (1 + &reserve) * demand[period];

con Excess_ub {period in PERIODS, type in TYPES}:
    Excess[period,type]
    <= (max_level[type] - min_level[type]) * NumWorking[period,type];

con Startup_con {period in PERIODS, type in TYPES}:
    NumStartup[period,type]
    >= NumWorking[period,type]
    - (if period - 1 in PERIODS then NumWorking[period-1,type]
        else NumWorking[card(PERIODS),type]);

```

The following statements call the mixed integer linear programming solver, print the optimal solution, and create a data set that contains the optimal solution, with one observation per period-type pair:

```

solve;
print NumWorking NumStartup Excess Output;
create data sol_data from [period type] NumWorking NumStartup Excess Output;

```

Figure 15.1 shows the output from the mixed integer linear programming solver.

Figure 15.1 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	45
Bounded Above	0
Bounded Below	15
Bounded Below and Above	30
Free	0
Fixed	0
Binary	0
Integer	30
Number of Constraints	40
Linear LE (<=)	15
Linear EQ (=)	0
Linear GE (>=)	25
Linear Range	0
Constraint Coefficients	120
Solution Summary	
Solver	MILP
Objective Function	TotalCost
Solution Status	Optimal
Objective Value	988540
Iterations	29
Best Bound	.
Nodes	1
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	3.552714E-15

Figure 15.1 *continued*

	[1]	[2]	Num Working	Num Startup	Excess	Output
1	1		12	0	0	10200
1	2		3	0	1050	4800
1	3		0	0	0	0
2	1		12	0	5800	16000
2	2		8	5	4000	14000
2	3		0	0	0	0
3	1		12	0	800	11000
3	2		8	0	4000	14000
3	3		0	0	0	0
4	1		12	0	11050	21250
4	2		9	1	4500	15750
4	3		2	2	0	3000
5	1		12	0	1050	11250
5	2		9	0	4500	15750
5	3		0	0	0	0

The following statements fix the integer variables to their optimal values, call the linear programming solver, and use the `.dual` constraint suffix to compute marginal costs. The RELAXINT option in the SOLVE statement relaxes the integer variables to be continuous.

```

fix NumWorking;
fix NumStartup;
solve with LP relaxint;
print NumWorking NumStartup Excess Output;
print {period in PERIODS} (demand_con[period].dual / length[period]);

```

Figure 15.2 shows the output from the linear programming solver when the integer variables are fixed to their optimal values. As expected, the same optimal solution is obtained; the point of calling the LP solver is to obtain the dual variables used to compute the marginal costs, as shown in Figure 15.3.

Figure 15.2 Output from Linear Programming Solver, Integer Variables Fixed

Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	45
Bounded Above	0
Bounded Below	15
Bounded Below and Above	0
Free	0
Fixed	30
Binary	0
Integer	30
Number of Constraints	40
Linear LE (\leq)	15
Linear EQ ($=$)	0
Linear GE (\geq)	25
Linear Range	0
Constraint Coefficients	120

Solution Summary	
Solver	Dual Simplex
Objective Function	TotalCost
Solution Status	Optimal
Objective Value	988540
Iterations	5
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0

		Num	Num		
[1]	[2]	Working	Startup	Excess	Output
1	1	12	0	0	10200
1	2	3	0	1050	4800
1	3	0	0	0	0
2	1	12	0	5800	16000
2	2	8	5	4000	14000
2	3	0	0	0	0
3	1	12	0	800	11000
3	2	8	0	4000	14000
3	3	0	0	0	0
4	1	12	0	11050	21250
4	2	9	1	4500	15750
4	3	2	2	0	3000
5	1	12	0	1050	11250
5	2	9	0	4500	15750
5	3	0	0	0	0

Figure 15.3 Marginal Costs for Demand Constraints

[1]	
1	1.3
2	2.0
3	2.0
4	2.0
5	2.0

The following statements unfix the integer variables, call the linear programming solver, and use the `.dual` constraint suffix to compute marginal costs:

```

unfix NumWorking;
unfix NumStartup;
solve with LP relaxint;
print NumWorking NumStartup Excess Output;
print {period in PERIODS} (demand_con[period].dual / length[period]);
print {period in PERIODS} (reserve_con[period].dual / length[period]);
quit;

```

Figure 15.4 shows the output from the linear programming solver when the integer variables are unfixed and relaxed to be continuous. As expected, some relaxed integer variables now take fractional values, and the total cost is slightly less than before. Figure 15.5 and Figure 15.6 show the resulting marginal costs.

Figure 15.4 Output from Linear Programming Solver, Integer Variables Unfixed and Relaxed

Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	45
Bounded Above	0
Bounded Below	15
Bounded Below and Above	30
Free	0
Fixed	0
Binary	0
Integer	30
Number of Constraints	40
Linear LE (<=)	15
Linear EQ (=)	0
Linear GE (>=)	25
Linear Range	0
Constraint Coefficients	120

Figure 15.4 continued

Solution Summary					
Solver		Dual Simplex			
Objective Function		TotalCost			
Solution Status		Optimal			
Objective Value		985164.28571			
Iterations		24			
Primal Infeasibility		2.273737E-13			
Dual Infeasibility		0			
Bound Infeasibility		0			
[1]	[2]	Num Working	Num Startup	Excess	Output
1	1	12.0000	0.0000	0.0	10200
1	2	2.7429	0.0000	1371.4	4800
1	3	0.0000	0.0000	0.0	0
2	1	12.0000	0.0000	5000.0	15200
2	2	8.4571	5.7143	4228.6	14800
2	3	0.0000	0.0000	0.0	0
3	1	12.0000	0.0000	0.0	10200
3	2	8.4571	0.0000	4228.6	14800
3	3	0.0000	0.0000	0.0	0
4	1	12.0000	0.0000	11050.0	21250
4	2	9.6000	1.1429	4800.0	16800
4	3	1.3000	1.3000	0.0	1950
5	1	12.0000	0.0000	0.0	10200
5	2	9.6000	0.0000	4800.0	16800
5	3	0.0000	0.0000	0.0	0

Figure 15.5 Marginal Costs for Demand Constraints

[1]	
1	1.7619
2	2.0000
3	1.7857
4	2.0000
5	1.8601

Figure 15.6 Marginal Costs for Reserve Constraints

[1]	
1	0.000000
2	0.000000
3	0.000000
4	0.041667
5	0.000000

Features Demonstrated

The following features are demonstrated in this example:

- problem types: mixed integer linear programming, linear programming
- multiple input data sets
- IMPVAR statement
- modeling startup costs
- IF-THEN/ELSE expression
- CARD function
- FIX and UNFIX statements
- LP solver option RELAXINT
- `.dual` constraint suffix

Chapter 16

Hydro Power: How to Generate and Combine Hydro and Thermal Electricity Generation

Contents

Problem Statement	189
Mathematical Programming Formulation	190
Input Data	192
PROC OPTMODEL Statements and Output	192
Features Demonstrated	197

Problem Statement

This is an extension of the Tariff Rates (Power Generation) problem of Section 12.15 [Chapter 15].¹ In addition to the thermal generators a reservoir powers two hydro generators: one of type A and one of type B. When a hydro generator is running, it operates at a fixed level and the depth of the reservoir decreases. The costs associated with each hydro generator are a fixed start-up cost and a running cost per hour. The characteristics of each type of generator are shown in Table 16.1.

Table 16.1

	Operating level	Cost per hour	Reservoir depth reduction per hour	Start-up cost
Hydro A	900 MW	£90	0.31 metres	£1500
Hydro B	1400 MW	£150	0.47 metres	£1200

For environmental reasons, the reservoir must be maintained at a depth of between 15 and 20 metres. Also, at midnight each night, the reservoir must be 16 metres deep. Thermal generators can be used to pump water into the reservoir. To increase the level of the reservoir by 1 metre requires 3000 MWh of electricity. You may assume that rainfall does not affect the reservoir level.

At any time it must be possible to meet an increase in demand for electricity of up to 15%. This can be achieved by any combination of the following: switching on a hydro generator (even if this would cause the reservoir depth to fall below 15 metres); using the output of a thermal generator which is used for pumping water into the reservoir; and increasing the operating level of a thermal generator to its maximum. Thermal generators cannot be switched on instantaneously to meet increased demand (although hydro generators can be).

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 247–248).

Which generators should be working in which periods of the day, and how should the reservoir be maintained to minimize the total cost?

Mathematical Programming Formulation

This formulation builds on the formulation used in Chapter 15. This section includes only the new elements of the formulation.

Index Sets and Their Members

The following additional index set and its members are used in this example:

- $\text{hydro} \in \text{HYDROS}$

Parameters

Table 16.2 shows the additional parameters used in this example.

Table 16.2 Parameters

Parameter Name	Interpretation
<i>hydro_level[hydro]</i>	Operating level (in MW) per hydro generator
<i>hydro_unit_cost[hydro]</i>	Hourly cost (in pounds) of operating each hydro generator
<i>hydro_depth_rate[hydro]</i>	Reservoir depth reduction (in meters) per hour per hydro generator
<i>hydro_startup_cost[hydro]</i>	Cost of starting up each hydro generator
<i>min_depth</i>	Lower bound on reservoir depth (in meters)
<i>max_depth</i>	Upper bound on reservoir depth (in meters)
<i>midnight_depth</i>	Required depth (in meters) of reservoir at midnight each night
<i>meters_per_mwh</i>	Reservoir level increase (in meters) per megawatt-hour of electricity

Variables

Table 16.3 shows the additional variables used in this example.

Table 16.3 Variables

Variable Name	Interpretation
HydroNumWorking[period,hydro]	Number of units per type of hydro generator operating per period
HydroNumStartup[period,hydro]	Number of units per type of hydro generator starting up per period
Depth[period]	Reservoir depth (in meters) per period
Pump[period]	Number of megawatts used by thermal generators to pump per period
HydroOutput[period,hydro]	Number of megawatt-hours generated by hydro generators

Objective

The objective is to minimize the following function:

$$\begin{aligned}
 \text{TotalCost} = & \sum_{\substack{\text{period} \in \text{PERIODS}, \\ \text{type} \in \text{TYPES}}} (\text{unit_cost}[\text{type}] \cdot \text{length}[\text{period}] \cdot \text{NumWorking}[\text{period}, \text{type}] \\
 & + \text{excess_cost}[\text{type}] \cdot \text{length}[\text{period}] \cdot \text{Excess}[\text{period}, \text{type}] \\
 & + \text{startup_cost}[\text{type}] \cdot \text{NumStartup}[\text{period}, \text{type}]) \\
 + & \sum_{\substack{\text{period} \in \text{PERIODS}, \\ \text{hydro} \in \text{HYDROS}}} (\text{hydro_unit_cost}[\text{hydro}] \cdot \text{length}[\text{period}] \cdot \text{HydroNumWorking}[\text{period}, \text{hydro}] \\
 & + \text{hydro_startup_cost}[\text{hydro}] \cdot \text{HydroNumStartup}[\text{period}, \text{hydro}])
 \end{aligned}$$

Constraints

The following modified and additional constraints are used in this example:

- for $\text{period} \in \text{PERIODS}$ and $\text{hydro} \in \text{HYDROS}$,

$$\text{HydroOutput}[\text{period}, \text{hydro}] = \text{hydro_level}[\text{hydro}] \cdot \text{HydroNumWorking}[\text{period}, \text{hydro}]$$

- for $\text{period} \in \text{PERIODS}$,

$$\begin{aligned}
 & \sum_{\text{type} \in \text{TYPES}} \text{Output}[\text{period}, \text{type}] + \sum_{\text{hydro} \in \text{HYDROS}} \text{HydroOutput}[\text{period}, \text{hydro}] - \text{Pump}[\text{period}] \\
 & \geq \text{demand}[\text{period}]
 \end{aligned}$$

- for $\text{period} \in \text{PERIODS}$,

$$\begin{aligned}
 & \sum_{\text{type} \in \text{TYPES}} \text{max_level}[\text{type}] \cdot \text{NumWorking}[\text{period}, \text{type}] \\
 + & \sum_{\text{hydro} \in \text{HYDROS}} \text{hydro_level}[\text{hydro}] \cdot \text{HydroNumWorking}[\text{period}, \text{hydro}] \cdot \text{ub} \\
 & \geq (1 + \text{reserve}) \cdot \text{demand}[\text{period}]
 \end{aligned}$$

- for period \in PERIODS and hydro \in HYDROS,

$\text{HydroNumStartup}[\text{period}, \text{hydro}]$
 $\geq \text{HydroNumWorking}[\text{period}, \text{hydro}]$
 – (if period $- 1 \in$ PERIODS, then $\text{HydroNumWorking}[\text{period}-1, \text{hydro}]$;
 else $\text{HydroNumWorking}[|\text{PERIODS}|, \text{hydro}]$)

- for period \in PERIODS,

(if period $+ 1 \in$ PERIODS, then $\text{Depth}[\text{period}+1]$; else $\text{Depth}[1]$)
 $= \text{Depth}[\text{period}] + \text{meters_per_mwh} \cdot \text{length}[\text{period}] \cdot \text{Pump}[\text{period}]$
 – $\sum_{\text{hydro} \in \text{HYDROS}} \text{hydro_depth_rate}[\text{hydro}] \cdot \text{length}[\text{period}] \cdot \text{HydroNumWorking}[\text{period}, \text{hydro}]$

Input Data

The following additional data sets and macro variables contain the input data that are used in this example:

```
data hydro_data;
  input hydro $ level unit_cost depth_rate startup_cost;
  datalines;
A  900  90 0.31 1500
B 1400 150 0.47 1200
;

%let min_depth = 15;
%let max_depth = 20;
%let midnight_depth = 16;
%let meters_per_mwh = 1/3000;
```

PROC OPTMODEL Statements and Output

For completeness, all statements are shown. Statements that are new or changed from [Chapter 15](#) are indicated.

```
proc optmodel;
  set PERIODS;
  num length {PERIODS};
  num demand {PERIODS};
  read data period_data into PERIODS=[_N_] length demand;

  set TYPES;
  num num_avail {TYPES};
  num min_level {TYPES};
  num max_level {TYPES};
  num unit_cost {TYPES};
```

```

num excess_cost {TYPES};
num startup_cost {TYPES};
read data type_data into TYPES=[_N_]
    num_avail min_level max_level unit_cost excess_cost startup_cost;

var NumWorking {PERIODS, type in TYPES} >= 0 <= num_avail[type] integer;
var Excess {PERIODS, TYPES} >= 0;
var NumStartup {PERIODS, type in TYPES} >= 0 <= num_avail[type] integer;
impvar Output {period in PERIODS, type in TYPES} =
    min_level[type] * NumWorking[period,type] + Excess[period,type];

```

The following statements declare the additional index set and parameters and then read the additional input data:

```

set <str> HYDROS;
num hydro_level {HYDROS};
num hydro_unit_cost {HYDROS};
num hydro_depth_rate {HYDROS};
num hydro_startup_cost {HYDROS};
read data hydro_data into HYDROS=[hydro]
    hydro_level=level hydro_unit_cost=unit_cost hydro_depth_rate=depth_rate
    hydro_startup_cost=startup_cost;

```

The following statements declare additional variables and fix the value of Depth[1]:

```

var HydroNumWorking {PERIODS, HYDROS} binary;
var HydroNumStartup {PERIODS, HYDROS} binary;
var Depth {PERIODS} >= &min_depth <= &max_depth;
fix Depth[1] = &midnight_depth;
var Pump {PERIODS} >= 0;

```

The following IMPVAR statement declares HydroOutput as a new implicit variable:

```

impvar HydroOutput {period in PERIODS, hydro in HYDROS} =
    hydro_level[hydro] * HydroNumWorking[period,hydro];

```

The following MIN statement is a modification of the objective declaration from [Chapter 15](#):

```

min TotalCost =
    sum {period in PERIODS, type in TYPES} (
        unit_cost[type] * length[period] * NumWorking[period,type]
        + excess_cost[type] * length[period] * Excess[period,type]
        + startup_cost[type] * NumStartup[period,type])
    + sum {period in PERIODS, hydro in HYDROS} (
        hydro_unit_cost[hydro] * length[period] * HydroNumWorking[period,hydro]
        + hydro_startup_cost[hydro] * HydroNumStartup[period,hydro]);

```

The following two CON statements are modified from [Chapter 15](#):

```

con Demand_con {period in PERIODS}:
    sum {type in TYPES} Output[period,type]
    + sum {hydro in HYDROS} HydroOutput[period,hydro]
    - Pump[period]
    >= demand[period];

con Reserve_con {period in PERIODS}:

```

```

    sum {type in TYPES} max_level[type] * NumWorking[period,type]
+ sum {hydro in HYDROS} hydro_level[hydro] * HydroNumWorking[period,hydro].ub
>= (1 + &reserve) * demand[period];

con Excess_ub {period in PERIODS, type in TYPES}:
    Excess[period,type]
<= (max_level[type] - min_level[type]) * NumWorking[period,type];

con Startup_con {period in PERIODS, type in TYPES}:
    NumStartup[period,type]
>= NumWorking[period,type]
- (if period - 1 in PERIODS then NumWorking[period-1,type]
    else NumWorking[card(PERIODS),type]);

```

The following two CON statements declare the final two additional constraints:

```

con Hydro_startup_con {period in PERIODS, hydro in HYDROS}:
    HydroNumStartup[period,hydro]
>= HydroNumWorking[period,hydro]
- (if period - 1 in PERIODS then HydroNumWorking[period-1,hydro]
    else HydroNumWorking[card(PERIODS),hydro]);

con Depth_con {period in PERIODS}:
    (if period + 1 in PERIODS then Depth[period+1] else Depth[1])
= Depth[period]
+ &meters_per_mwh * length[period] * Pump[period]
- sum {hydro in HYDROS}
    hydro_depth_rate[hydro] * length[period] * HydroNumWorking[period,hydro];

```

The following statements call the mixed integer linear programming solver, print the optimal solution, and create several data sets that contain various parts of the optimal solution, with variables grouped according to their index sets:

```

solve;
print NumWorking NumStartup Excess Output;
print HydroNumWorking HydroNumStartup HydroOutput;
print Pump Depth;
create data sol_data1 from [period type] NumWorking NumStartup Excess Output;
create data sol_data2 from [period hydro] HydroNumWorking HydroNumStartup HydroOutput;
create data sol_data3 from [period] Pump Depth;
quit;

```

Figure 16.1 shows the output from the mixed integer linear programming solver.

Figure 16.1 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	75
Bounded Above	0
Bounded Below	20
Bounded Below and Above	54
Free	0
Fixed	1
Binary	20
Integer	30
Number of Constraints	55
Linear LE (\leq)	15
Linear EQ ($=$)	5
Linear GE (\geq)	35
Linear Range	0
Constraint Coefficients	190
Solution Summary	
Solver	MILP
Objective Function	TotalCost
Solution Status	Optimal_Rgap
Objective Value	986630
Iterations	194
Best Bound	986615.13202
Nodes	34
Relative Gap	0.0000150697
Absolute Gap	14.867975626
Primal Infeasibility	1.136868E-12
Bound Infeasibility	0
Integer Infeasibility	0

Figure 16.1 *continued*

[1]	[2]	Num Working	Num Startup	Excess	Output
1	1	12	0	0	10200
1	2	3	0	1500	5250
1	3	0	0	0	0
2	1	12	0	4050	14250
2	2	9	6	4500	15750
2	3	0	0	0	0
3	1	12	0	365	10565
3	2	9	0	4500	15750
3	3	0	0	0	0
4	1	12	0	11150	21350
4	2	9	0	4500	15750
4	3	1	1	0	1500
5	1	12	0	0	10200
5	2	9	0	4500	15750
5	3	0	0	0	0

[1]	[2]	Hydro Num Working	Hydro Num Startup	Hydro Output
1	A	0	0	0
1	B	0	0	0
2	A	0	0	0
2	B	0	0	0
3	A	0	0	0
3	B	0	0	0
4	A	0	0	0
4	B	1	1	1400
5	A	0	0	0
5	B	1	0	1400

[1]	Pump	Depth
1	450	16.00
2	0	16.90
3	1315	16.90
4	0	19.53
5	350	18.12

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- multiple input and output data sets
- FIX statement
- IMPVAR statement
- `.ub` variable suffix
- modeling startup costs
- IF-THEN/ELSE expression
- CARD function

Chapter 17

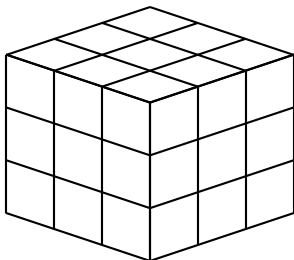
Three-Dimensional Noughts and Crosses: A Combinatorial Problem

Contents	
Problem Statement	199
Mathematical Programming Formulation	199
Input Data	201
PROC OPTMODEL Statements and Output	201
Features Demonstrated	205

Problem Statement

Twenty-seven cells are arranged $3 \times 3 \times 3$ in a three-dimensional array as shown in Figure 17.1.¹

Figure 17.1



Three cells are regarded as lying in the same line if they are on the same horizontal or vertical line or the same diagonal. Diagonals exist on each horizontal and vertical section and connecting opposite vertices of the cube. (There are 49 lines altogether.)

Given 13 white balls (noughts) and 14 black balls (crosses), arrange them, one to a cell, so as to minimize the number of lines with balls all of one colour.

Mathematical Programming Formulation

The formulation shown here generalizes the problem to an $n \times n \times n$ array and an arbitrary number of colors.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 248–249).

Index Sets and Their Members

The following index sets and their members are used in this example:

- $(i, j, k) \in \text{CELLS}$
- $\text{color} \in \text{COLORS}$
- $\text{line} \in \text{LINES}$
- $(i, j, k) \in \text{CELLS_line}[\text{line}]$

Parameters

Table 17.1 shows the parameters used in this example.

Table 17.1 Parameters

Parameter Name	Interpretation
n	Number of cells per line
num_lines	Number of lines
$\text{num_balls}[\text{color}]$	Number of balls of each color
$\text{assigned_color}[i,j,k]$	Color assigned to cell (i, j, k)

Variables

Table 17.2 shows the variables used in this example.

Table 17.2 Variables

Variable Name	Interpretation
$\text{IsColor}[i,j,k,\text{color}]$	1 if cell (i, j, k) is color; 0 otherwise
$\text{IsMonochromatic}[\text{line}]$	1 if line is monochromatic; 0 otherwise

Objective

The objective is to minimize the following function:

$$\text{NumMonochromaticLines} = \sum_{\text{line} \in \text{LINES}} \text{IsMonochromatic}[\text{line}]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $(i, j, k) \in \text{CELLS}$,

$$\sum_{\text{color} \in \text{COLORS}} \text{IsColor}[i, j, k, \text{color}] = 1$$

- for $\text{color} \in \text{COLORS}$,

$$\sum_{(i, j, k) \in \text{CELLS}} \text{IsColor}[i, j, k, \text{color}] = \text{num_balls}[\text{color}]$$

- for $\text{line} \in \text{LINES}$ and $\text{color} \in \text{COLORS}$,

$$\sum_{(i, j, k) \in \text{CELLS_line}[\text{line}]} \text{IsColor}[i, j, k, \text{color}] - |\text{CELLS_line}[\text{line}]| + 1 \leq \text{IsMonochromatic}[\text{line}]$$

Input Data

The following data set and macro variable contain the input data that are used in this example:

```
data color_data;
  input num_balls;
  datalines;
13
14
;

%let n = 3;
```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare parameters and index sets and then read input data:

```
proc optmodel;
  num n = &n;
  set CELLS = 1..n cross 1..n cross 1..n;

  set COLORS;
  num num_balls {COLORS};
  read data color_data into COLORS=[_N_] num_balls;
```

The following statements declare the IsColor variables and the first two families of constraints:

```

var IsColor {CELLS, COLORS} binary;
con IsColor_con {<i,j,k> in CELLS}:
    sum {color in COLORS} IsColor[i,j,k,color] = 1;
con Num_balls_con {color in COLORS}:
    sum {<i,j,k> in CELLS} IsColor[i,j,k,color] = num_balls[color];

```

The following statements declare the IsMonochromatic variables and the objective:

```

num num_lines init 0;
set LINES = 1..num_lines;
var IsMonochromatic {LINES} binary;

min NumMonochromaticLines = sum {line in LINES} IsMonochromatic[line];

```

The following SET statement declares (but does not populate) an index set that will contain the cells for each line:

```

set <num,num,num> CELLS_line {LINES};

```

The following FOR loops use the SETOF operator to populate CELLS_line for each row:

```

for {i in 1..n, j in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {k in 1..n} <i,j,k>;
end;
for {i in 1..n, k in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {j in 1..n} <i,j,k>;
end;
for {j in 1..n, k in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {i in 1..n} <i,j,k>;
end;

```

The following FOR loops use the SETOF operator to populate CELLS_line for each face diagonal:

```

for {i in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {j in 1..n} <i,j,j>;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {j in 1..n} <i,j,n+1-j>;
end;
for {j in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {i in 1..n} <i,j,i>;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {i in 1..n} <i,j,n+1-i>;
end;
for {k in 1..n} do;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {i in 1..n} <i,i,k>;
    num_lines = num_lines + 1;
    CELLS_line[num_lines] = setof {i in 1..n} <i,n+1-i,k>;
end;

```

The following statements use the SETOF operator to populate CELLS_line for each cube diagonal:

```
num_lines = num_lines + 1;
CELLS_line[num_lines] = setof {t in 1..n} <t,t,t>;
num_lines = num_lines + 1;
CELLS_line[num_lines] = setof {t in 1..n} <t,t,n+1-t>;
num_lines = num_lines + 1;
CELLS_line[num_lines] = setof {t in 1..n} <t,n+1-t,t>;
num_lines = num_lines + 1;
CELLS_line[num_lines] = setof {t in 1..n} <t,n+1-t,n+1-t>;
```

The following PUT statements demonstrate that num_lines is $((n+2)^3 - n^3)/2$, which is 49 when $n = 3$.

```
put num_lines=;
put (((n+2)^3 - n^3) / 2)=;
```

The following CON statement enforces the rule that if all cells in a line are the same color, then $IsMonochromatic[line] = 1$:

```
con Link_con {line in LINES, color in COLORS}:
    sum {<i,j,k> in CELLS_line[line]} IsColor[i,j,k,color]
    - card(CELLS_line[line]) + 1
    <= IsMonochromatic[line];
```

The following statements call the mixed integer linear programming solver and use the `.sol` variable suffix to populate $assigned_color[i,j,k]$:

```
solve;
num assigned_color {CELLS};
for {<i,j,k> in CELLS} do;
    for {color in COLORS: IsColor[i,j,k,color].sol > 0.5} do;
        assigned_color[i,j,k] = color;
    leave;
end;
end;
```

The following statements print the values of $assigned_color[i,j,k]$ for each i and display in the log which cells make up each monochromatic line:

```
for {i in 1..n}
    print {j in 1..n, k in 1..n} assigned_color[i,j,k];
for {line in LINES: IsMonochromatic[line].sol > 0.5}
    put CELLS_line[line]=;
```

By default, the PUT statement writes to the log. You can use the following FILE statement to redirect log output to the listing:

```
file print;
for {line in LINES: IsMonochromatic[line].sol > 0.5}
    put CELLS_line[line]=;
quit;
```

Figure 17.2 shows the output from the mixed integer linear programming solver.

Figure 17.2 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure			
Problem Summary			
Objective Sense	Minimization		
Objective Function	NumMonochromaticLines		
Objective Type	Linear		
Number of Variables	103		
Bounded Above	0		
Bounded Below	0		
Bounded Below and Above	103		
Free	0		
Fixed	0		
Binary	103		
Integer	0		
Number of Constraints	127		
Linear LE (\leq)	98		
Linear EQ ($=$)	29		
Linear GE (\geq)	0		
Linear Range	0		
Constraint Coefficients	500		
Solution Summary			
Solver	MILP		
Objective Function	NumMonochromaticLines		
Solution Status	Optimal		
Objective Value	4		
Iterations	66779		
Best Bound	.		
Nodes	6451		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	2.999823E-12		
Bound Infeasibility	2.999934E-12		
Integer Infeasibility	2.999934E-12		
assigned_color			
	1	2	3
1	2	1	2
2	1	1	1
3	1	2	2

Figure 17.2 *continued*

assigned_color			
	1	2	3
1	1	2	1
2	2	2	1
3	2	1	2

assigned_color			
	1	2	3
1	1	1	2
2	2	2	2
3	1	2	1

Figure 17.3 shows the output from the PUT statement.

Figure 17.3 Output from PUT Statement

```

The OPTMODEL Procedure

CELLS_line[2]={<1,2,1>,<1,2,2>,<1,2,3>}
CELLS_line[8]={<3,2,1>,<3,2,2>,<3,2,3>}
CELLS_line[34]={<1,1,1>,<2,1,2>,<3,1,3>}
CELLS_line[38]={<1,3,1>,<2,3,2>,<3,3,3>}

```

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming
- sets of tuples
- set operators CROSS and SETOF
- sets indexed by other sets
- modeling if-then constraints using binary variables
- CARD function
- PUT statement
- `.sol` variable suffix
- FILE statement

Chapter 18

Optimizing a Constraint: Reconstructing an Integer Programming Constraint More Simply

Contents

Problem Statement	207
Mathematical Programming Formulation	207
Input Data	209
PROC OPTMODEL Statements and Output	209
Features Demonstrated	214

Problem Statement

In an integer programming problem the following constraint occurs:¹

$$9x_1 + 13x_2 - 14x_3 + 17x_4 + 13x_5 - 19x_6 + 23x_7 + 21x_8 \leq 37.$$

All the variables occurring in this constraint are 0-1 variables, i.e. they can only take the value of 0 or 1.

Find the ‘simplest’ version of this constraint. The objective is to find another constraint involving these variables which is logically equivalent to the original constraint but which has the smallest possible absolute value of the right-hand side (with all coefficients of similar signs to the original coefficients).

If the objective were to find an equivalent constraint where the sum of the absolute values of the coefficients (apart from the right-hand side coefficient) were a minimum what would be the result?

Mathematical Programming Formulation

The formulation shown here differs from Williams and does not require transforming the original constraint into a standard form with positive coefficients. The new constraint to be found is

$$\sum_{j \in \text{VARS}} \text{Alpha}[j] \cdot x_j \leq \text{Alpha}[0]$$

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, p. 249).

Index Sets and Their Members

The following index sets and their members are used in this example:

- $j \in \text{VARS}$
- $j \in \text{VARS0} = \text{VARS} \cup \{0\}$
- $\text{point} \in \text{FEAS_POINTS}$
- $\text{point} \in \text{INFEAS_POINTS}$

Parameters

Table 18.1 shows the parameters used in this example.

Table 18.1 Parameters

Parameter Name	Interpretation
n	Number of decision variables in constraint
$a[j]$	Original constraint coefficient of variable j , with $a[0]$ equal to original right-hand side
$x_feas[\text{point}, j]$	Value of variable j in feasible point
$x_infeas[\text{point}, j]$	Value of variable j in infeasible point

Variables

Table 18.2 shows the variables used in this example.

Table 18.2 Variables

Variable Name	Interpretation
Scale[j]	Nonnegative scalar multiplier for coefficient of variable j
Alpha[j]	Coefficient of variable j in new constraint

Objectives

The first objective is to minimize the absolute value of the new right-hand side:

$$\text{Objective1} = |a[0]| \cdot \text{Scale}[0]$$

The second objective is to minimize the sum of absolute values of the new left-hand side coefficients:

$$\text{Objective2} = \sum_{j \in \text{VARS}} |a[j]| \cdot \text{Scale}[j]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $j \in \text{VAR}S0$,

$$\text{Alpha}[j] = a[j] \cdot \text{Scale}[j]$$

- for $\text{point} \in \text{FEAS_POINTS}$,

$$\sum_{j \in \text{VAR}S} \text{Alpha}[j] \cdot x_{\text{feas}}[\text{point}, j] \leq \text{Alpha}[0]$$

- for $\text{point} \in \text{INFEAS_POINTS}$,

$$\sum_{j \in \text{VAR}S} \text{Alpha}[j] \cdot x_{\text{infeas}}[\text{point}, j] \geq \text{Alpha}[0] + 1$$

Input Data

The following data set and macro variables contain the input data that are used in this example:

```
data a_data;
  input a @@;
  datalines;
9 13 -14 17 13 -19 23 21
;

%let b = 37;

/* populate macro variable n as number of decision variables */
%let dsid = %sysfunc(open(a_data));
%let n = %sysfunc(attrn(&dsid, NOBS));
%let rc = %sysfunc(close(&dsid));
```

PROC OPTMODEL Statements and Output

The following PROC TRANSPOSE statements and DATA step create an input data set for the CLP procedure in SAS/OR software:

```

proc transpose data=a_data out=trans(drop=_name_) prefix=x;
run;

data condata_feas(drop=j);
  length _type_ $8;
  array x[&n];
  set trans;
  _type_ = 'le';
  _rhs_ = &b;
  output;
  do j = 1 to &n;
    x[j] = 1;
  end;
  _type_ = 'binary';
  _rhs_ = .;
  output;
run;

```

The following PROC CLP statements find all feasible solutions and save them in the out_feas data set:

```

proc clp data=condata_feas out=out_feas allsolns usecondatavars=1;
run;

```

The following DATA step creates another input data set for PROC CLP, for the purpose of finding all binary solutions that violate the original constraint:

```

data condata_infeas;
  set condata_feas;
  if _N_ = 1 then do;
    _type_ = 'ge';
    _rhs_ = _rhs_ + 1;
  end;
run;

```

The following PROC CLP statements find all such solutions and save them in the out_infeas data set:

```

proc clp data=condata_infeas out=out_infeas allsolns usecondatavars=1;
run;

```

The first several PROC OPTMODEL statements declare sets and parameters and read the original constraint data:

```

proc optmodel;
  set VARS;
  set VARS0 = VARS union {0};
  num a {VARS0};
  read data a_data into VARS=[_N_] a;
  a[0] = &b;

```

The following statements declare the FEAS_POINTS set and the x_feas parameter and populate them by reading the out_feas data set:

```

set FEAS_POINTS;
num x_feas {FEAS_POINTS, VARS};
read data out_feas into FEAS_POINTS=[_N_]
  {j in VARS} <x_feas[_N_,j]=col('x' || j)>;

```

The following statements declare the INFEAS_POINTS set and the x_{infeas} parameter and populate them by reading the out_infeas data set:

```

set INFEAS_POINTS;
num x_infeas {INFEAS_POINTS, VARS};
read data out_infeas into INFEAS_POINTS=[_N_]
  {j in VARS} <x_infeas[_N_,j]=col('x' || j)>;

```

The following statements declare the variables and constraints:

```

var Scale {VARS0} >= 0;
impvar Alpha {j in VARS0} = a[j] * Scale[j];

con Feas_con {point in FEAS_POINTS}:
  sum {j in VARS} Alpha[j] * x_feas[point,j] <= Alpha[0];
con Infeas_con {point in INFEAS_POINTS}:
  sum {j in VARS} Alpha[j] * x_infeas[point,j] >= Alpha[0] + 1;

```

The following statements solve the problem by using the first objective and then print the solution:

```

min Objective1 = abs(a[0]) * Scale[0];
solve;
print a Scale Alpha;

```

Figure 18.1 shows the output from the linear programming solver for the first objective.

Figure 18.1 Output from Linear Programming Solver, First Objective

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	Objective1
Objective Type	Linear
Number of Variables	9
Bounded Above	0
Bounded Below	9
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	256
Linear LE (<=)	152
Linear EQ (=)	0
Linear GE (>=)	104
Linear Range	0
Constraint Coefficients	1280

Figure 18.1 *continued*

Solution Summary			
Solver	Dual Simplex		
Objective Function	Objective1		
Solution Status	Optimal		
Objective Value	25		
Iterations	19		
Primal Infeasibility	7.771561E-15		
Dual Infeasibility	0		
Bound Infeasibility	0		
[1]	a	Scale	Alpha
0	37	0.67568	25
1	9	0.66667	6
2	13	0.69231	9
3	-14	0.71429	-10
4	17	0.70588	12
5	13	0.69231	9
6	-19	0.68421	-13
7	23	0.69565	16
8	21	0.66667	14

The following statements solve the problem by using the second objective and then print the solution:

```
min Objective2 = sum {j in VARS} abs(a[j]) * Scale[j];
solve;
print a Scale Alpha;
quit;
```

Figure 18.2 shows the output from the linear programming solver for the second objective.

Figure 18.2 Output from Linear Programming Solver, Second Objective

Problem Summary	
Objective Sense	Minimization
Objective Function	Objective2
Objective Type	Linear
Number of Variables	9
Bounded Above	0
Bounded Below	9
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	256
Linear LE (\leq)	152
Linear EQ ($=$)	0
Linear GE (\geq)	104
Linear Range	0
Constraint Coefficients	1280

Solution Summary	
Solver	Dual Simplex
Objective Function	Objective2
Solution Status	Optimal
Objective Value	89
Iterations	23
Primal Infeasibility	3.108624E-15
Dual Infeasibility	0
Bound Infeasibility	0

[1]	a	Scale	Alpha
0	37	0.67568	25
1	9	0.66667	6
2	13	0.69231	9
3	-14	0.71429	-10
4	17	0.70588	12
5	13	0.69231	9
6	-19	0.68421	-13
7	23	0.69565	16
8	21	0.66667	14

For this test instance, it turns out that the optimal solutions for the two different objectives agree.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- CLP procedure
- set operator UNION
- IMPVAR statement
- ABS function
- reading multiple data sets
- multiple objectives

Chapter 19

Distribution 1: Which Factories and Depots to Supply Which Customers

Contents

Problem Statement	215
Mathematical Programming Formulation	218
Input Data	220
PROC OPTMODEL Statements and Output	221
Features Demonstrated	230

Problem Statement

A company has two factories, one at Liverpool and one at Brighton.¹ In addition it has four depots with storage facilities at Newcastle, Birmingham, London and Exeter. The company sells its product to six customers C1, C2, . . . , C6. Customers can be supplied either from a depot or from the factory direct (see Figure 19.1).

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 249–251).

Figure 19.1

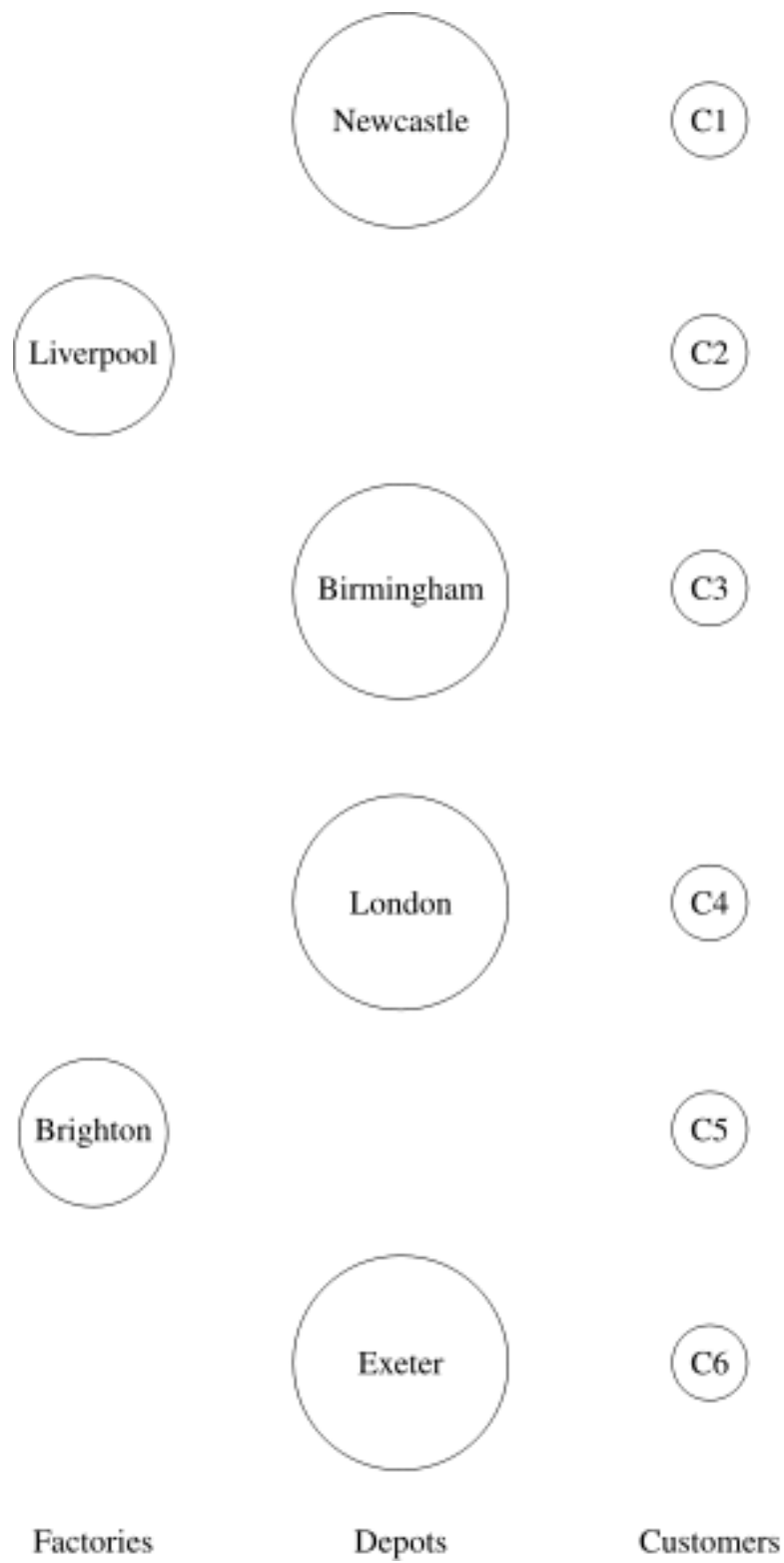


Table 19.1

Supplied to	Supplier ^a					
	Liverpool factory	Brighton factory	Newcastle depot	Birmingham depot	London depot	Exeter depot
<i>Depots</i>						
Newcastle	0.5	—				
Birmingham	0.5	0.3				
London	1.0	0.5				
Exeter	0.2	0.2				
<i>Customers</i>						
C1	1.0	2.0	—	1.0	—	—
C2	—	—	1.5	0.5	1.5	—
C3	1.5	—	0.5	0.5	2.0	0.2
C4	2.0	—	1.5	1.0	—	1.5
C5	—	—	—	0.5	0.5	0.5
C6	1.0	—	1.0	—	1.5	1.5

^aA dash indicates the impossibility of certain suppliers for certain depots or customers.

The distribution costs (which are borne by the company) are known; they are given in Table 19.1 (in £ per ton delivered).

Certain customers have expressed preferences for being supplied from factories or depots which they are used to. The preferred suppliers are

- C1 Liverpool (factory)
- C2 Newcastle (depot)
- C3 No preferences
- C4 No preferences
- C5 Birmingham (depot)
- C6 Exeter or London (depots)

Each factory has a monthly capacity given below which cannot be exceeded:

Liverpool 150,000 tons
Brighton 200,000 tons

Each depot has a maximum monthly throughput given below which cannot be exceeded:

Newcastle 70,000 tons
Birmingham 50,000 tons
London 100,000 tons
Exeter 40,000 tons

Each customer has a monthly requirement given below which must be met:

C1	50,000 tons
C2	10,000 tons
C3	40,000 tons
C4	35,000 tons
C5	60,000 tons
C6	20,000 tons

The company would like to determine:

- (1) What distribution pattern would minimize overall cost?
- (2) What the effect of increasing factory and depot capacities would be on distribution costs?
- (3) What the effects of small changes in costs, capacities and requirements would be on the distribution pattern?
- (4) Would it be possible to meet all customers preferences regarding suppliers and if so what would the extra cost of doing this be?

Mathematical Programming Formulation

The problem is represented as a network flow problem with side constraints. Each node corresponds to a factory, depot, or customer, and each arc represents a shipment of the product from one place to another.

Index Sets and Their Members

The following index sets and their members are used in this example:

- $(i, j) \in \text{ARCS}$
- $i, \text{factory} \in \text{FACTORIES}$
- $i, \text{depot} \in \text{DEPOTS}$
- $i, \text{customer} \in \text{CUSTOMERS}$
- $i \in \text{NODES}$
- $(i, j) \in \text{PREFERRED_ARCS}$
- $j \in \text{CUSTOMERS_WITH_PREFERENCES}$

Parameters

Table 19.2 shows the parameters used in this example.

Table 19.2 Parameters

Parameter Name	Interpretation
$cost[i,j]$	Distribution costs (in £ per ton delivered)
$capacity[factory]$	Monthly capacity per factory (in tons)
$throughput[depot]$	Maximum monthly throughput per depot (in tons)
$demand[customer]$	Monthly demand per customer (in tons)
$supply[i]$	Supply at node i (in tons)

Variables

Table 19.3 shows the variables used in this example.

Table 19.3 Variables

Variable Name	Interpretation
$Flow[i,j]$	Flow across arc (i, j)

Objectives

One objective is to minimize the following function:

$$\text{TotalCost} = \sum_{(i,j) \in \text{ARCS}} cost[i,j] \cdot \text{Flow}[i,j]$$

Another objective is to minimize the following function:

$$\text{NonpreferredFlow} = \sum_{\substack{(i,j) \in \text{ARCS} \setminus \text{PREFERRED_ARCS} \\ j \in \text{CUSTOMERS_WITH_PREFERENCES}}} \text{Flow}[i,j]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $i \in \text{NODES}$,

$$\sum_{(i,j) \in \text{ARCS}} \text{Flow}[i,j] - \sum_{(j,i) \in \text{ARCS}} \text{Flow}[j,i] \leq supply[i]$$

- for $i \in \text{DEPOTS}$,

$$\sum_{(i,j) \in \text{ARCS}} \text{Flow}[i,j] \leq \text{throughput}[i]$$

- $\text{NonpreferredFlow} \leq \text{NonpreferredFlow.sol}$

Input Data

The following data sets contain the input data that are used in this example:

```
data arc_data;
    input j $11. i $11. cost;
    datalines;
Newcastle Liverpool 0.5
Birmingham Liverpool 0.5
Birmingham Brighton 0.3
London Liverpool 1.0
London Brighton 0.5
Exeter Liverpool 0.2
Exeter Brighton 0.2
C1 Liverpool 1.0
C1 Brighton 2.0
C1 Birmingham 1.0
C2 Newcastle 1.5
C2 Birmingham 0.5
C2 London 1.5
C3 Liverpool 1.5
C3 Newcastle 0.5
C3 Birmingham 0.5
C3 London 2.0
C3 Exeter 0.2
C4 Liverpool 2.0
C4 Newcastle 1.5
C4 Birmingham 1.0
C4 Exeter 1.5
C5 Birmingham 0.5
C5 London 0.5
C5 Exeter 0.5
C6 Liverpool 1.0
C6 Newcastle 1.0
C6 London 1.5
C6 Exeter 1.5
;

data customer_data;
    input customer $ demand;
    datalines;
C1 50000
C2 10000
```

```

C3 40000
C4 35000
C5 60000
C6 20000
;

data factory_data;
    input factory $10. capacity;
    datalines;
Liverpool 150000
Brighton 200000
;

data depot_data;
    input depot $11. throughput;
    datalines;
Newcastle 70000
Birmingham 50000
London 100000
Exeter 40000
;

data preferred_arc_data;
    input j $ i $11.;
    datalines;
C1 Liverpool
C2 Newcastle
C5 Birmingham
C6 Exeter
C6 London
;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```

proc optmodel;
    set <str,str> ARCS;
    num cost {ARCS};
    read data arc_data into ARCS=[i j] cost;

    set <str> FACTORIES;
    num capacity {FACTORIES};
    read data factory_data into FACTORIES=[factory] capacity;

    set <str> DEPOTS;
    num throughput {DEPOTS};
    read data depot_data into DEPOTS=[depot] throughput;

    set <str> CUSTOMERS;

```

```

num demand {CUSTOMERS};
read data customer_data into CUSTOMERS=[customer] demand;

```

The following statements use the UNION set operator to declare the `NODES` index set, declare the *supply* parameter with an initial value of 0, and populate *supply* for both factories and customers:

```

set NODES = FACTORIES union DEPOTS union CUSTOMERS;
num supply {NODES} init 0;
for {i in FACTORIES} supply[i] = capacity[i];
for {i in CUSTOMERS} supply[i] = -demand[i];

```

The following statements declare the variables, constraints, and TotalCost objective:

```

var Flow {ARCS} >= 0;

con Flow_balance_con {i in NODES}:
    sum {<(i),j> in ARCS} Flow[i,j] - sum {<j,(i)> in ARCS} Flow[j,i] <= supply[i];

con Depot_con {i in DEPOTS}:
    sum {<(i),j> in ARCS} Flow[i,j] <= throughput[i];

min TotalCost = sum {<i,j> in ARCS} cost[i,j] * Flow[i,j];

```

The following statements call the default linear programming solver (which is dual simplex), print the positive variables in the resulting optimal solution, and print the left-hand side (*.body*), right-hand side (*.ub*), and dual value (*.dual*) of each constraint:

```

put 'Minimizing TotalCost...';
solve;
print {<i,j> in ARCS: Flow[i,j].sol > 0} Flow;
print Flow_balance_con.body Flow_balance_con.ub Flow_balance_con.dual;
print Depot_con.body Depot_con.ub Depot_con.dual;

```

Figure 19.2 shows the output when you use the (default) dual simplex solver.

Figure 19.2 Output from Dual Simplex Solver, Minimizing TotalCost

The OPTMODEL Procedure		
Problem Summary		
Objective Sense	Minimization	
Objective Function	TotalCost	
Objective Type	Linear	
Number of Variables		29
Bounded Above		0
Bounded Below		29
Bounded Below and Above		0
Free		0
Fixed		0
Number of Constraints		16
Linear LE (<=)		16
Linear EQ (=)		0
Linear GE (>=)		0
Linear Range		0
Constraint Coefficients		75
Solution Summary		
Solver	Dual Simplex	
Objective Function	TotalCost	
Solution Status	Optimal	
Objective Value	198500	
Iterations	12	
Primal Infeasibility	0	
Dual Infeasibility	0	
Bound Infeasibility	0	
[1]	[2]	Flow
Birmingham	C2	10000
Birmingham	C4	35000
Birmingham	C5	5000
Brighton	Birmingham	50000
Brighton	Exeter	40000
Brighton	London	55000
Exeter	C3	40000
Liverpool	C1	50000
Liverpool	C6	20000
London	C5	55000

Figure 19.2 *continued*

[1]	Flow_balance_ con.BODY	Flow_ balance_ con.UB	Flow_balance_ con.DUAL
Birmingham	0	0	-0.3
Brighton	145000	200000	0.0
C1	-50000	-50000	-1.0
C2	-10000	-10000	-1.0
C3	-40000	-40000	-0.7
C4	-35000	-35000	-1.5
C5	-60000	-60000	-1.0
C6	-20000	-20000	-1.0
Exeter	0	0	-0.2
Liverpool	70000	150000	0.0
London	0	0	-0.5
Newcastle	0	0	-0.2

[1]	Depot_ con. BODY	Depot_ con.UB	Depot_ con. DUAL
Birmingham	50000	50000	-0.2
Exeter	40000	40000	-0.3
London	55000	100000	0.0
Newcastle	0	70000	0.0

The following statements call the network simplex linear programming solver and print the same solution information as before:

```
put 'Minimizing TotalCost using network simplex...';
solve with LP / solver=ns;
print {<i,j> in ARCS: Flow[i,j].sol > 0} Flow;
print Flow_balance_con.body Flow_balance_con.ub Flow_balance_con.dual;
print Depot_con.body Depot_con.ub Depot_con.dual;
```

Figure 19.3 shows the output when you use the SOLVER=NS option to invoke the network simplex solver.

Figure 19.3 Output from Network Simplex Solver, Minimizing TotalCost

Problem Summary		
Objective Sense	Minimization	
Objective Function	TotalCost	
Objective Type	Linear	
Number of Variables	29	
Bounded Above	0	
Bounded Below	29	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	16	
Linear LE (\leq)	16	
Linear EQ ($=$)	0	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	75	
Solution Summary		
Solver	Network Simplex	
Objective Function	TotalCost	
Solution Status	Optimal	
Objective Value	198500	
Iterations	16	
Iterations2	4	
Primal Infeasibility	0	
Dual Infeasibility	0	
Bound Infeasibility	0	
[1]	[2]	Flow
Birmingham	C2	10000
Birmingham	C4	35000
Birmingham	C5	5000
Brighton	Birmingham	50000
Brighton	Exeter	40000
Brighton	London	55000
Exeter	C3	40000
Liverpool	C1	50000
Liverpool	C6	20000
London	C5	55000

Figure 19.3 continued

[1]	Flow_balance_ con.BODY	Flow_ balance_ con.UB	Flow_balance_ con.DUAL
Birmingham	0	0	-0.3
Brighton	145000	200000	0.0
C1	-50000	-50000	-1.0
C2	-10000	-10000	-1.0
C3	-40000	-40000	-1.0
C4	-35000	-35000	-1.5
C5	-60000	-60000	-1.0
C6	-20000	-20000	-1.0
Exeter	0	0	-0.2
Liverpool	70000	150000	0.0
London	0	0	-0.5
Newcastle	0	0	-0.5

[1]	Depot_ con. BODY	Depot_ con.UB	Depot_ con. DUAL
Birmingham	50000	50000	-0.2
Exeter	40000	40000	-0.6
London	55000	100000	0.0
Newcastle	0	70000	0.0

The following statements call the linear programming solver to minimize NonpreferredFlow and print both objectives, the positive variables in the resulting optimal solution, and the flow along nonpreferred arcs:

```

set <str,str> PREFERRED_ARCS;
read data preferred_arc_data into PREFERRED_ARCS=[i j];
set CUSTOMERS_WITH_PREFERENCES = setof {<i,j> in PREFERRED_ARCS} j;
min NonpreferredFlow =
    sum {<i,j> in ARCS diff PREFERRED_ARCS: j in CUSTOMERS_WITH_PREFERENCES}
        Flow[i,j];

put 'Minimizing NonpreferredFlow...';
solve;
print TotalCost NonpreferredFlow;
print {<i,j> in ARCS: Flow[i,j].sol > 0} Flow;
print
    {<i,j> in ARCS diff PREFERRED_ARCS: j in CUSTOMERS_WITH_PREFERENCES} Flow;
```

Figure 19.4 shows the output from the linear programming solver.

Figure 19.4 Output from Linear Programming Solver, Minimizing NonpreferredFlow

Problem Summary		
Objective Sense	Minimization	
Objective Function	NonpreferredFlow	
Objective Type	Linear	
Number of Variables	29	
Bounded Above	0	
Bounded Below	29	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	16	
Linear LE (\leq)	16	
Linear EQ ($=$)	0	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	75	
Solution Summary		
Solver	Dual Simplex	
Objective Function	NonpreferredFlow	
Solution Status	Optimal	
Objective Value	10000	
Iterations	13	
Primal Infeasibility	0	
Dual Infeasibility	0	
Bound Infeasibility	0	
Total Cost	Nonpreferred Flow	
293000	10000	
[1]	[2]	Flow
Birmingham	C5	50000
Brighton	Birmingham	50000
Brighton	Exeter	15000
Exeter	C5	10000
Exeter	C6	5000
Liverpool	C1	50000
Liverpool	C3	40000
Liverpool	C4	35000
Liverpool	London	15000
Liverpool	Newcastle	10000
London	C6	15000
Newcastle	C2	10000

Figure 19.4 *continued*

	[1]	[2]	Flow
	Birmingham	C1	0
	Birmingham	C2	0
	Brighton	C1	0
	Exeter	C5	10000
	Liverpool	C6	0
	London	C2	0
	London	C5	0
	Newcastle	C6	0

The following CON statement declares a constraint that limits NonpreferredFlow to the minimum value found in the previous solve:

```
con Objective_cut:
    NonpreferredFlow <= NonpreferredFlow.sol;
```

The following statements call the linear programming solver to minimize TotalCost with a constrained NonpreferredFlow and print the same solution information as before:

```
put 'Minimizing TotalCost with constrained NonpreferredFlow...';
solve obj TotalCost;
print TotalCost NonpreferredFlow;
print {<i,j> in ARCS: Flow[i,j].sol > 0} Flow;
print
    {<i,j> in ARCS diff PREFERRED_ARCS: j in CUSTOMERS_WITH_PREFERENCES} Flow;
quit;
```

Figure 19.5 shows the output from the linear programming solver. As expected, NonpreferredFlow remains at its minimum value and TotalCost is less than its value from Figure 19.4.

Figure 19.5 Output from Linear Programming Solver, Minimizing TotalCost with Constrained NonpreferredFlow

Problem Summary		
Objective Sense	Minimization	
Objective Function	TotalCost	
Objective Type	Linear	
Number of Variables	29	
Bounded Above	0	
Bounded Below	29	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	17	
Linear LE (\leq)	17	
Linear EQ ($=$)	0	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	83	
Solution Summary		
Solver	Dual Simplex	
Objective Function	TotalCost	
Solution Status	Optimal	
Objective Value	246000	
Iterations	17	
Primal Infeasibility	0	
Dual Infeasibility	0	
Bound Infeasibility	0	
Total Cost	Nonpreferred Flow	
246000	10000	
[1]	[2]	Flow
Birmingham	C5	50000
Brighton	Birmingham	50000
Brighton	Exeter	40000
Brighton	London	30000
Exeter	C3	40000
Liverpool	C1	50000
Liverpool	C4	35000
Liverpool	Newcastle	10000
London	C5	10000
London	C6	20000
Newcastle	C2	10000

Figure 19.5 *continued*

	[1]	[2]	Flow
	Birmingham	C1	0
	Birmingham	C2	0
	Brighton	C1	0
	Exeter	C5	0
	Liverpool	C6	0
	London	C2	0
	London	C5	10000
	Newcastle	C6	0

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming (network flow with side constraints)
- sets of tuples
- reading multiple data sets
- set operators UNION, SETOF, and DIFF
- INIT option
- implicit slice
- using a colon (:) to select members of a set
- **.body** constraint suffix
- **.ub** constraint suffix
- **.dual** constraint suffix
- multiple objectives and the OBJ option
- SOLVER= option

Chapter 20

Depot Location (Distribution 2): Where Should New Depots Be Built

Contents

Problem Statement	231
Mathematical Programming Formulation	232
Input Data	234
PROC OPTMODEL Statements and Output	235
Features Demonstrated	239

Problem Statement

In the distribution problem there is a possibility of opening new depots at Bristol and Northampton as well as of enlarging the Birmingham depot.¹

It is not considered desirable to have more than four depots and if necessary Newcastle or Exeter (or both) can be closed down.

The monthly costs (in interest charges) of the possible new depots and expansion at Birmingham are given in Table 20.1 together with the potential monthly throughputs.

The monthly savings of closing down the Newcastle and Exeter depots are given in Table 20.2.

Table 20.1

	Cost (£1000)	Throughput (1000 tons)
Bristol	12	30
Northampton	4	25
Birmingham (expansion)	3	20

Table 20.2

	Saving (£1000)
Newcastle	10
Exeter	5

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 251–252).

Table 20.3

Supplied to	Supplier			
	Liverpool factory	Brighton factory	Bristol depot	Northampton depot
<i>New depots</i>				
Bristol	0.6	0.4		
Northampton	0.4	0.3		
<i>Customers</i>				
C1			1.2	—
C2			0.6	0.4
C3	As given for distribution problem		0.5	—
C4			—	0.5
C5			0.3	0.6
C6			0.8	0.9

The distribution costs involving the new depots are given in Table 20.3 (in £ per ton delivered).

Which new depots should be built? Should Birmingham be expanded? Should Exeter or Newcastle be closed down? What would be the best resultant distribution pattern to minimize overall costs?

Mathematical Programming Formulation

This formulation builds on the formulation used in Chapter 19. This section includes only the new elements of the formulation.

Index Sets and Their Members

The following additional index set and its members are used in this example:

- $i, \text{depot} \in \text{EXPAND_DEPOTS}$

Parameters

Table 20.4 shows the additional parameters used in this example.

Table 20.4 Parameters

Parameter Name	Interpretation
<i>open_cost[depot]</i>	Monthly costs for opening each depot (in £)
<i>close_savings[depot]</i>	Monthly savings for closing each depot (in £)
<i>expand_throughput[depot]</i>	Potential monthly throughput per depot (in tons)
<i>expand_cost[depot]</i>	Monthly costs for expanding each depot (in £)
<i>max_num_depots</i>	Maximum number of depots allowed to be open

Variables

Table 20.5 shows the additional variables used in this example.

Table 20.5 Variables

Variable Name	Interpretation
IsOpen[depot]	1 if depot is open; 0 otherwise
Expand[depot]	1 if depot is expanded; 0 otherwise
FixedCost	Fixed cost that depends only on the binary variables
VariableCost	Variable cost that depends only on the continuous variables

Objective

The objective is to minimize the following function:

$$\text{TotalCost} = \text{FixedCost} + \text{VariableCost}$$

where

$$\begin{aligned} \text{FixedCost} = & \sum_{\text{depot} \in \text{DEPOTS}} (\text{open_cost}[\text{depot}] \cdot \text{IsOpen}[\text{depot}] - \text{close_savings}[\text{depot}] \cdot (1 - \text{IsOpen}[\text{depot}])) \\ & + \sum_{\text{depot} \in \text{EXPAND_DEPOTS}} \text{expand_cost}[\text{depot}] \cdot \text{Expand}[\text{depot}] \end{aligned}$$

and

$$\text{VariableCost} = \sum_{(i,j) \in \text{ARCS}} \text{cost}[i,j] \cdot \text{Flow}[i,j]$$

Constraints

The following additional and modified constraints are used in this example:

- bounds on variables

- $\sum_{i \in \text{DEPOTS}} \text{IsOpen}[i] \leq \text{max_num_depots}$
- for $i \in \text{DEPOTS}$,

$$\sum_{(i,j) \in \text{ARCS}} \text{Flow}[i,j] \leq \text{throughput}[i] \cdot \text{IsOpen}[i] + (\text{if } i \in \text{EXPAND_DEPOTS, then } \text{expand_throughput}[i] \cdot \text{Expand}[i])$$
- for $i \in \text{EXPAND_DEPOTS}$,

$$\text{Expand}[i] \leq \text{IsOpen}[i]$$

Input Data

The following additional and modified data sets and macro variable contain the input data that are used in this example:

```
data arc_data;
  input j $12. i $12. cost;
  datalines;
Newcastle  Liverpool  0.5
Birmingham  Liverpool  0.5
Birmingham  Brighton  0.3
London      Liverpool  1.0
London      Brighton  0.5
Exeter      Liverpool  0.2
Exeter      Brighton  0.2
C1          Liverpool  1.0
C1          Brighton  2.0
C1          Birmingham 1.0
C2          Newcastle 1.5
C2          Birmingham 0.5
C2          London    1.5
C3          Liverpool  1.5
C3          Newcastle 0.5
C3          Birmingham 0.5
C3          London    2.0
C3          Exeter    0.2
C4          Liverpool  2.0
C4          Newcastle 1.5
C4          Birmingham 1.0
C4          Exeter    1.5
C5          Birmingham 0.5
C5          London    0.5
C5          Exeter    0.5
C6          Liverpool  1.0
C6          Newcastle 1.0
C6          London    1.5
C6          Exeter    1.5
Bristol     Liverpool  0.6
```

```

Bristol      Brighton      0.4
Northampton Liverpool      0.4
Northampton Brighton      0.3
C1           Bristol       1.2
C2           Bristol       0.6
C2           Northampton 0.4
C3           Bristol       0.5
C4           Northampton 0.5
C5           Bristol       0.3
C5           Northampton 0.6
C6           Bristol       0.8
C6           Northampton 0.9
;

data depot_data;
    input depot $12. throughput cost savings;
    datalines;
Newcastle    70000      0 10000
Birmingham   50000      0      .
London       100000     0      .
Exeter       40000      0   5000
Bristol      30000 12000      0
Northampton  25000   4000      0
;

data expand_depot_data;
    input depot $12. throughput cost;
    datalines;
Birmingham   20000   3000
;

%let max_num_depots = 4;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```

proc optmodel;
    set <str,str> ARCS;
    num cost {ARCS};
    read data arc_data into ARCS=[i j] cost;

    set <str> FACTORIES;
    num capacity {FACTORIES};
    read data factory_data into FACTORIES=[factory] capacity;

    set <str> DEPOTS;
    num throughput {DEPOTS};
    num open_cost {DEPOTS};
    num close_savings {DEPOTS};

```

```

read data depot_data into DEPOTS=[depot]
  throughput open_cost=cost close_savings=savings;

set <str> EXPAND_DEPOTS;
num expand_throughput {EXPAND_DEPOTS};
num expand_cost {EXPAND_DEPOTS};
read data expand_depot_data into EXPAND_DEPOTS=[depot]
  expand_throughput=throughput expand_cost=cost;

set <str> CUSTOMERS;
num demand {CUSTOMERS};
read data customer_data into CUSTOMERS=[customer] demand;

```

The following statements are the same as in [Chapter 19](#):

```

set NODES = FACTORIES union DEPOTS union CUSTOMERS;
num supply {NODES} init 0;
for {i in FACTORIES} supply[i] = capacity[i];
for {i in CUSTOMERS} supply[i] = -demand[i];

var Flow {ARCS} >= 0;

con Flow_balance_con {i in NODES}:
  sum {<(i),j> in ARCS} Flow[i,j] - sum {<j,(i)> in ARCS} Flow[j,i] <= supply[i];

```

The following statements declare the additional variables and constraints:

```

var IsOpen {DEPOTS} binary;
var Expand {EXPAND_DEPOTS} binary;

con Max_num_depots_con:
  sum {i in DEPOTS} IsOpen[i] <= &max_num_depots;

con Depot_con {i in DEPOTS}:
  sum {<(i),j> in ARCS} Flow[i,j]
<= throughput[i] * IsOpen[i]
+ (if i in EXPAND_DEPOTS then expand_throughput[i] * Expand[i]);

con Expand_con {i in EXPAND_DEPOTS}:
  Expand[i] <= IsOpen[i];

```

The following statements fix the IsOpen variable to 1 for depots that are not eligible to be closed (indicated by a missing value for *close_savings* in the input data):

```

for {i in DEPOTS: close_savings[i] = .} do;
  close_savings[i] = 0;
  fix IsOpen[i] = 1;
end;

```

The following statements declare FixedCost and VariableCost as implicit variables and TotalCost as the objective:

```

impvar FixedCost =
    sum {depot in DEPOTS}
        (open_cost[depot] * IsOpen[depot] - close_savings[depot] * (1 - IsOpen[depot]))
    + sum {depot in EXPAND_DEPOTS} expand_cost[depot] * Expand[depot];
impvar VariableCost = sum {<i,j> in ARCS} cost[i,j] * Flow[i,j];
min TotalCost = FixedCost + VariableCost;

```

The following statements call the mixed integer linear programming solver and then print the various parts of the objective, the positive variables in the resulting optimal solution, and the additional decision variables:

```

solve;
print FixedCost VariableCost TotalCost;
print {<i,j> in ARCS: Flow[i,j].sol > 0} Flow;
print IsOpen Expand;
quit;

```

Figure 20.1 shows the output from the mixed integer linear programming solver.

Figure 20.1 Output from Mixed Integer Linear Programming Solver

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	49
Bounded Above	0
Bounded Below	42
Bounded Below and Above	5
Free	0
Fixed	2
Binary	5
Integer	2
Number of Constraints	22
Linear LE (<=)	22
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	125

Figure 20.1 continued

Solution Summary		
Solver	MILP	
Objective Function	TotalCost	
Solution Status	Optimal	
Objective Value	174000	
Iterations	22	
Best Bound	.	
Nodes	1	
Relative Gap	0	
Absolute Gap	0	
Primal Infeasibility	0	
Bound Infeasibility	0	
Integer Infeasibility	0	
Fixed Cost	Variable Cost	Total Cost
-3000	177000	174000
[1]	[2]	Flow
Birmingham	C2	10000
Birmingham	C4	10000
Birmingham	C5	50000
Brighton	Birmingham	70000
Brighton	Exeter	40000
Brighton	London	10000
Brighton	Northampton	25000
Exeter	C3	40000
Liverpool	C1	50000
Liverpool	C6	20000
London	C5	10000
Northampton	C4	25000
[1]	Is Open	Expand
Birmingham	1	1
Bristol	0	
Exeter	1	
London	1	
Newcastle	0	
Northampton	1	

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (fixed costs)
- sets of tuples
- reading multiple data sets
- set operator UNION
- INIT option
- implicit slice
- IF-THEN expression
- modeling if-then constraints using binary variables
- using a colon (:) to select members of a set
- FIX statement
- IMPVAR statement

Chapter 21

Agricultural Pricing: What Prices to Charge for Dairy Products

Contents

Problem Statement	241
Mathematical Programming Formulation	242
Input Data	244
PROC OPTMODEL Statements and Output	244
Features Demonstrated	249

Problem Statement

The government of a country wants to decide what prices should be charged for its dairy products, milk, butter and cheese.¹ All these products arise directly or indirectly from the country's raw milk production. This raw milk is usefully divided into the two components of fat and dry matter. After subtracting the quantities of fat and dry matter which are used for making products for export or consumption on the farms there is a total yearly availability of 600,000 tons of fat and 750,000 tons of dry matter. This is all available for producing milk, butter and two kinds of cheese for domestic consumption.

The percentage compositions of the products are given in [Table 21.1](#).

For the previous year the domestic consumption and prices for the products are given in [Table 21.2](#).

Table 21.1

	Fat	Dry matter	Water
Milk	4	9	87
Butter	80	2	18
Cheese 1	35	30	35
Cheese 2	25	40	35

Table 21.2

	Milk	Butter	Cheese 1	Cheese 2
Domestic consumption (1000 tons)	4820	320	210	70
Price (£/ton)	297	720	1050	815

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 252–253).

Price elasticities of demand, relating consumer demand to the prices of each product, have been calculated on the basis of past statistics. The price elasticity E of a product is defined by

$$E = \frac{\text{percentage decrease in demand}}{\text{percentage increase in price}}.$$

For the two makes of cheese there will be some degree of substitution in consumer demand depending on relative prices. This is measured by cross-elasticity of demand with respect to price. The cross-elasticity E_{AB} from a product A to a product B is defined by

$$E_{AB} = \frac{\text{percentage increase in demand for A}}{\text{percentage increase in price of B}}.$$

The elasticities and cross-elasticities are given in [Table 21.3](#).

The objective is to determine what prices and resultant demand will maximize total revenue.

It is, however, politically unacceptable to allow a certain price index to rise. As a result of the way this index is calculated this limitation simply demands that the new prices must be such that the total cost of last year's consumption would not be increased. A particularly important additional requirement is to quantify the economic cost of this political limitation.

Table 21.3

Milk	Butter	Cheese 1	Cheese 2	Cheese 1 to Cheese 2	Cheese 2 to Cheese 1
				Cheese 2	Cheese 1
0.4	2.7	1.1	0.4	0.1	0.4

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{raw} \in \text{RAWS}$
- $\text{product}, i, j \in \text{PRODUCTS}$

Parameters

Table 21.4 shows the parameters used in this example.

Table 21.4 Parameters

Parameter Name	Interpretation
<i>supply[raw]</i>	Supply of each raw material (in tons)
<i>prev_demand[product]</i>	Demand for each product in previous year (in tons)
<i>prev_price[product]</i>	Price of each product in previous year (in £/ton)
<i>percent[product,raw]</i>	Percentage composition for each product and raw
<i>elasticity[i,j]</i>	percentage increase in demand for product <i>i</i> percentage increase in price of product <i>j</i>

Variables

Table 21.5 shows the variables used in this example.

Table 21.5 Variables

Variable Name	Interpretation
Price[product]	Price of each product (in £/ton)
Demand[product]	Demand for each product (in tons)

Objective

The objective is to maximize the following (bilinear) quadratic function:

$$\text{TotalRevenue} = \sum_{\text{product} \in \text{PRODUCTS}} \text{Price}[\text{product}] \cdot \text{Demand}[\text{product}]$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $i \in \text{PRODUCTS}$,

$$\frac{\text{Demand}[i] - \text{prev_demand}[i]}{\text{prev_demand}[i]} = \sum_{j \in \text{PRODUCTS}} \text{elasticity}[i,j] \cdot \frac{\text{Price}[j] - \text{prev_price}[j]}{\text{prev_price}[j]}$$

- $\text{raw} \in \text{RAWS}$ such that *supply[raw]* is not missing,

$$\sum_{\text{product} \in \text{PRODUCTS}} \frac{\text{percent}[\text{product},\text{raw}]}{100} \cdot \text{Demand}[\text{product}] \leq \text{supply}[\text{raw}]$$

$$\begin{aligned}
 & \bullet \sum_{\text{product} \in \text{PRODUCTS}} \text{prev_demand}[\text{product}] \cdot \text{Price}[\text{product}] \\
 & \leq \sum_{\text{product} \in \text{PRODUCTS}} \text{prev_demand}[\text{product}] \cdot \text{prev_price}[\text{product}]
 \end{aligned}$$

Input Data

The following data sets contain the input data that are used in this example:

```

/* missing supply indicates unbounded */
data raw_material_data;
    input raw $10. supply;
    datalines;
Fat          600000
DryMatter    750000
Water        .
;

data product_data;
    input product $ Fat DryMatter Water prev_demand prev_price;
    datalines;
Milk         4  9 87 4820000 297
Butter       80  2 18  320000 720
Cheese1      35 30 35  210000 1050
Cheese2      25 40 35   70000 815
;

data elasticity_data;
    input i $ j $ elasticity;
    datalines;
Milk    Milk    -0.4
Butter   Butter  -2.7
Cheese1  Cheese1 -1.1
Cheese2  Cheese2 -0.4
Cheese1  Cheese2  0.1
Cheese2  Cheese1  0.4
;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```

proc optmodel;
    set <str> RAWS;
    num supply {RAWS};
    read data raw_material_data into RAWS=[raw] supply;

```

```

set <str> PRODUCTS;
num prev_demand {PRODUCTS};
num prev_price {PRODUCTS};
num percent {PRODUCTS, RAWS};
read data product_data into PRODUCTS=[product]
    {raw in RAWS} <percent[product,raw]=col(raw)> prev_demand prev_price;

num elasticity {PRODUCTS, PRODUCTS} init 0;
read data elasticity_data into [i j] elasticity;

```

The following model declaration statements correspond directly to the mathematical programming formulation described earlier:

```

var Price {PRODUCTS} >= 0;

var Demand {PRODUCTS} >= 0;

max TotalRevenue = sum {product in PRODUCTS} Price[product] * Demand[product];

con Demand_con {i in PRODUCTS}:
    (Demand[i] - prev_demand[i]) / prev_demand[i]
    = sum {j in PRODUCTS} elasticity[i,j] * (Price[j] - prev_price[j]) / prev_price[j];

con Supply_con {raw in RAWS: supply[raw] ne .}:
    sum {product in PRODUCTS} (percent[product,raw]/100) * Demand[product]
    <= supply[raw];

con Price_index_con:
    sum {product in PRODUCTS} prev_demand[product] * Price[product]
    <= sum {product in PRODUCTS} prev_demand[product] * prev_price[product];

```

Because the objective function contains a product of decision variables, PROC OPTMODEL automatically recognizes that this model is a nonlinear programming problem, and the first SOLVE statement calls the default nonlinear programming solver, which is interior point:

```

solve;
print Price Demand;
print Price_index_con.dual;

```

Figure 21.1 shows the output when you use the (default) interior point solver.

Figure 21.1 Output from Interior Point Solver

The OPTMODEL Procedure		
Problem Summary		
Objective Sense	Maximization	
Objective Function	TotalRevenue	
Objective Type	Quadratic	
Number of Variables	8	
Bounded Above	0	
Bounded Below	8	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	7	
Linear LE (<=)	3	
Linear EQ (=)	4	
Linear GE (>=)	0	
Linear Range	0	
Solution Summary		
Solver	NLP/INTERIORPOINT	
Objective Function	TotalRevenue	
Solution Status	Optimal	
Objective Value	1991585592.1	
Iterations	40	
Optimality Error	5E-9	
Infeasibility	3.473448E-10	
[1]	Price	Demand
Butter	667.29	383255
Cheese1	889.89	251695
Cheese2	1066.17	57101
Milk	303.83	4775679
	Price_	
	index_	
	con.DUAL	
	0.6161	

To invoke the active set solver (which is not the default NLP solver), you can use the SOLVER= option in the SOLVE statement:


```

solve with NLP / solver=activeset;
print Price Demand;
print Price_index_con.dual;

```

Figure 21.2 shows the output when you use the SOLVER=ACTIVESET option to invoke the active set solver.

Figure 21.2 Output from Active Set Solver

Problem Summary		
Objective Sense	Maximization	
Objective Function	TotalRevenue	
Objective Type	Quadratic	
Number of Variables	8	
Bounded Above	0	
Bounded Below	8	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	7	
Linear LE (<=)	3	
Linear EQ (=)	4	
Linear GE (>=)	0	
Linear Range	0	
Solution Summary		
Solver	NLP/ACTIVESET	
Objective Function	TotalRevenue	
Solution Status	Best Feasible	
Objective Value	1991585596.5	
Iterations	214	
Optimality Error	2.806767E-12	
Infeasibility	2.3841858E-7	
[1]	Price	Demand
Butter	667.29	383255
Cheese1	889.89	251695
Cheese2	1066.17	57101
Milk	303.83	4775678
	Price_ index_ con.DUAL	
	0.6161	

Because this model has a quadratic objective and all constraints are linear, you can also use a `WITH QP` clause to invoke the quadratic programming solver:

```
solve with QP;
print Price Demand;
print Price_index_con.dual;
quit;
```

Figure 21.3 shows the output when you use the `WITH QP` option to invoke the quadratic programming solver.

Figure 21.3 Output from Quadratic Programming Solver

Problem Summary		
Objective Sense	Maximization	
Objective Function	TotalRevenue	
Objective Type	Quadratic	
Number of Variables	8	
Bounded Above	0	
Bounded Below	8	
Bounded Below and Above	0	
Free	0	
Fixed	0	
Number of Constraints	7	
Linear LE (\leq)	3	
Linear EQ ($=$)	4	
Linear GE (\geq)	0	
Linear Range	0	
Constraint Coefficients	22	
Solution Summary		
Solver	QP	
Objective Function	TotalRevenue	
Solution Status	Optimal	
Objective Value	1991585591.9	
Iterations	12	
Primal Infeasibility	3.615938E-16	
Dual Infeasibility	1.9418945E-9	
Bound Infeasibility	0	
Duality Gap	2.812057E-13	
Complementarity	0	
[1]	Price	Demand
Butter	667.29	383255
Cheese1	889.89	251695
Cheese2	1066.17	57101
Milk	303.83	4775678

Figure 21.3 *continued*

Price_ index_ con.DUAL
0.6161

The optimal solutions and dual values for all three solvers agree.

You can also replace the Demand and Demand_con declarations with the following IMPVAR and CON statements to perform the substitutions described on page 299 of Williams:

```
impvar Demand {i in PRODUCTS} =
  prev_demand[i] * (1 +
    sum {j in PRODUCTS}
      elasticity[i,j] * (Price[j] - prev_price[j]) / prev_price[j]);

con Demand_nonnegative {i in PRODUCTS}:
  Demand[i] >= 0;
```

The resulting formulation is mathematically equivalent but yields a concave objective function, as shown in Williams.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: nonlinear programming (quadratic)
- reading multiple data sets
- INIT option
- reading sparse two-dimensional data
- using a colon (:) to select members of a set
- .dual constraint suffix
- SOLVER= option
- WITH clause
- IMPVAR statement

Chapter 22

Efficiency Analysis: How to Use Data Envelopment Analysis to Compare Efficiencies of Garages

Contents

Problem Statement	251
Mathematical Programming Formulation	252
Input Data	254
PROC OPTMODEL Statements and Output	255
Features Demonstrated	261

Problem Statement

A car manufacturer wants to evaluate the efficiencies of different garages who have received a franchise to sell its cars.¹ The method to be used is Data Envelopment Analysis (DEA). References to this technique are given in Section 3.2. Each garage has a certain number of measurable ‘inputs’. These are taken to be: *Staff*, *Showroom Space*, *Catchment Population* in different economic categories and annual *Enquiries* for different brands of car. Each garage also has a certain number of measurable ‘outputs’. These are taken to be: *Number Sold* of different brands of car and annual *Profit*. Table 22.1 gives the inputs and outputs for each of the 28 franchised garages.

A central assumption of DEA (although modified models can be built to alter this assumption) is that constant returns to scale are possible, i.e. doubling a garage’s inputs should lead to a doubling of all its outputs. A garage is deemed to be efficient if it is not possible to find a mixture of proportions of other garages whose combined inputs do not exceed those of the garage being considered, but whose outputs are equal to, or exceed, those of the garage. Should this not be possible then the garage is deemed to be inefficient and the comparator garages can be identified.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 253–255).

Table 22.1

		Inputs						Outputs		
		Staff	Show-room space (100 m ²)	Popn. cat. 1 (1000s)	Popn. cat. 2 (1000s)	Enq. Alpha model (100s)	Enq. Beta model (100s)	Alpha sales (1000s)	Beta sales (1000s)	Profit (millions)
Garage										
1	Winchester	7	8	10	12	8.5	4	2	0.6	1.5
2	Andover	6	6	20	30	9	4.5	2.3	0.7	1.6
3	Basingstoke	2	3	40	40	2	1.5	0.8	0.25	0.5
4	Poole	14	9	20	25	10	6	2.6	0.86	1.9
5	Woking	10	9	10	10	11	5	2.4	1	2
6	Newbury	24	15	15	13	25	1.9	8	2.6	4.5
7	Portsmouth	6	7	50	40	8.5	3	2.5	0.9	1.6
8	Alresford	8	7.5	5	8	9	4	2.1	0.85	2
9	Salisbury	5	5	10	10	5	2.5	2	0.65	0.9
10	Guildford	8	10	30	35	9.5	4.5	2.05	0.75	1.7
11	Alton	7	8	7	8	3	2	1.9	0.70	0.5
12	Weybridge	5	6.5	9	12	8	4.5	1.8	0.63	1.4
13	Dorchester	6	7.5	10	10	7.5	4	1.5	0.45	1.45
14	Bridport	11	8	8	10	10	6	2.2	0.65	2.2
15	Weymouth	4	5	10	10	7.5	3.5	1.8	0.62	1.6
16	Portland	3	3.5	3	20	2	1.5	0.9	0.35	0.5
17	Chichester	5	5.5	8	10	7	3.5	1.2	0.45	1.3
18	Petersfield	21	12	6	6	15	8	6	0.25	2.9
19	Petworth	6	5.5	2	2	8	5	1.5	0.55	1.55
20	Midhurst	3	3.6	3	3	2.5	1.5	0.8	0.20	0.45
21	Reading	30	29	120	80	35	20	7	2.5	8
22	Southampton	25	16	110	80	27	12	6.5	3.5	5.4
23	Bournemouth	19	10	90	22	25	13	5.5	3.1	4.5
24	Henley	7	6	5	7	8.5	4.5	1.2	0.48	2
25	Maidenhead	12	8	7	10	12	7	4.5	2	2.3
26	Fareham	4	6	1	1	7.5	3.5	1.1	0.48	1.7
27	Romsey	2	2.5	1	1	2.5	1	0.4	0.1	0.55
28	Ringwood	2	3.5	2	2	1.9	1.2	0.3	0.09	0.4

A linear programming model can be built to identify efficient and inefficient garages and their comparators.

Mathematical Programming Formulation

The formulation described here applies to each garage k . The PROC OPTMODEL statements call the linear programming solver in a loop, with one solve per garage.

Index Sets and Their Members

The following index sets and their members are used in this example:

- $i \in \text{INPUTS}$
- $i \in \text{OUTPUTS}$
- $j, k \in \text{GARAGES}$
- $j, g_1 \in \text{INEFFICIENT_GARAGES}$
- $j, g_2 \in \text{EFFICIENT_GARAGES}$

Parameters

Table 22.2 shows the parameters used in this example.

Table 22.2 Parameters

Parameter Name	Interpretation
<i>garage_name[garage]</i>	Name of garage
<i>input[i,j]</i>	Input coefficient of input i for garage j
<i>output[i,j]</i>	Output coefficient of output i for garage j
k	Dummy index for looping over garages
<i>efficiency_number[k]</i>	Efficiency number of garage k
<i>weight_sol[k,j]</i>	Weight of garage j when maximizing inefficiency of garage k

Variables

Table 22.3 shows the variables used in this example.

Table 22.3 Variables

Variable Name	Interpretation
Weight[j]	Weight of garage j
Inefficiency	Inefficiency of current garage

Objective

The objective is to maximize Inefficiency.

Constraints

The following constraints are used in this example:

- bounds on variables
- for $i \in \text{INPUTS}$,

$$\sum_{j \in \text{GARAGES}} \text{input}[i,j] \cdot \text{Weight}[j] \leq \text{input}[i,k]$$

- for $i \in \text{OUTPUTS}$,

$$\sum_{j \in \text{GARAGES}} \text{output}[i,j] \cdot \text{Weight}[j] \geq \text{output}[i,k] \cdot \text{Inefficiency}$$

Input Data

The following data sets contain the input data that are used in this example:

```
data inputs;
    input input $9.;
    datalines;
staff
showroom
pop1
pop2
alpha_enq
beta_enq
;

data outputs;
    input output $11.;
    datalines;
alpha_sales
beta_sales
profit
;

data garage_data;
    input garage_name $12. staff showroom pop1 pop2 alpha_enq beta_enq
        alpha_sales beta_sales profit;
    datalines;
Winchester 7 8 10 12 8.5 4 2 0.6 1.5
Andover 6 6 20 30 9 4.5 2.3 0.7 1.6
Basingstoke 2 3 40 40 2 1.5 0.8 0.25 0.5
Poole 14 9 20 25 10 6 2.6 0.86 1.9
Woking 10 9 10 10 11 5 2.4 1 2
Newbury 24 15 15 13 25 19 8 2.6 4.5
```


Portsmouth	6	7	50	40	8.5	3	2.5	0.9	1.6
Alresford	8	7.5	5	8	9	4	2.1	0.85	2
Salisbury	5	5	10	10	5	2.5	2	0.65	0.9
Guildford	8	10	30	35	9.5	4.5	2.05	0.75	1.7
Alton	7	8	7	8	3	2	1.9	0.7	0.5
Weybridge	5	6.5	9	12	8	4.5	1.8	0.63	1.4
Dorchester	6	7.5	10	10	7.5	4	1.5	0.45	1.45
Bridport	11	8	8	10	10	6	2.2	0.65	2.2
Weymouth	4	5	10	10	7.5	3.5	1.8	0.62	1.6
Portland	3	3.5	3	2	2	1.5	0.9	0.35	0.5
Chichester	5	5.5	8	10	7	3.5	1.2	0.45	1.3
Petersfield	21	12	6	8	15	8	6	0.25	2.9
Petworth	6	5.5	2	2	8	5	1.5	0.55	1.55
Midhurst	3	3.6	3	3	2.5	1.5	0.8	0.2	0.45
Reading	30	29	120	80	35	20	7	2.5	8
Southampton	25	16	110	80	27	12	6.5	3.5	5.4
Bournemouth	19	10	90	12	25	13	5.5	3.1	4.5
Henley	7	6	5	7	8.5	4.5	1.2	0.48	2
Maidenhead	12	8	7	10	12	7	4.5	2	2.3
Fareham	4	6	1	1	7.5	3.5	1.1	0.48	1.7
Romsey	2	2.5	1	1	2.5	1	0.4	0.1	0.55
Ringwood	2	3.5	2	2	1.9	1.2	0.3	0.09	0.4

;

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```
proc optmodel;
  set <str> INPUTS;
  read data inputs into INPUTS=[input];

  set <str> OUTPUTS;
  read data outputs into OUTPUTS=[output];

  set <num> GARAGES;
  str garage_name {GARAGES};
  num input {INPUTS, GARAGES};
  num output {OUTPUTS, GARAGES};
  read data garage_data into GARAGES=[_N_] garage_name
    {i in INPUTS} <input[i,_N_]=col(i)>
    {i in OUTPUTS} <output[i,_N_]=col(i)>;

  num k;
  num efficiency_number {GARAGES};
  num weight_sol {GARAGES, GARAGES};
```

The following statements correspond directly to the mathematical programming formulation described earlier:

```

var Weight {GARAGES} >= 0;
var Inefficiency >= 0;

max Objective = Inefficiency;

con Input_con {i in INPUTS}:
    sum {j in GARAGES} input[i,j] * Weight[j] <= input[i,k];

con Output_con {i in OUTPUTS}:
    sum {j in GARAGES} output[i,j] * Weight[j] >= output[i,k] * Inefficiency;

```

The following statements loop over all garages, call the linear programming solver once per garage, and store the results in the parameters *efficiency_number* and *weight_sol*:

```

do k = GARAGES;
    solve;
    efficiency_number[k] = 1 / Inefficiency.sol;
    for {j in GARAGES}
        weight_sol[k,j] = (if Weight[j].sol > 1e-6 then Weight[j].sol else .);
    end;

```

Note that the DO loop contains no declaration statements. As the value of *k* changes, the SOLVE statement automatically updates the constraints to use the values of *input[i,k]* and *output[i,k]*. The approach shown here is much more efficient than the following alternatives:

- Calling PROC OPTMODEL once per garage
- Enlarging the set of decision variables by including an additional index, resulting in variables *Weight[k,j]* and *Inefficiency[k]*. Within the DO loop, you must then fix most of the variables to 0 and rely on the presolver to remove them, but that approach uses much more memory and computational time.

After the DO loop terminates, the following statements partition the garages into two sets by using a threshold on the resulting efficiency numbers:

```

set EFFICIENT_GARAGES = {j in GARAGES: efficiency_number[j] >= 1};
set INEFFICIENT_GARAGES = GARAGES diff EFFICIENT_GARAGES;

```

The following statements print the efficiency numbers, as shown in [Figure 22.1](#), and write them to the *efficiency_data* data set:

```

print garage_name efficiency_number;
create data efficiency_data from [garage] garage_name efficiency_number;

```

Figure 22.1 *efficiency_number* Parameter

The OPTMODEL Procedure		
	garage_name	efficiency_number
[1]		
1	Winchester	0.84017
2	Andover	0.91738
3	Basingstoke	1.00000
4	Poole	0.86189
5	Woking	0.86732
6	Newbury	1.00000
7	Portsmouth	1.00000
8	Alresford	1.00000
9	Salisbury	1.00000
10	Guildford	0.81417
11	Alton	1.00000
12	Weybridge	0.85435
13	Dorchester	0.83920
14	Bridport	0.97101
15	Weymouth	1.00000
16	Portland	1.00000
17	Chichester	0.82434
18	Petersfield	1.00000
19	Petworth	0.98824
20	Midhurst	0.82928
21	Reading	0.98205
22	Southampton	1.00000
23	Bournemouth	1.00000
24	Henley	1.00000
25	Maidenhead	1.00000
26	Fareham	1.00000
27	Romsey	1.00000
28	Ringwood	0.87587

The following CREATE DATA statements write the inefficient garages and the corresponding multiples of efficient garages to SAS data sets (in both dense and sparse form), as in Table 14.8 in Williams:

```

create data weight_data_dense from [inefficient_garage]=INEFFICIENT_GARAGES
  garage_name
  efficiency_number
  {efficient_garage in EFFICIENT_GARAGES}
  <col('w' || efficient_garage)=weight_sol[inefficient_garage,efficient_garage]>;
create data weight_data_sparse from
  [inefficient_garage efficient_garage]=
  {g1 in INEFFICIENT_GARAGES, g2 in EFFICIENT_GARAGES: weight_sol[g1,g2] ne .}
  weight_sol;
quit;

```

The following statements sort the efficiency_data data set by efficiency and print the results, shown in Figure 22.2:

```
proc sort data=efficiency_data;
  by descending efficiency_number;
run;
```

```
proc print;
run;
```

Figure 22.2 efficiency_data Data Set

Obs	garage	garage_name	efficiency_ number
1	3	Basingstoke	1.00000
2	6	Newbury	1.00000
3	7	Portsmouth	1.00000
4	8	Alresford	1.00000
5	9	Salisbury	1.00000
6	11	Alton	1.00000
7	15	Weymouth	1.00000
8	16	Portland	1.00000
9	18	Petersfield	1.00000
10	22	Southampton	1.00000
11	23	Bournemouth	1.00000
12	24	Henley	1.00000
13	25	Maidenhead	1.00000
14	26	Fareham	1.00000
15	27	Romsey	1.00000
16	19	Petworth	0.98824
17	21	Reading	0.98205
18	14	Bridport	0.97101
19	2	Andover	0.91738
20	28	Ringwood	0.87587
21	5	Woking	0.86732
22	4	Poole	0.86189
23	12	Weybridge	0.85435
24	1	Winchester	0.84017
25	13	Dorchester	0.83920
26	20	Midhurst	0.82928
27	17	Chichester	0.82434
28	10	Guildford	0.81417

The following statements sort the `weight_data_dense` data set by efficiency and print the results, shown in Figure 22.3:

```
proc sort data=weight_data_dense;
  by descending efficiency_number;
run;
```

```
proc print;
run;
```

Figure 22.3 weight_data_dense Data Set

Obs	inefficient_ garage	garage_ name	efficiency_ number	w3	w6	w7	w8	w9
1	19	Petworth	0.98824	.	0.066345	.	.	.
2	21	Reading	0.98205	1.26862
3	14	Bridport	0.97101	0.03278
4	2	Andover	0.91738
5	28	Ringwood	0.87587	0.00771
6	5	Woking	0.86732	.	.	.	0.95253	.
7	4	Poole	0.86189	0.32859
8	12	Weybridge	0.85435
9	1	Winchester	0.84017	.	.	0.00528	0.41627	0.40328
10	13	Dorchester	0.83920	0.13436	.	.	0.10448	.
11	20	Midhurst	0.82928	0.05957
12	17	Chichester	0.82434	0.05825	.	.	0.09682	.
13	10	Guildford	0.81417	0.42459	.	0.14961	0.62272	.

Obs	w11	w15	w16	w18	w22	w23	w24	w25	w26	w27
1	.	.	.	0.015212	.	.	.	0.03409	0.67493	.
2	.	0.54441	1.19914	.	.	.	2.86247	0.13753	.	.
3	.	.	0.46969	.	.	.	0.78310	0.19489	.	.
4	.	0.85714	0.21429	.	.
5	.	.	0.31973	.	.	.	0.14649	.	.	.
6	0.021078	.	.	.	0.009092662	.	.	0.14838	.	.
7	.	.	0.75733	.	.	.	0.43442	0.34463	.	.
8	.	0.79656	0.14524	0.01773	.
9	.	0.33333	0.09614
10	.	0.11929	0.75163	.	.	.	0.03532	.	0.47905	.
11	.	0.06651	0.47189	0.043482	.	.	.	0.00894	.	.
12	.	0.33543	0.16523	.	.	.	0.23637	.	0.15424	.
13	.	0.19180	0.16807

The weight_data_sparse data set contains the same information in sparse format, as shown in [Figure 22.4](#):

```
proc print data=weight_data_sparse;
run;
```

Figure 22.4 weight_data_sparse Data Set

Obs	inefficient_ garage	efficient_ garage	weight_ sol
1	1	7	0.00528
2	1	8	0.41627
3	1	9	0.40328
4	1	15	0.33333
5	1	16	0.09614
6	2	15	0.85714
7	2	25	0.21429
8	4	3	0.32859
9	4	16	0.75733
10	4	24	0.43442
11	4	25	0.34463
12	5	8	0.95253
13	5	11	0.02108
14	5	22	0.00909
15	5	25	0.14838
16	10	3	0.42459
17	10	7	0.14961
18	10	8	0.62272
19	10	15	0.19180
20	10	16	0.16807
21	12	15	0.79656
22	12	25	0.14524
23	12	26	0.01773
24	13	3	0.13436
25	13	8	0.10448
26	13	15	0.11929
27	13	16	0.75163
28	13	24	0.03532
29	13	26	0.47905
30	14	3	0.03278
31	14	16	0.46969
32	14	24	0.78310
33	14	25	0.19489
34	17	3	0.05825
35	17	8	0.09682
36	17	15	0.33543
37	17	16	0.16523
38	17	24	0.23637
39	17	26	0.15424
40	19	6	0.06635
41	19	18	0.01521
42	19	25	0.03409
43	19	26	0.67493
44	20	9	0.05957
45	20	15	0.06651
46	20	16	0.47189
47	20	18	0.04348
48	20	25	0.00894
49	21	3	1.26862
50	21	15	0.54441
51	21	16	1.19914
52	21	24	2.86247
53	21	25	0.13753
54	28	3	0.00771
55	28	16	0.31973
56	28	24	0.14649

Features Demonstrated

The following features are demonstrated in this example:

- problem type: linear programming
- numeric and string index sets
- reading and writing multiple data sets
- reading dense two-dimensional data
- calling a solver in a DO loop
- `.sol` variable suffix
- storing a solution in a numeric parameter
- using a colon (:) to select members of a set
- set operator DIFF
- writing dense two-dimensional data
- writing sparse two-dimensional data

Chapter 23

Milk Collection: How to Route and Assign Milk Collection Lorries to Farms

Contents

Problem Statement	263
Mathematical Programming Formulation	265
Input Data	267
PROC OPTMODEL Statements and Output	268
Features Demonstrated	279

Problem Statement

A small milk processing company is committed to collecting milk from 20 farms and taking it back to the depot for processing.¹ The company has one tanker lorry with a capacity for carrying 80,000 litres of milk. Eleven of the farms are small and need a collection only every other day. The other nine farms need a collection every day. The positions of the farms in relation to the depot (numbered 1) are given in [Table 23.1](#) together with their collection requirements.

Find the optimal route for the tanker lorry on each day, bearing in mind that it has to (i) visit all the ‘every day’ farms, (ii) visit some of the ‘every other day’ farms, and (iii) work within its capacity. On alternate days it must again visit the ‘every day’ farms but also visit the ‘every other day’ farms not visited on the previous day.

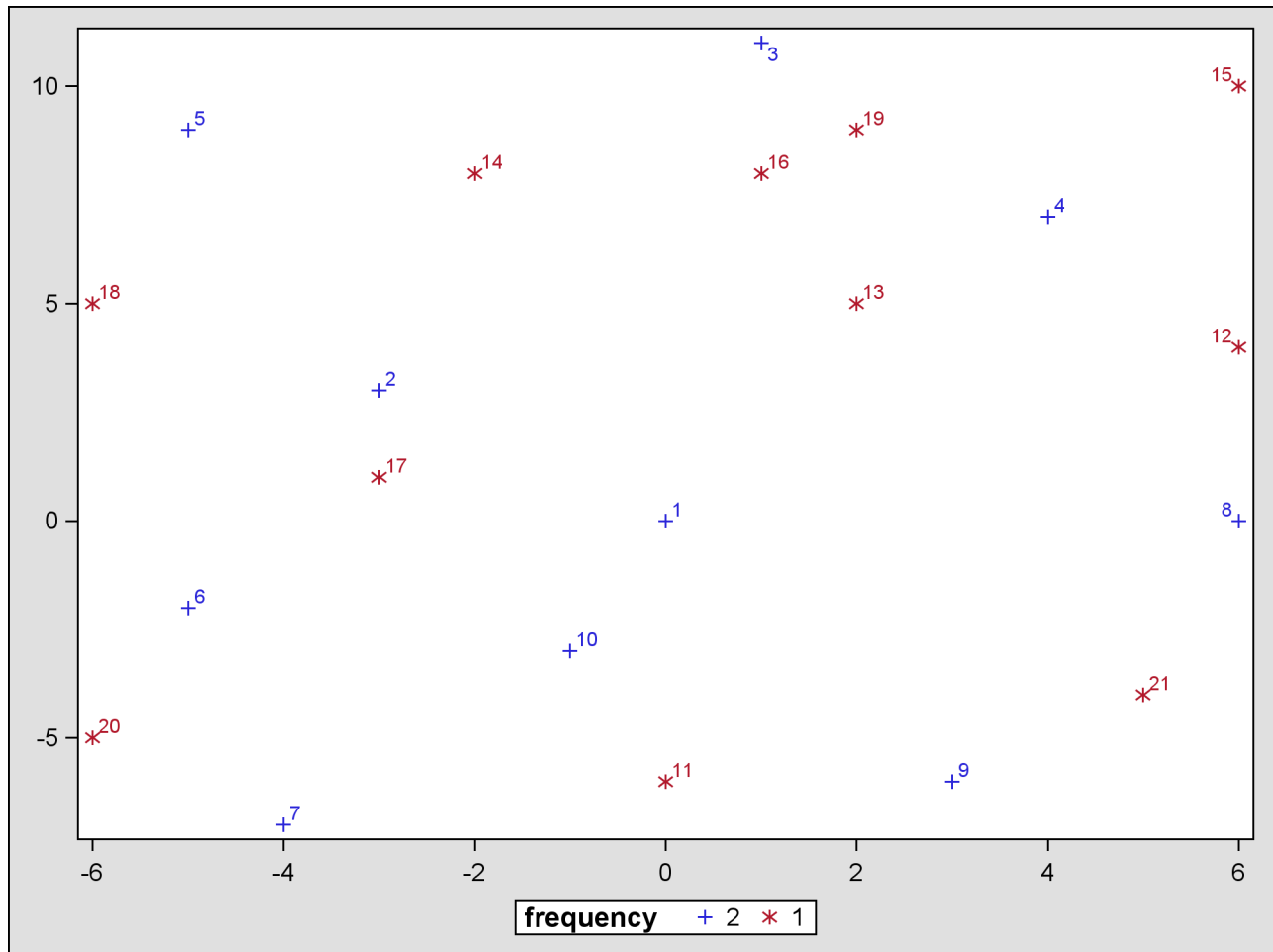
For convenience a map of the area considered is given in [Figure 23.1](#).

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 255–256).

Table 23.1

Farm	Position 10 miles:		Collection frequency	Collection requirement (1000 litres)
	East	North		
1 (Depot)	0	0	—	—
2	−3	3	Every day	5
3	1	11	Every day	4
4	4	7	Every day	3
5	−5	9	Every day	6
6	−5	−2	Every day	7
7	−4	−7	Every day	3
8	6	0	Every day	4
9	3	−6	Every day	6
10	−1	−3	Every day	5
11	0	−6	Every other day	4
12	6	4	Every other day	7
13	2	5	Every other day	3
14	−2	8	Every other day	4
15	6	10	Every other day	5
16	1	8	Every other day	6
17	−3	1	Every other day	8
18	−6	5	Every other day	5
19	2	9	Every other day	7
20	−6	−5	Every other day	6
21	5	−4	Every other day	6

Figure 23.1



Mathematical Programming Formulation

The problem is formulated as a periodic vehicle routing problem with a node for each farm. Williams uses the fact that the number of days in the planning horizon is two to reduce the number of decision variables and constraints. The formulation shown here generalizes to any number of days and arbitrary frequencies (not just 1 or 2).

Index Sets and Their Members

The following index sets and their members are used in this example:

- $i, j \in \text{NODES}$
- $(i, j) \in \text{EDGES}$

- $d \in \text{DAYS}$
- $i, j \in \text{NODES_TEMP}$
- $(i, j) \in \text{EDGES_SOL}[\text{iter}, d]$
- $c_i, c_j \in \text{COMPONENT_IDS}$
- $k \in \text{COMPONENT}[c_i]$
- $k \in \text{SUBTOUR}[s]$

Parameters

Table 23.2 shows the parameters used in this example.

Table 23.2 Parameters

Parameter Name	Interpretation
<i>east[i]</i>	East coordinate of node <i>i</i>
<i>north[i]</i>	North coordinate of node <i>i</i>
<i>frequency[i]</i>	Number of times node <i>i</i> must be visited (1 for every other day, 2 for every day)
<i>requirement[i]</i>	Collection requirement of node <i>i</i> (in 1,000 litres)
<i>distance[i,j]</i>	Distance between nodes <i>i</i> and <i>j</i> (in miles)
<i>distance_scale</i>	Distance scale
<i>num_days</i>	Number of days in planning horizon
<i>capacity</i>	Capacity of the lorry (in 1,000 litres)
<i>depot</i>	Node number of depot
<i>num_subtours</i>	Number of subtours in current formulation
<i>iter</i>	Iteration number for each round of subtour elimination constraints
<i>num_components[d]</i>	Number of connected components in solution for day <i>d</i>
<i>component_id[i]</i>	Connected component containing node <i>i</i>
<i>ci,cj</i>	Dummy indices for members of COMPONENT_IDS

Variables

Table 23.3 shows the variables used in this example.

Table 23.3 Variables

Variable Name	Interpretation
UseNode[i,d]	1 if node <i>i</i> is visited on day <i>d</i> ; 0 otherwise
UseEdge[i,j,d]	1 if edge (<i>i, j</i>) is traversed on day <i>d</i> ; 0 otherwise

Objective

The objective is to minimize the following function:

$$\text{TotalDistance} = \sum_{(i,j) \in \text{EDGES}} \sum_{d \in \text{DAYS}} \text{distance}[i,j] \cdot \text{UseEdge}[i,j,d]$$

Constraints

The following constraints are used in this example:

- bounds on variables

- for $d \in \text{DAYS}$,

$$\sum_{i \in \text{NODES}} \text{requirement}[i] \cdot \text{UseNode}[i,d] \leq \text{capacity}$$

- for $i \in \text{NODES}$,

$$\sum_{d \in \text{DAYS}} \text{UseNode}[i,d] = \text{frequency}[i]$$

- for $k \in \text{NODES}$ and $d \in \text{DAYS}$,

$$\sum_{(i,j) \in \text{EDGES}: k \in \{i,j\}} \text{UseEdge}[i,j,d] = 2 \cdot \text{UseNode}[k,d]$$

- for $d \in \text{DAYS} \setminus \{1\}$,

$$\sum_{i \in \text{NODES}} \text{UseNode}[i,d] \leq \sum_{i \in \text{NODES}} \text{UseNode}[i,d-1]$$

- for $s \in \{1, \dots, \text{num_subtours}\}$ and $k \in \text{SUBTOUR}[s]$ and $d \in \text{DAYS}$,

$$\sum_{\substack{i \in \text{NODES} \setminus \text{SUBTOUR}[s], \\ j \in \text{SUBTOUR}[s]: \\ (i,j) \in \text{EDGES}}} \text{UseEdge}[i,j,d] + \sum_{\substack{i \in \text{SUBTOUR}[s], \\ j \in \text{NODES} \setminus \text{SUBTOUR}[s]: \\ (i,j) \in \text{EDGES}}} \text{UseEdge}[i,j,d] \geq 2 \cdot \text{UseNode}[k,d]$$

Input Data

The following data set and macro variables contain the input data that are used in this example:

```

data farm_data;
    farm = _N_;
    input east north frequency requirement;
    datalines;
0 0 2 0
-3 3 2 5
1 11 2 4
4 7 2 3
-5 9 2 6
-5 -2 2 7
-4 -7 2 3
6 0 2 4
3 -6 2 6
-1 -3 2 5
0 -6 1 4
6 4 1 7
2 5 1 3
-2 8 1 4
6 10 1 5
1 8 1 6
-3 1 1 8
-6 5 1 5
2 9 1 7
-6 -5 1 6
5 -4 1 6
;

%let distance_scale = 10;
%let num_days = 2;
%let capacity = 80;
%let depot = 1;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```

proc optmodel;
    set NODES;
    num east {NODES};
    num north {NODES};
    num frequency {NODES};
    num requirement {NODES};
    read data farm_data into NODES=[_N_] east north frequency requirement;

    set EDGES = {i in NODES, j in NODES: i < j};
    num distance {<i,j> in EDGES} =
        &distance_scale * sqrt((east[i]-east[j])^2+(north[i]-north[j])^2);

    set DAYS = 1..&num_days;

```

The following model declaration statements correspond directly to the mathematical programming formulation described earlier:

```

var UseNode {NODES, DAYS} binary;
var UseEdge {EDGES, DAYS} binary;

min TotalDistance = sum {<i,j> in EDGES, d in DAYS} distance[i,j] * UseEdge[i,j,d];

con Capacity_con {d in DAYS}:
    sum {i in NODES} requirement[i] * UseNode[i,d] <= &capacity;

con Frequency_con {i in NODES}:
    sum {d in DAYS} UseNode[i,d] = frequency[i];

con Two_match {k in NODES, d in DAYS}:
    sum {<i,j> in EDGES: k in {i,j}} UseEdge[i,j,d] = 2 * UseNode[k,d];

```

The following statements declare optional symmetry-breaking constraints to reduce the number of essentially identical branch-and-bound nodes that are explored by the mixed integer linear programming solver:

```

/* several alternatives for symmetry-breaking constraints */
* con Symmetry {d in DAYS diff {1}}:
    sum {<i,j> in EDGES} distance[i,j] * UseEdge[i,j,d]
<= sum {<i,j> in EDGES} distance[i,j] * UseEdge[i,j,d-1];
* con Symmetry {d in DAYS diff {1}}:
    sum {i in NODES} requirement[i] * UseNode[i,d]
<= sum {i in NODES} requirement[i] * UseNode[i,d-1];
con Symmetry {d in DAYS diff {1}}:
    sum {i in NODES} UseNode[i,d]
<= sum {i in NODES} UseNode[i,d-1];

```

Williams breaks symmetry instead by fixing $\text{UseNode}[11,1] = 1$. The symmetry-breaking constraints shown here apply to the general formulation with any number of days and arbitrary frequencies.

The following statements declare the subtour elimination constraints:

```

num num_subtours init 0;
set SUBTOUR {1..num_subtours}; /* subset of nodes not containing depot node */
/* if node k in SUBTOUR[s] is used on day d,
    then must use at least two edges across partition induced by SUBTOUR[s] */
con Subtour_elimination {s in 1..num_subtours, k in SUBTOUR[s], d in DAYS}:
    sum {i in NODES diff SUBTOUR[s], j in SUBTOUR[s]: <i,j> in EDGES} UseEdge[i,j,d]
+ sum {i in SUBTOUR[s], j in NODES diff SUBTOUR[s]: <i,j> in EDGES} UseEdge[i,j,d]
>= 2 * UseNode[k,d];

```

The following statements declare the index sets and parameters that are needed to detect violated subtour elimination constraints:

```

num iter init 0;
num num_components {DAYS};
set NODES_TEMP;
set <num,num> EDGES_SOL {1..iter, DAYS};
num component_id {NODES_TEMP};

```

```

set COMPONENT_IDS;
set COMPONENT {COMPONENT_IDS};
num ci;
num cj;

```

The following DO UNTIL loop implements dynamic generation of subtour elimination constraints (“row generation”):

```

/* loop until each day's support graph is connected */
do until (and {d in DAYS} num_components[d] = 1);
  iter = iter + 1;
  solve;
  /* find connected components for each day */
  for {d in DAYS} do;
    NODES_TEMP = {i in NODES: UseNode[i,d].sol > 0.5};
    EDGES_SOL[iter,d] = {<i,j> in EDGES: UseEdge[i,j,d].sol > 0.5};
    /* initialize each node to its own component */
    COMPONENT_IDS = NODES_TEMP;
    num_components[d] = card(NODES_TEMP);
    for {i in NODES_TEMP} do;
      component_id[i] = i;
      COMPONENT[i] = {i};
    end;
    /* if i and j are in different components, merge the two components */
    for {<i,j> in EDGES_SOL[iter,d]} do;
      ci = component_id[i];
      cj = component_id[j];
      if ci ne cj then do;
        /* update smaller component */
        if card(COMPONENT[ci]) < card(COMPONENT[cj]) then do;
          for {k in COMPONENT[ci]} component_id[k] = cj;
          COMPONENT[cj] = COMPONENT[cj] union COMPONENT[ci];
          COMPONENT_IDS = COMPONENT_IDS diff {ci};
        end;
        else do;
          for {k in COMPONENT[cj]} component_id[k] = ci;
          COMPONENT[ci] = COMPONENT[ci] union COMPONENT[cj];
          COMPONENT_IDS = COMPONENT_IDS diff {cj};
        end;
      end;
    end;
    num_components[d] = card(COMPONENT_IDS);
    put num_components[d]=;
    /* create subtour from each component not containing depot node */
    for {k in COMPONENT_IDS: &depot not in COMPONENT[k]} do;
      num_subtours = num_subtours + 1;
      SUBTOUR[num_subtours] = COMPONENT[k];
      put SUBTOUR[num_subtours]=;
    end;
  end;
  print capacity_con.body capacity_con.ub;
  print num_components;
end;

```


The body of the loop calls the mixed integer linear programming solver, finds the connected components of the support graph of the resulting solution, and adds any subtours found. Note that the DO UNTIL loop contains no declaration statements. As the value of *num_subtours* changes, the SOLVE statement automatically updates the subtour elimination constraints.

After the DO UNTIL loop terminates, the following statements output the edges that appear in each iteration of subtour elimination and write the value of *iter* to a SAS macro variable named *num_iters*:

```
create data sol_data from
  [iter d i j]={it in 1..iter, d in DAYS, <i,j> in EDGES_SOL[it,d]}
  xi=east[i] yi=north[i] xj=east[j] yj=north[j];
call symput('num_iters',put(iter,best.));
quit;
```

The following SAS macro calls PROC SGPLOT to plot the solution that results from each iteration:

```
%macro showPlots;
  %do iter = 1 %to &num_iters;
    %do d = 1 %to &num_days;
      /* create annotate data set to draw subtours */
      data sganno(keep=drawspace linethickness function x1 y1 x2 y2);
        retain drawspace "datavalue" linethickness 1;
        set sol_data(rename=(xi=x1 yi=y1 xj=x2 yj=y2));
        where iter = &iter and d = &d;
        function = 'line';
      run;

      title1 "iter = &iter, day = &d";
      title2;
      proc sgplot data=farm_data sganno=sganno;
        scatter y=north x=east / group=frequency datalabel=farm;
        xaxis display=(nolabel);
        yaxis display=(nolabel);
      run;
      quit;
    %end;
  %end;
%mend showPlots;
%showPlots;
```

Figure 23.2 and Figure 23.3 show the output from the mixed integer linear programming solver for the first iteration, before any subtour elimination constraints have been generated.

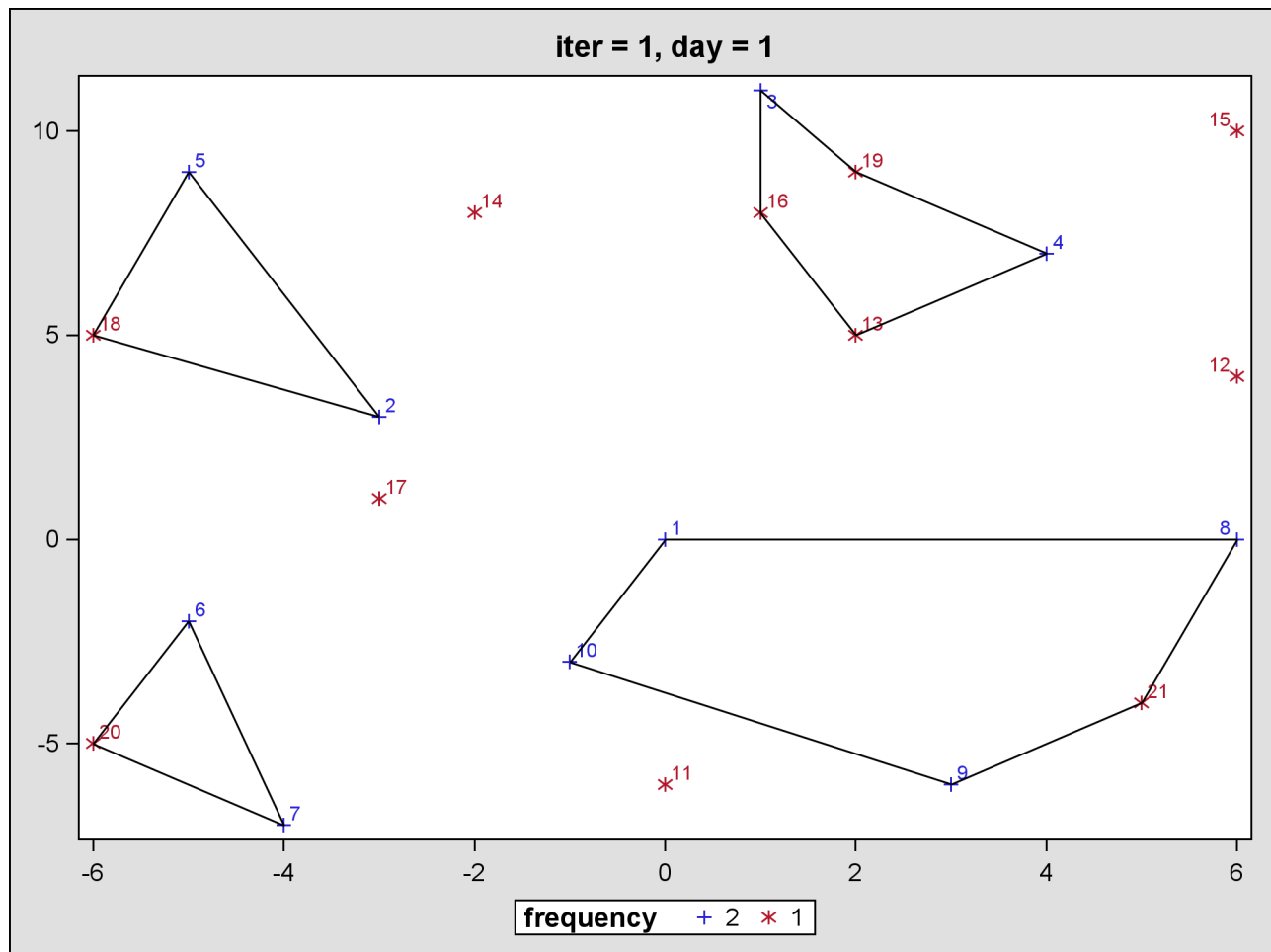
Figure 23.2 Output from Mixed Integer Linear Programming Solver, Iteration 1, Day 1

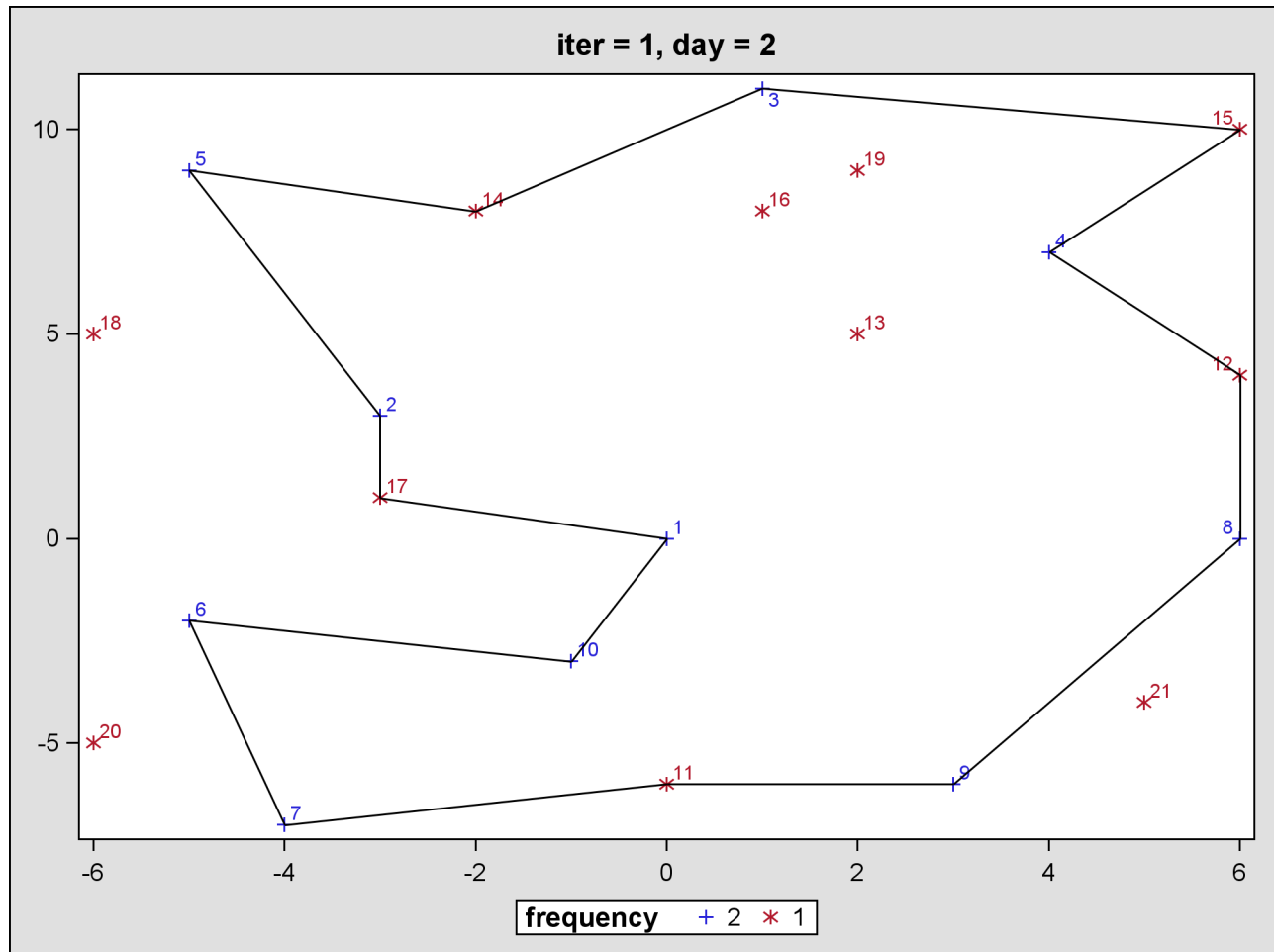
Figure 23.3 Output from Mixed Integer Linear Programming Solver, Iteration 1, Day 2

Figure 23.4 and Figure 23.5 show the output from the mixed integer linear programming solver for the second iteration, after the first round of subtour elimination constraints.

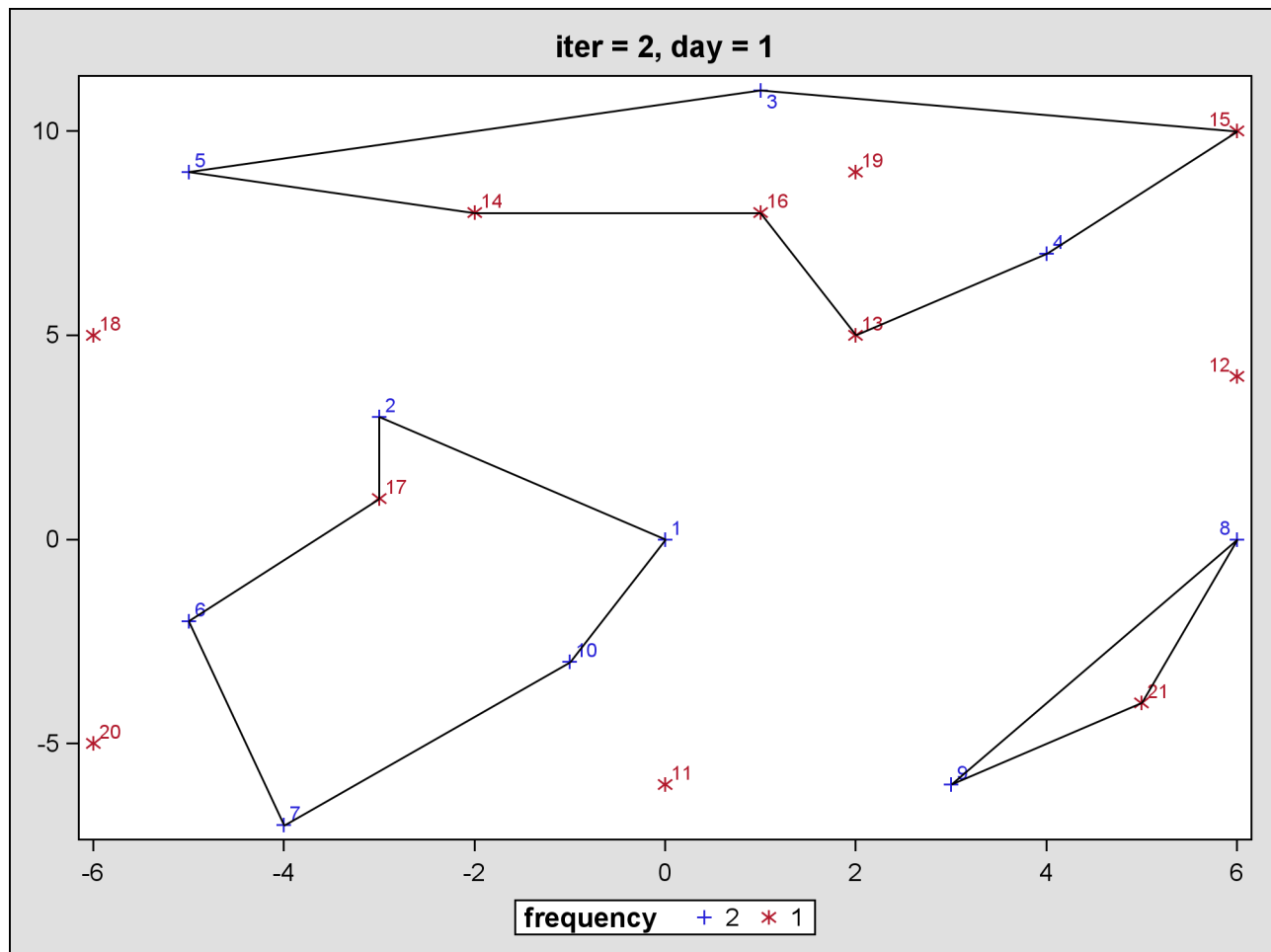
Figure 23.4 Output from Mixed Integer Linear Programming Solver, Iteration 2, Day 1

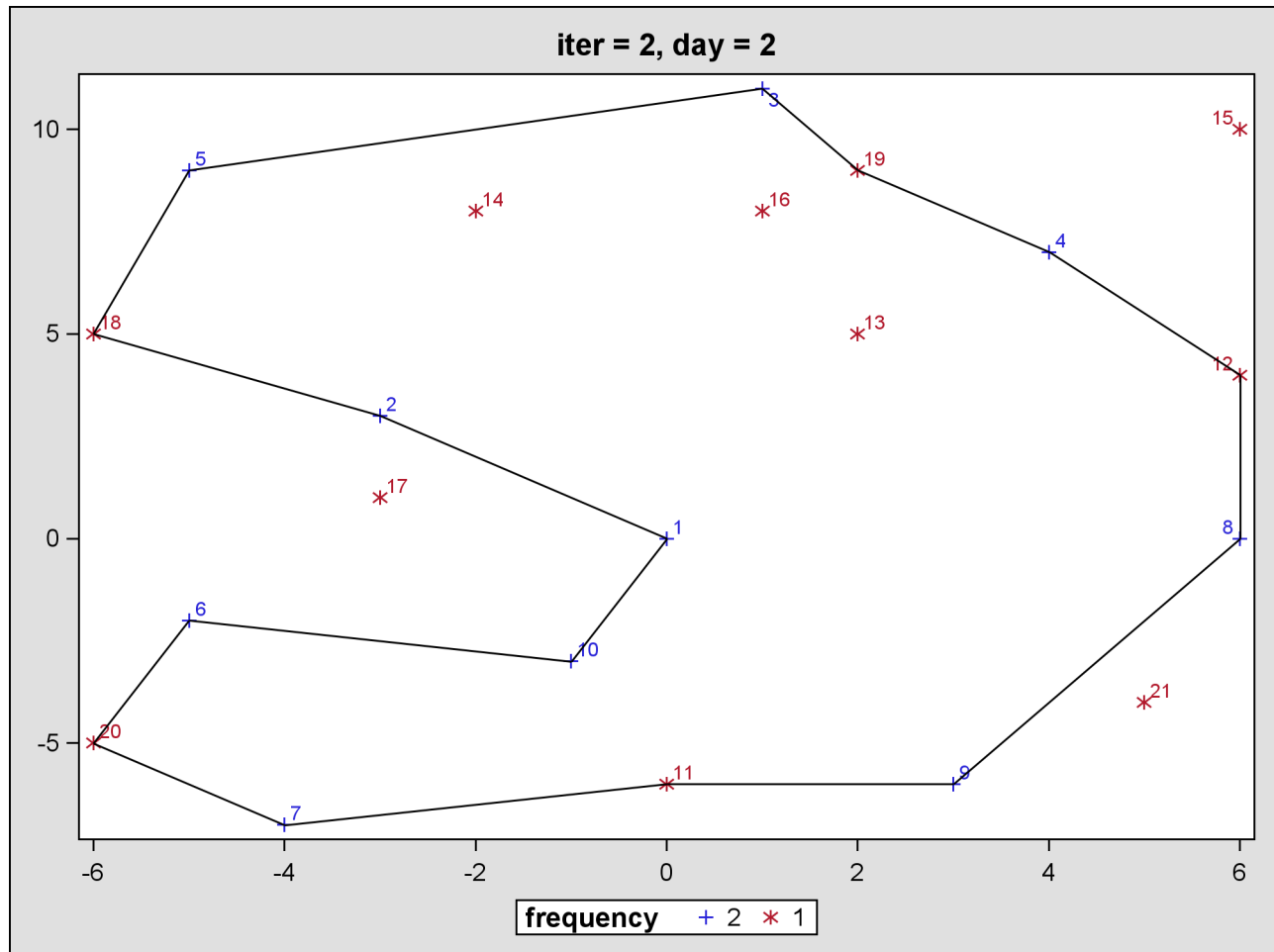
Figure 23.5 Output from Mixed Integer Linear Programming Solver, Iteration 2, Day 2

Figure 23.6 and Figure 23.7 show the output from the mixed integer linear programming solver for the third iteration, after the second round of subtour elimination constraints.

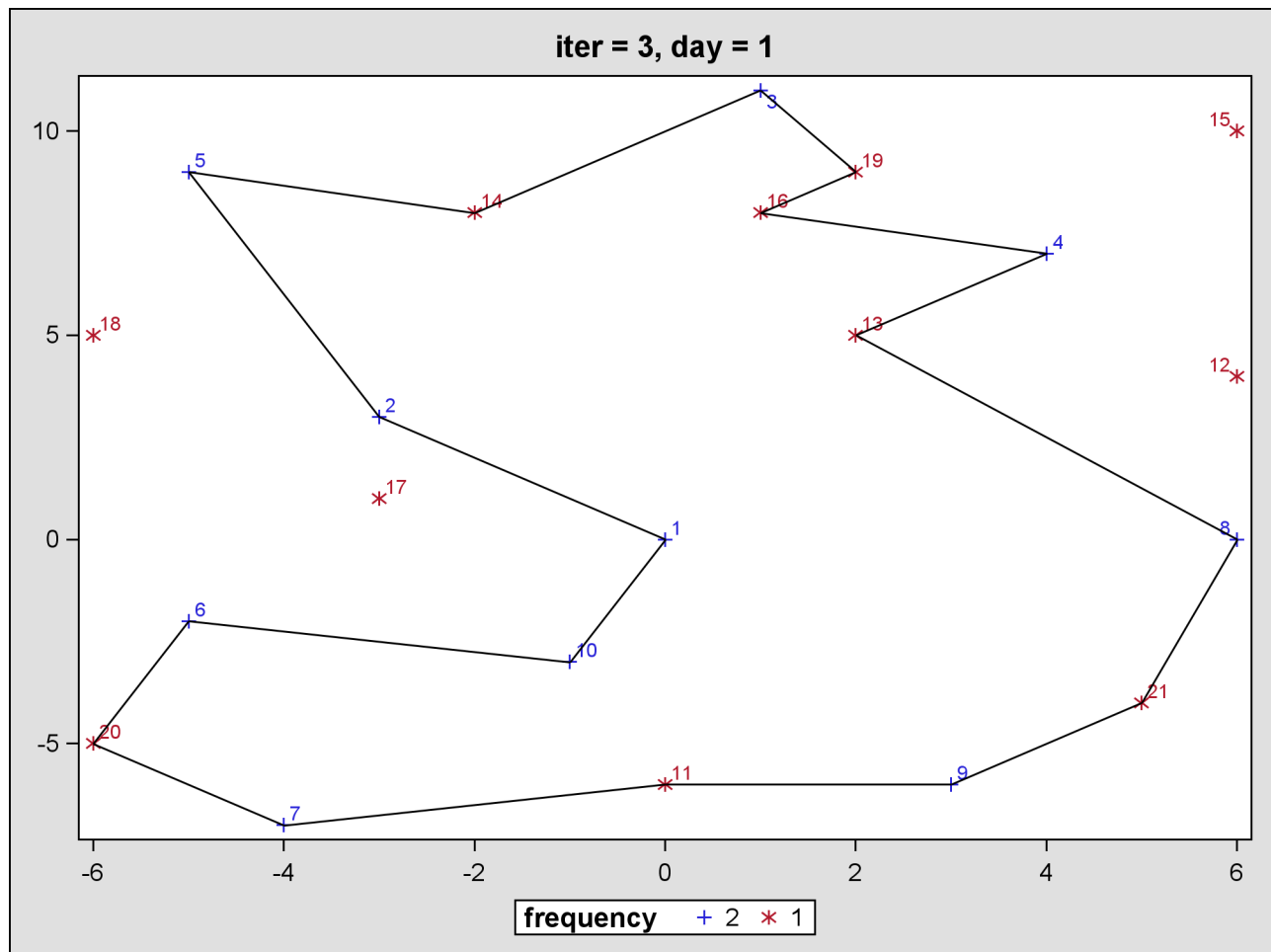
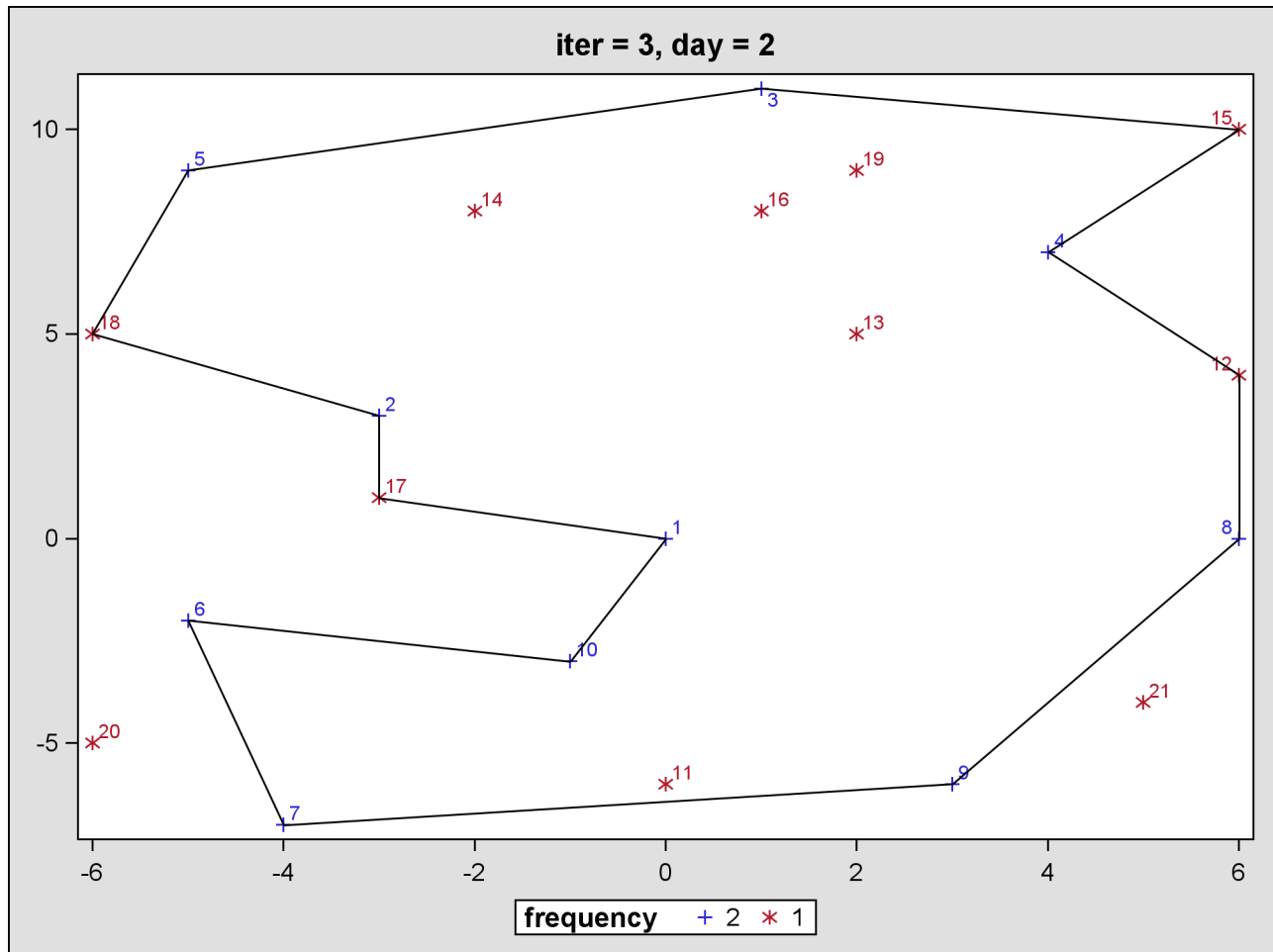
Figure 23.6 Output from Mixed Integer Linear Programming Solver, Iteration 3, Day 1

Figure 23.7 Output from Mixed Integer Linear Programming Solver, Iteration 3, Day 2

Since no more subtour elimination constraints are violated, the DO UNTIL loop terminates with an optimal solution. Figure 23.8 shows the final problem and solution summaries from the mixed integer linear programming solver.

Figure 23.8 Final Problem and Solution Summaries from Mixed Integer Linear Programming Solver

Problem Summary	
Objective Sense	Minimization
Objective Function	TotalDistance
Objective Type	Linear
Number of Variables	462
Bounded Above	0
Bounded Below	0
Bounded Below and Above	462
Free	0
Fixed	0
Binary	462
Integer	0
Number of Constraints	108
Linear LE (\leq)	3
Linear EQ ($=$)	63
Linear GE (\geq)	42
Linear Range	0
Constraint Coefficients	4192
Solution Summary	
Solver	MILP
Objective Function	TotalDistance
Solution Status	Optimal
Objective Value	1230.5623785
Iterations	163
Best Bound	.
Nodes	1
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	8.604138E-16
Bound Infeasibility	4.440892E-16
Integer Infeasibility	1.0000001E-8

The optimal objective value differs slightly from the one given in Williams, perhaps due to rounding of distances by Williams.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (vehicle routing)
- sets of tuples
- sets indexed by other sets
- set operators UNION and DIFF
- INIT option
- CARD function
- using a colon (:) to select members of a set
- AND aggregation operator
- symmetry-breaking constraints
- calling a solver in a DO UNTIL loop
- row generation
- connected components
- `.sol` variable suffix
- CALL SYMPUT
- SGPLOT procedure

Chapter 24

Yield Management: What Quantities of Airline Tickets to Sell at What Prices and What Times

Contents

Problem Statement	281
Mathematical Programming Formulation	284
Input Data	289
PROC OPTMODEL Statements and Output	291
Features Demonstrated	310

Problem Statement

An airline is selling tickets for flights to a particular destination.¹ The flight will depart in three weeks' time. It can use up to six planes each costing £50,000 to hire. Each plane has:

- 37 First Class seats
- 38 Business Class seats
- 47 Economy Class seats

Up to 10% of seats in any one category can be transferred to an adjacent category.

It wishes to decide a price for each of these seats. There will be further opportunities to update these prices after one week and two weeks. Once a customer has purchased a ticket there is no cancellation option.

For administrative simplicity three price level options are possible in each class (one of which must be chosen). The same option need not be chosen for each class. These are given in [Table 24.1](#) for the current period (period 1) and two future periods.

¹Reproduced with permission of John Wiley & Sons Ltd. (Williams 1999, pp. 256–258).

Table 24.1

	Option 1	Option 2	Option 3	
First	£1200	£1000	£950	Period 1
Business	£900	£800	£600	
Economy	£500	£300	£200	
First	£1400	£1300	£1150	Period 2
Business	£1100	£900	£750	
Economy	£700	£400	£350	
First	£1500	£900	£850	Period 3
Business	£820	£800	£500	
Economy	£480	£470	£450	

Demand is uncertain but will be affected by price. Forecasts have been made of these demands according to a probability distribution which divides the demand levels into three scenarios for each period. The probabilities of the three scenarios in each period are:

Scenario 1	0.1
Scenario 2	0.7
Scenario 3	0.2

The forecast demands are shown in [Table 24.2](#).

Decide price levels for the current period, how many seats to sell in each class (depending on demand), the provisional number of planes to book and provisional price levels and seats to sell in future periods in order to maximize expected yield. You should schedule to be able to meet commitments under all possible combinations of scenarios.

With hindsight (i.e. not known until the beginning of the next period) it turned out that demand in each period (depending on the price level you chose) was as shown in [Table 24.3](#).

Use the actual demands that resulted from the prices you set in period 1 to rerun the model at the beginning of period 2 to set price levels for period 2 and provisional price levels for period 3.

Repeat this procedure with a rerun at the beginning of period 3. Give the final operational solution.

Contrast this solution to one obtained at the beginning of period 1 by pricing to maximize yield based on expected demands.

Table 24.2

	Price option 1	Price option 2	Price option 3	
First	10	15	20	Period 1
Business	20	25	35	Scenario 1
Economy	45	55	60	
First	20	25	35	Period 1
Business	40	42	45	Scenario 2
Economy	50	52	63	
First	45	50	60	Period 1
Business	45	46	47	Scenario 3
Economy	55	56	64	
First	20	25	35	Period 2
Business	42	45	46	Scenario 1
Economy	50	52	60	
First	10	40	50	Period 2
Business	50	60	80	Scenario 2
Economy	60	65	90	
First	50	55	80	Period 2
Business	20	30	50	Scenario 3
Economy	10	40	60	
First	30	35	40	Period 3
Business	40	50	55	Scenario 1
Economy	50	60	80	
First	30	40	60	Period 3
Business	10	40	45	Scenario 2
Economy	50	60	70	
First	50	70	80	Period 3
Business	40	45	60	Scenario 3
Economy	60	65	70	

Table 24.3

	Price option 1	Price option 2	Price option 3	
First	25	30	40	Period 1
Business	50	40	45	
Economy	50	53	65	
First	22	45	50	Period 2
Business	45	55	75	
Economy	50	60	80	
First	45	60	75	Period 3
Business	20	40	50	
Economy	55	60	75	

Mathematical Programming Formulation

Index Sets and Their Members

The following index sets and their members are used in this example:

- $\text{period} \in \text{PERIODS}$
- $\text{class} \in \text{CLASSES}$
- $\text{option} \in \text{OPTIONS}$
- $\text{scenario}, i \in \text{SCENARIOS}$
- $(i, j) \in \text{SCENARIOS2} = \text{SCENARIOS} \times \text{SCENARIOS}$
- $(i, j, k) \in \text{SCENARIOS3} = \text{SCENARIOS2} \times \text{SCENARIOS}$

Parameters

Table 24.4 shows the parameters used in this example.

Table 24.4 Parameters

Parameter Name	Interpretation
<i>num_seats[class]</i>	Number of seats per class per plane
<i>price[period,class,option]</i>	Price level (in £)
<i>prob[scenario]</i>	Probability of demand level scenario per period
<i>demand[period,scenario,class,option]</i>	Demand for period, scenario, and class if price level option is chosen
<i>actual_demand[period,class,option]</i>	Actual demand for period and class if price level option is chosen
<i>expected_demand[period,class,option]</i>	Expected demand for period and class if price level option is chosen
<i>num_periods</i>	Number of periods
<i>num_planes</i>	Number of planes available to be used
<i>plane_cost</i>	Cost to hire each plane (in £)
<i>transfer_fraction_ub</i>	Upper bound on fraction of seats that can be transferred to another category
<i>num_options</i>	Number of price level options
<i>actual_price[period,class]</i>	Actual price level for period and class
<i>actual_sales[period,class]</i>	Actual sales for period and class
<i>actual_revenue[period,class]</i>	Actual revenue for period and class
<i>current_period</i>	Current period for optimization

Variables

Table 24.5 shows the variables used in this example.

Table 24.5 Variables

Variable Name	Interpretation
P1[class,option]	1 if price option is chosen for class in period 1; 0 otherwise
P2[i,class,option]	1 if price option is chosen for class in period 2 as a result of scenario i in period 1; 0 otherwise
P3[i,j,class,option]	1 if price option is chosen for class in period 3 as a result of scenarios i and j in periods 1 and 2, respectively; 0 otherwise
S1[i,class,option]	Sales for class in period 1 under price option and scenario i
S2[i,j,class,option]	Sales for class in period 2 under price option and scenarios i and j in periods 1 and 2, respectively
S3[i,j,k,class,option]	Sales for class in period 3 under price option and scenarios i, j, k in periods 1, 2, 3, respectively
R1[i,class,option]	Revenue for class in period 1 under price option and scenario i
R2[i,j,class,option]	Revenue for class in period 2 under price option and scenarios i and j in periods 1 and 2, respectively
R3[i,j,k,class,option]	Revenue for class in period 3 under price option and scenarios i, j, k in periods 1, 2, 3, respectively
TransferFrom[i,j,k,class]	Number of seats transferred from class under scenarios i, j, k in periods 1, 2, 3, respectively
TransferTo[i,j,k,class]	Number of seats transferred to class under scenarios i, j, k in periods 1, 2, 3, respectively
NumPlanes	Number of planes to use

Objective

The objective is to maximize the following quadratic function:

$$\begin{aligned}
 & \text{ExpectedYield} \\
 = & \left(\begin{aligned} & \text{if } \text{current_period} \leq 1, \text{ then } \sum_{\substack{i \in \text{SCENARIOS}, \\ \text{class} \in \text{CLASSES}, \\ \text{option} \in \text{OPTIONS}}} \text{prob}[i] \cdot R1[i, \text{class}, \text{option}] \end{aligned} \right) \\
 + & \left(\begin{aligned} & \text{if } \text{current_period} \leq 2, \text{ then } \sum_{\substack{(i,j) \in \text{SCENARIOS2}, \\ \text{class} \in \text{CLASSES}, \\ \text{option} \in \text{OPTIONS}}} \text{prob}[i] \cdot \text{prob}[j] \cdot R2[i, j, \text{class}, \text{option}] \end{aligned} \right) \\
 + & \left(\begin{aligned} & \text{if } \text{current_period} \leq 3, \text{ then } \sum_{\substack{(i,j,k) \in \text{SCENARIOS3}, \\ \text{class} \in \text{CLASSES}, \\ \text{option} \in \text{OPTIONS}}} \text{prob}[i] \cdot \text{prob}[j] \cdot \text{prob}[k] \cdot R3[i, j, k, \text{class}, \text{option}] \end{aligned} \right) \\
 + & \sum_{\substack{\text{period} \in 1 \dots \text{current_period}-1, \\ \text{class} \in \text{CLASSES}}} \text{actual_revenue}[\text{period}, \text{class}] \\
 - & \text{plane_cost} \cdot \text{NumPlanes}
 \end{aligned}$$

where

$$\begin{aligned}
 R1[i, \text{class}, \text{option}] &= \text{price}[1, \text{class}, \text{option}] \cdot P1[\text{class}, \text{option}] \cdot S1[i, \text{class}, \text{option}] \\
 R2[i, j, \text{class}, \text{option}] &= \text{price}[2, \text{class}, \text{option}] \cdot P2[\text{class}, \text{option}] \cdot S2[i, j, \text{class}, \text{option}] \\
 R3[i, j, k, \text{class}, \text{option}] &= \text{price}[3, \text{class}, \text{option}] \cdot P3[\text{class}, \text{option}] \cdot S3[i, j, k, \text{class}, \text{option}]
 \end{aligned}$$

Constraints

The following constraints are used in this example:

- bounds on variables
- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$,

$$\begin{aligned}
 & \sum_{\text{option} \in \text{OPTIONS}} (\text{S1}[i, \text{class}, \text{option}] + \text{S2}[i, j, \text{class}, \text{option}] + \text{S3}[i, j, k, \text{class}, \text{option}]) \\
 & + \text{TransferFrom}[i, j, k, \text{class}] - \text{TransferTo}[i, j, k, \text{class}] \\
 & \leq \text{num_seats}[\text{class}] \cdot \text{NumPlanes}
 \end{aligned}$$

- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$,

$$\text{TransferFrom}[i,j,k,\text{class}] \leq \text{transfer_fraction_ub} \cdot \text{num_seats}[\text{class}]$$

- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$,

$$\text{TransferTo}[i,j,k,\text{class}] \leq \text{transfer_fraction_ub} \cdot \text{num_seats}[\text{class}]$$

- for $(i, j, k) \in \text{SCENARIOS3}$,

$$\sum_{\text{class} \in \text{CLASSES}} \text{TransferFrom}[i,j,k,\text{class}] = \sum_{\text{class} \in \text{CLASSES}} \text{TransferTo}[i,j,k,\text{class}]$$

- for $\text{class} \in \text{CLASSES}$,

$$\sum_{\text{option} \in \text{OPTIONS}} \text{P1}[\text{class}, \text{option}] = 1$$

- for $i \in \text{SCENARIOS}$ and $\text{class} \in \text{CLASSES}$,

$$\sum_{\text{option} \in \text{OPTIONS}} \text{P2}[i, \text{class}, \text{option}] = 1$$

- for $(i, j) \in \text{SCENARIOS2}$ and $\text{class} \in \text{CLASSES}$,

$$\sum_{\text{option} \in \text{OPTIONS}} \text{P3}[i,j,\text{class}, \text{option}] = 1$$

- for $i \in \text{SCENARIOS}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\text{S1}[i, \text{class}, \text{option}] \leq \text{demand}[1,i,\text{class}, \text{option}] \cdot \text{P1}[\text{class}, \text{option}]$$

- for $(i, j) \in \text{SCENARIOS2}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\text{S2}[i,j,\text{class}, \text{option}] \leq \text{demand}[2,j,\text{class}, \text{option}] \cdot \text{P2}[i, \text{class}, \text{option}]$$

- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\text{S3}[i,j,k,\text{class}, \text{option}] \leq \text{demand}[3,k,\text{class}, \text{option}] \cdot \text{P3}[i,j,\text{class}, \text{option}]$$

- for $i \in \text{SCENARIOS}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\text{R1}[i, \text{class}, \text{option}] \leq \text{price}[1,\text{class}, \text{option}] \cdot \text{S1}[i, \text{class}, \text{option}]$$

- for $i \in \text{SCENARIOS}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\begin{aligned} & \text{price}[1,\text{class}, \text{option}] \cdot \text{S1}[i, \text{class}, \text{option}] - \text{R1}[i, \text{class}, \text{option}] \\ & \leq \text{price}[1,\text{class}, \text{option}] \cdot \text{demand}[1,i,\text{class}, \text{option}] \cdot (1 - \text{P1}[\text{class}, \text{option}]) \end{aligned}$$

- for $(i, j) \in \text{SCENARIOS2}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$R2[i,j,\text{class},\text{option}] \leq \text{price}[2,\text{class},\text{option}] \cdot S2[i,j,\text{class},\text{option}]$$

- for $(i, j) \in \text{SCENARIOS2}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\begin{aligned} & \text{price}[2,\text{class},\text{option}] \cdot S2[i,j,\text{class},\text{option}] - R2[i,j,\text{class},\text{option}] \\ & \leq \text{price}[2,\text{class},\text{option}] \cdot \text{demand}[2,j,\text{class},\text{option}] \cdot (1 - P2[i,\text{class},\text{option}]) \end{aligned}$$

- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$R3[i,j,k,\text{class},\text{option}] \leq \text{price}[3,\text{class},\text{option}] \cdot S3[i,j,k,\text{class},\text{option}]$$

- for $(i, j, k) \in \text{SCENARIOS3}$ and $\text{class} \in \text{CLASSES}$ and $\text{option} \in \text{OPTIONS}$,

$$\begin{aligned} & \text{price}[3,\text{class},\text{option}] \cdot S3[i,j,k,\text{class},\text{option}] - R3[i,j,k,\text{class},\text{option}] \\ & \leq \text{price}[3,\text{class},\text{option}] \cdot \text{demand}[3,k,\text{class},\text{option}] \cdot (1 - P3[i,j,\text{class},\text{option}]) \end{aligned}$$

Input Data

The following data sets and macro variables contain the input data that are used in this example:

```
data class_data;
    input class $9. num_seats;
    datalines;
First      37
Business   38
Economy    47
;

data price_data;
    input period class $9. price1-price3;
    datalines;
1 First      1200 1000 950
1 Business    900 800 600
1 Economy     500 300 200
2 First      1400 1300 1150
2 Business   1100 900 750
2 Economy     700 400 350
3 First      1500 900 850
3 Business    820 800 500
3 Economy     480 470 450
;

data scenario_data;
    input prob;
    datalines;
0.1
```

```

0.7
0.2
;

data demand_data;
    input period scenario class $9. demand1-demand3;
    datalines;
1 1 First      10 15 20
1 1 Business   20 25 35
1 1 Economy    45 55 60
1 2 First      20 25 35
1 2 Business   40 42 45
1 2 Economy    50 52 63
1 3 First      45 50 60
1 3 Business   45 46 47
1 3 Economy    55 56 64
2 1 First      20 25 35
2 1 Business   42 45 46
2 1 Economy    50 52 60
2 2 First      10 40 50
2 2 Business   50 60 80
2 2 Economy    60 65 90
2 3 First      50 55 80
2 3 Business   20 30 50
2 3 Economy    10 40 60
3 1 First      30 35 40
3 1 Business   40 50 55
3 1 Economy    50 60 80
3 2 First      30 40 60
3 2 Business   10 40 45
3 2 Economy    50 60 70
3 3 First      50 70 80
3 3 Business   40 45 60
3 3 Economy    60 65 70
;

data actual_demand_data;
    input period class $9. demand1-demand3;
    datalines;
1 First      25 30 40
1 Business   50 40 45
1 Economy    50 53 65
2 First      22 45 50
2 Business   45 55 75
2 Economy    50 60 80
3 First      45 60 75
3 Business   20 40 50
3 Economy    55 60 75
;

%let num_periods = 3;
%let num_planes = 6;
%let plane_cost = 50000;
%let transfer_fraction_ub = 0.10;
%let num_options = 3;

```

PROC OPTMODEL Statements and Output

The first several PROC OPTMODEL statements declare index sets and parameters and then read input data:

```
proc optmodel;
  set PERIODS = 1..&num_periods;

  set <str> CLASSES;
  num num_seats {CLASSES};
  read data class_data into CLASSES=[class] num_seats;

  set OPTIONS = 1..&num_options;

  num price {PERIODS, CLASSES, OPTIONS};
  read data price_data into [period class]
    {option in OPTIONS} <price[period,class,option]=col('price' || option)>;

  set SCENARIOS;
  num prob {SCENARIOS};
  read data scenario_data into SCENARIOS=[_N_] prob;
  set SCENARIOS2 = SCENARIOS cross SCENARIOS;
  set SCENARIOS3 = SCENARIOS2 cross SCENARIOS;

  num demand {PERIODS, SCENARIOS, CLASSES, OPTIONS};
  read data demand_data into [period scenario class]
    {option in OPTIONS} <demand[period,scenario,class,option]=col('demand' || option)>;

  num actual_demand {PERIODS, CLASSES, OPTIONS};
  read data actual_demand_data into [period class]
    {option in OPTIONS} <actual_demand[period,class,option]=col('demand' || option)>;

  num actual_price {PERIODS, CLASSES};
  num actual_sales {PERIODS, CLASSES};
  num actual_revenue {PERIODS, CLASSES};

  num current_period;
```

The following VAR statements declare the decision variables:

```
var P1 {CLASSES, OPTIONS} binary;
var P2 {SCENARIOS, CLASSES, OPTIONS} binary;
var P3 {SCENARIOS2, CLASSES, OPTIONS} binary;

var S1 {SCENARIOS, CLASSES, OPTIONS} >= 0;
var S2 {SCENARIOS2, CLASSES, OPTIONS} >= 0;
var S3 {SCENARIOS3, CLASSES, OPTIONS} >= 0;

var R1 {SCENARIOS, CLASSES, OPTIONS} >= 0;
var R2 {SCENARIOS2, CLASSES, OPTIONS} >= 0;
```

```

var R3 {SCENARIOS3, CLASSES, OPTIONS} >= 0;

var TransferFrom {SCENARIOS3, CLASSES} >= 0;
var TransferTo {SCENARIOS3, CLASSES} >= 0;

var NumPlanes >= 0 <= &num_planes integer;

```

The following CON statement declares the constraints that enforce seat capacities:

```

con NumPlanes_con {<i,j,k> in SCENARIOS3, class in CLASSES}:
    sum {option in OPTIONS}
        (S1[i,class,option] + S2[i,j,class,option] + S3[i,j,k,class,option])
    + TransferFrom[i,j,k,class] - TransferTo[i,j,k,class]
    <= num_seats[class] * NumPlanes;

```

The following statements declare the constraints that restrict adjustment between classes:

```

for {<i,j,k> in SCENARIOS3, class in CLASSES} do;
    TransferFrom[i,j,k,class].ub = &transfer_fraction_ub * num_seats[class];
    TransferTo[i,j,k,class].ub   = &transfer_fraction_ub * num_seats[class];
end;
con Balance_con {<i,j,k> in SCENARIOS3}:
    sum {class in CLASSES} TransferFrom[i,j,k,class]
    = sum {class in CLASSES} TransferTo[i,j,k,class];

```

The following CON statements declare the constraints that enforce one price level per class:

```

con P1_con {class in CLASSES}:
    sum {option in OPTIONS} P1[class,option] = 1;
con P2_con {i in SCENARIOS, class in CLASSES}:
    sum {option in OPTIONS} P2[i,class,option] = 1;
con P3_con {<i,j> in SCENARIOS2, class in CLASSES}:
    sum {option in OPTIONS} P3[i,j,class,option] = 1;

```

The following CON statements declare the constraints that sales cannot exceed demand:

```

con S1_con {i in SCENARIOS, class in CLASSES, option in OPTIONS}:
    S1[i,class,option] <= demand[1,i,class,option] * P1[class,option];
con S2_con {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS}:
    S2[i,j,class,option] <= demand[2,j,class,option] * P2[i,class,option];
con S3_con {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS}:
    S3[i,j,k,class,option] <= demand[3,k,class,option] * P3[i,j,class,option];

```

The following CON statements encode one possible linearization of the quadratic objective:

```

/* R1[i,class,option] =
    price[1,class,option] * P1[class,option] * S1[i,class,option] */
con R1_con_a {i in SCENARIOS, class in CLASSES, option in OPTIONS}:
    R1[i,class,option] <= price[1,class,option] * S1[i,class,option];
con R1_con_b {i in SCENARIOS, class in CLASSES, option in OPTIONS}:
    price[1,class,option] * S1[i,class,option] - R1[i,class,option]
<= price[1,class,option] * demand[1,i,class,option] * (1 - P1[class,option]);

/* R2[i,j,class,option] =
    price[2,class,option] * P2[i,class,option] * S2[i,j,class,option] */
con R2_con_a {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS}:

```

```

R2[i,j,class,option] <= price[2,class,option] * S2[i,j,class,option];
con R2_con_b {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS}:
    price[2,class,option] * S2[i,j,class,option] - R2[i,j,class,option]
<= price[2,class,option] * demand[2,j,class,option] * (1 - P2[i,class,option]);

/* R3[i,j,k,class,option] =
    price[3,class,option] * P3[i,j,class,option] * S3[i,j,k,class,option] */
con R3_con_a {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS}:
    R3[i,j,k,class,option] <= price[3,class,option] * S3[i,j,k,class,option];
con R3_con_b {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS}:
    price[3,class,option] * S3[i,j,k,class,option] - R3[i,j,k,class,option]
<= price[3,class,option] * demand[3,k,class,option] * (1 - P3[i,j,class,option]);

```

An alternative “compact linearization” (not shown) involves fewer constraints, as in [Chapter 10](#).

The following MAX statement declares the linearized objective, which depends on *current_period*:

```

max ExpectedYield =
    (if current_period <= 1
     then sum {i in SCENARIOS, class in CLASSES, option in OPTIONS}
         prob[i] * R1[i,class,option])
+ (if current_period <= 2
     then sum {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS}
         prob[i] * prob[j] * R2[i,j,class,option])
+ (if current_period <= 3
     then sum {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS}
         prob[i] * prob[j] * prob[k] * R3[i,j,k,class,option])
+ sum {period in 1..current_period-1, class in CLASSES}
    actual_revenue[period,class]
- &plane_cost * NumPlanes;

```

The following NUM statements use the *.sol* variable suffix to compute the recommended prices from the optimal values of the decision variables:

```

num price_sol_1 {class in CLASSES} =
    sum {option in OPTIONS} price[1,class,option] * P1[class,option].sol;
num price_sol_2 {class in CLASSES, i in SCENARIOS} =
    sum {option in OPTIONS} price[2,class,option] * P2[i,class,option].sol;
num price_sol_3 {class in CLASSES, <i,j> in SCENARIOS2} =
    sum {option in OPTIONS} price[3,class,option] * P3[i,j,class,option].sol;

```

The following NUM statements use the *.sol* variable suffix to compute the recommended numbers of seats to sell:

```

num remaining_seats {class in CLASSES} =
    num_seats[class] * NumPlanes.sol
- sum {period in 1..current_period-1} actual_sales[period,class];
num sell_up_to_1 {class in CLASSES} =
    min(
        max {i in SCENARIOS, option in OPTIONS} S1[i,class,option].sol,
        remaining_seats[class]);
num sell_up_to_2 {class in CLASSES} =
    min(
        max {<i,j> in SCENARIOS2, option in OPTIONS} S2[i,j,class,option].sol,
        remaining_seats[class]);

```

```

num sell_up_to_3 {class in CLASSES} =
  min(
    max {<i,j,k> in SCENARIOS3, option in OPTIONS} S3[i,j,k,class,option].sol,
    remaining_seats[class]);

```

The following statements call the mixed integer linear programming solver to determine the optimal prices for period 1:

```

current_period = 1;
solve;
for {i in SCENARIOS, class in CLASSES, option in OPTIONS}
  S1[i,class,option] = round(S1[i,class,option].sol);
print price_sol_1;
print sell_up_to_1;
print {i in SCENARIOS, class in CLASSES, option in OPTIONS:
  S1[i,class,option].sol > 0} S1;
print price_sol_2;
print price_sol_3;
print NumPlanes ExpectedYield;

```

The following statements fix the resulting prices for period 1 and use the MIN function to limit sales based on actual demand:

```

for {class in CLASSES, option in OPTIONS} do;
  if P1[class,option].sol > 0.5 then do;
    fix P1[class,option] = 1;
    actual_price[1,class] = price_sol_1[class];
    actual_sales[1,class] =
      min(sell_up_to_1[class], actual_demand[1,class,option]);
    for {i in SCENARIOS} fix S1[i,class,option] = actual_sales[1,class];
  end;
  else fix P1[class,option] = 0;
end;
for {class in CLASSES}
  actual_revenue[1,class] = actual_price[1,class] * actual_sales[1,class];
print actual_price actual_sales actual_revenue;

```

Figure 24.1 shows the output from the mixed integer linear programming solver for period 1.

Figure 24.1 Output from Mixed Integer Linear Programming Solver, Period 1

The OPTMODEL Procedure	
Problem Summary	
Objective Sense	Maximization
Objective Function	ExpectedYield
Objective Type	Linear
Number of Variables	982
Bounded Above	0
Bounded Below	702
Bounded Below and Above	280
Free	0
Fixed	0
Binary	117
Integer	1
Number of Constraints	1200
Linear LE (\leq)	1134
Linear EQ ($=$)	66
Linear GE (\geq)	0
Linear Range	0
Constraint Coefficients	3708
Solution Summary	
Solver	MILP
Objective Function	ExpectedYield
Solution Status	Optimal
Objective Value	169543.624
Iterations	1095
Best Bound	.
Nodes	3
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	3.637979E-12
Bound Infeasibility	3.885781E-12
Integer Infeasibility	3.469447E-17
price_	
[1]	sol_1
Business	900
Economy	500
First	1200

Figure 24.1 *continued*

		sell_ up_to_	
[1]		1	
Business		45	
Economy		55	
First		45	
[1]	[2]	[3]	S1
1	Business	1	20
1	Economy	1	45
1	First	1	10
2	Business	1	40
2	Economy	1	50
2	First	1	20
3	Business	1	45
3	Economy	1	55
3	First	1	45
	price_sol_2		
	1	2	3
Business	1100	1100	1100
Economy	700	700	700
First	1150	1150	1300

Figure 24.1 continued

	[1]	[2]	[3]	price_ sol_3	
	Business	1	1	800	
	Business	1	2	800	
	Business	1	3	800	
	Business	2	1	800	
	Business	2	2	800	
	Business	2	3	800	
	Business	3	1	800	
	Business	3	2	800	
	Business	3	3	800	
	Economy	1	1	480	
	Economy	1	2	480	
	Economy	1	3	450	
	Economy	2	1	480	
	Economy	2	2	480	
	Economy	2	3	450	
	Economy	3	1	480	
	Economy	3	2	480	
	Economy	3	3	450	
	First	1	1	1500	
	First	1	2	1500	
	First	1	3	1500	
	First	2	1	1500	
	First	2	2	1500	
	First	2	3	1500	
	First	3	1	1500	
	First	3	2	1500	
	First	3	3	1500	
	Num Planes			Expected Yield	
	3			169544	
			actual_ price	actual_ sales	actual_ revenue
	1	Business	900	45	40500
	1	Economy	500	50	25000
	1	First	1200	25	30000

The following statements drop the period 1 constraints and call the mixed integer linear programming solver to determine the optimal prices for period 2:

```
drop P1_con S1_con R1_con_a R1_con_b;
current_period = 2;
solve;
for {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS}
    S2[i,j,class,option] = round(S2[i,j,class,option].sol);
print price_sol_2;
```

```

print sell_up_to_2;
print {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS:
      i = 1 and S2[1,j,class,option].sol > 0} S2;
print price_sol_3;
print NumPlanes ExpectedYield;

```

The following statements fix the resulting prices for period 2 and use the MIN function to limit sales based on actual demand:

```

for {i in SCENARIOS, class in CLASSES, option in OPTIONS} do;
  if P2[i,class,option].sol > 0.5 then do;
    fix P2[i,class,option] = 1;
    actual_price[2,class] = price_sol_2[class,i];
    actual_sales[2,class] =
      min(sell_up_to_2[class], actual_demand[2,class,option]);
    for {j in SCENARIOS} fix S2[i,j,class,option] = actual_sales[2,class];
  end;
  else fix P2[i,class,option] = 0;
end;
for {class in CLASSES}
  actual_revenue[2,class] = actual_price[2,class] * actual_sales[2,class];
print actual_price actual_sales actual_revenue;

```

Figure 24.2 shows the output from the mixed integer linear programming solver for period 2.

Figure 24.2 Output from Mixed Integer Linear Programming Solver, Period 2

Problem Summary	
Objective Sense	Maximization
Objective Function	ExpectedYield
Objective Type	Linear
Number of Variables	982
Bounded Above	0
Bounded Below	693
Bounded Below and Above	271
Free	0
Fixed	18
Binary	108
Integer	10
Number of Constraints	1116
Linear LE (<=)	1053
Linear EQ (=)	63
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	3510

Figure 24.2 continued

Solution Summary				
Solver	MILP			
Objective Function	ExpectedYield			
Solution Status	Optimal			
Objective Value	172968.66			
Iterations	769			
Best Bound	.			
Nodes	1			
Relative Gap	0			
Absolute Gap	0			
Primal Infeasibility	3.637979E-12			
Bound Infeasibility	0			
Integer Infeasibility	0			
price_sol_2				
	1	2	3	
Business	1100	1100	1100	
Economy	700	700	700	
First	1150	1150	1150	
sell_				
up_to_				
[1]	2			
Business	50			
Economy	60			
First	60			
[1]	[2]	[3]	[4]	S2
1	1	Business	1	42
1	1	Economy	1	50
1	1	First	3	35
1	2	Business	1	50
1	2	Economy	1	60
1	2	First	3	50
1	3	Business	1	20
1	3	Economy	1	10
1	3	First	3	60

Figure 24.2 continued

			price_ sol_3	
[1]	[2]	[3]		
Business	1	1	800	
Business	1	2	800	
Business	1	3	800	
Business	2	1	800	
Business	2	2	800	
Business	2	3	800	
Business	3	1	800	
Business	3	2	800	
Business	3	3	800	
Economy	1	1	480	
Economy	1	2	480	
Economy	1	3	450	
Economy	2	1	480	
Economy	2	2	480	
Economy	2	3	450	
Economy	3	1	480	
Economy	3	2	480	
Economy	3	3	450	
First	1	1	1500	
First	1	2	1500	
First	1	3	1500	
First	2	1	1500	
First	2	2	1500	
First	2	3	1500	
First	3	1	1500	
First	3	2	1500	
First	3	3	1500	
Num Planes		Expected Yield		
3		172969		
[1]	[2]	actual_ price	actual_ sales	actual_ revenue
1	Business	900	45	40500
1	Economy	500	50	25000
1	First	1200	25	30000
2	Business	1100	45	49500
2	Economy	700	50	35000
2	First	1150	50	57500

The following statements drop the period 2 constraints and call the mixed integer linear programming solver to determine the optimal prices for period 3:

```

current_period = 3;
drop P2_con S2_con R2_con_a R2_con_b;
solve;
for {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS}
    S3[i,j,k,class,option] = round(S3[i,j,k,class,option].sol);
print price_sol_3;
print sell_up_to_3;
print {<i,j,k> in SCENARIOS3, class in CLASSES, option in OPTIONS:
    <i,j> in {<1,1>} and S3[i,j,k,class,option].sol > 0} S3;
print NumPlanes ExpectedYield;

```

The following statements fix the resulting prices for period 3 and use the MIN function to limit sales based on actual demand:

```

for {<i,j> in SCENARIOS2, class in CLASSES, option in OPTIONS} do;
    if P3[i,j,class,option].sol > 0.5 then do;
        fix P3[i,j,class,option] = 1;
        actual_price[3,class] = price_sol_3[class,i,j];
        actual_sales[3,class] =
            min(sell_up_to_3[class], actual_demand[3,class,option]);
        for {k in SCENARIOS} fix S3[i,j,k,class,option] = actual_sales[3,class];
    end;
    else fix P3[i,j,class,option] = 0;
end;
for {class in CLASSES}
    actual_revenue[3,class] = actual_price[3,class] * actual_sales[3,class];
print actual_price actual_sales actual_revenue;

```

Figure 24.3 shows the output from the mixed integer linear programming solver for period 3.

Figure 24.3 Output from Mixed Integer Linear Programming Solver, Period 3

Problem Summary	
Objective Sense	Maximization
Objective Function	ExpectedYield
Objective Type	Linear
Number of Variables	982
Bounded Above	0
Bounded Below	666
Bounded Below and Above	244
Free	0
Fixed	72
Binary	81
Integer	37
Number of Constraints	864
Linear LE (<=)	810
Linear EQ (=)	54
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	2916

Figure 24.3 continued

Solution Summary			
Solver	MILP		
Objective Function	ExpectedYield		
Solution Status	Optimal		
Objective Value	176392.4		
Iterations	650		
Best Bound	.		
Nodes	1		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	4.440892E-16		
Bound Infeasibility	0		
Integer Infeasibility	0		
			price_
[1]	[2]	[3]	sol_3
Business	1	1	800
Business	1	2	800
Business	1	3	800
Business	2	1	800
Business	2	2	800
Business	2	3	800
Business	3	1	800
Business	3	2	800
Business	3	3	800
Economy	1	1	480
Economy	1	2	480
Economy	1	3	480
Economy	2	1	480
Economy	2	2	480
Economy	2	3	480
Economy	3	1	480
Economy	3	2	480
Economy	3	3	480
First	1	1	1500
First	1	2	1500
First	1	3	1500
First	2	1	1500
First	2	2	1500
First	2	3	1500
First	3	1	1500
First	3	2	1500
First	3	3	1500
			sell_
			up_to_
[1]			3
Business			24
Economy			41
First			36

Figure 24.3 continued

[1]	[2]	[3]	[4]	[5]	S3
1	1	1	Business	2	28
1	1	1	Economy	1	41
1	1	1	First	1	30
1	1	2	Business	2	28
1	1	2	Economy	1	41
1	1	2	First	1	30
1	1	3	Business	2	25
1	1	3	Economy	1	36
1	1	3	First	1	40

Num Planes	Expected Yield
3	176392

[1]	[2]	actual_ price	actual_ sales	actual_ revenue
1	Business	900	45	40500
1	Economy	500	50	25000
1	First	1200	25	30000
2	Business	1100	45	49500
2	Economy	700	50	35000
2	First	1150	50	57500
3	Business	800	24	19200
3	Economy	480	41	19680
3	First	1500	36	54000

The following statements print the expected yield that results from the optimal prices, with sales limited by actual demands:

```
current_period = 4;
print ExpectedYield;
quit;
```

Figure 24.4 shows the final expected yield.

Figure 24.4 Final Expected Yield

Expected Yield
180380

Maximizing yield based on expected demands is much simpler than the stochastic programming approach and requires only one solver call. The first several PROC OPTMODEL statements are the same as before:

```

proc optmodel;
  set PERIODS = 1..&num_periods;

  set <str> CLASSES;
  num num_seats {CLASSES};
  read data class_data into CLASSES=[class] num_seats;

  set OPTIONS = 1..&num_options;

  num price {PERIODS, CLASSES, OPTIONS};
  read data price_data into [period class]
    {option in OPTIONS} <price[period,class,option]=col('price' || option)>;

  set SCENARIOS;
  num prob {SCENARIOS};
  read data scenario_data into SCENARIOS=[_N_] prob;

  num demand {PERIODS, SCENARIOS, CLASSES, OPTIONS};
  read data demand_data into [period scenario class]
    {option in OPTIONS} <demand[period,scenario,class,option]=col('demand' || option)>;

  num actual_demand {PERIODS, CLASSES, OPTIONS};
  read data actual_demand_data into [period class]
    {option in OPTIONS} <actual_demand[period,class,option]=col('demand' || option)>;

  num actual_price {PERIODS, CLASSES};
  num actual_sales {PERIODS, CLASSES};
  num actual_revenue {PERIODS, CLASSES};

```

The following NUM statement declares the *expected_demand* parameter as a weighted sum of *demand*:

```

num expected_demand {period in PERIODS, class in CLASSES, option in OPTIONS} =
  sum {scenario in SCENARIOS}
    prob[scenario] * demand[period,scenario,class,option];

```

Note that the variables, constraints, and parameters are unified and simplified into fewer families than before:

```

var P {PERIODS, CLASSES, OPTIONS} binary;
var S {PERIODS, CLASSES, OPTIONS} >= 0;
var R {PERIODS, CLASSES, OPTIONS} >= 0;

var TransferFrom {CLASSES} >= 0;
var TransferTo {CLASSES} >= 0;

var NumPlanes >= 0 <= &num_planes integer;

con NumPlanes_con {class in CLASSES}:
  sum {period in PERIODS, option in OPTIONS} S[period,class,option]
  + TransferFrom[class] - TransferTo[class]
  <= num_seats[class] * NumPlanes;

for {class in CLASSES} do;
  TransferFrom[class].ub = &transfer_fraction_ub * num_seats[class];
  TransferTo[class].ub   = &transfer_fraction_ub * num_seats[class];

```

```

end;
con Balance_con:
    sum {class in CLASSES} TransferFrom[class]
    = sum {class in CLASSES} TransferTo[class];

con P_con {period in PERIODS, class in CLASSES}:
    sum {option in OPTIONS} P[period,class,option] = 1;

con S_con {period in PERIODS, class in CLASSES, option in OPTIONS}:
    S[period,class,option]
<= expected_demand[period,class,option] * P[period,class,option];

/* R[period,class,option] =
    price[period,class,option] * P[period,class,option] * S[period,class,option] */
con R_con_a {period in PERIODS, class in CLASSES, option in OPTIONS}:
    R[period,class,option] <= price[period,class,option] * S[period,class,option];
con R_con_b {period in PERIODS, class in CLASSES, option in OPTIONS}:
    price[period,class,option] * S[period,class,option] - R[period,class,option]
<= price[period,class,option] * expected_demand[period,class,option]
    * (1 - P[period,class,option]);

max Yield =
    sum {period in PERIODS, class in CLASSES, option in OPTIONS}
        R[period,class,option]
    - &plane_cost * NumPlanes;

num price_sol {period in PERIODS, class in CLASSES} =
    sum {option in OPTIONS} price[period,class,option] * P[period,class,option].sol;

solve;
for {period in PERIODS, class in CLASSES, option in OPTIONS}
    S[period,class,option] = round(S[period,class,option].sol);
print price_sol;
print {period in PERIODS, class in CLASSES, option in OPTIONS:
    S[period,class,option].sol > 0} S;
print NumPlanes Yield;
for {period in PERIODS, class in CLASSES, option in OPTIONS} do;
    if P[period,class,option].sol > 0.5 then do;
        actual_price[period,class] = price_sol[period,class];
        actual_sales[period,class] =
            min(S[period,class,option], actual_demand[period,class,option]);
        actual_revenue[period,class] =
            actual_price[period,class] * actual_sales[period,class];
        R[period,class,option] = actual_revenue[period,class];
    end;
end;
print actual_price actual_sales actual_revenue;
print Yield;

```

Figure 24.5 shows the output from the mixed integer linear programming solver for the problem based on expected demands.

Figure 24.5 Output from Mixed Integer Linear Programming Solver, Based on Expected Demands

The OPTMODEL Procedure			
Problem Summary			
Objective Sense	Maximization		
Objective Function	Yield		
Objective Type	Linear		
Number of Variables	88		
Bounded Above	0		
Bounded Below	54		
Bounded Below and Above	34		
Free	0		
Fixed	0		
Binary	27		
Integer	1		
Number of Constraints	94		
Linear LE (\leq)	84		
Linear EQ ($=$)	10		
Linear GE (\geq)	0		
Linear Range	0		
Constraint Coefficients	258		
Solution Summary			
Solver	MILP		
Objective Function	Yield		
Solution Status	Optimal		
Objective Value	180309		
Iterations	99		
Best Bound	.		
Nodes	1		
Relative Gap	0		
Absolute Gap	0		
Primal Infeasibility	1.455192E-11		
Bound Infeasibility	5.811324E-16		
Integer Infeasibility	5.811324E-16		
price_sol			
	Business	Economy	First
1	900	500	1200
2	1100	700	1150
3	800	480	1500

Figure 24.5 continued

[1]	[2]	[3]	S
1	Business	1	39
1	Economy	1	51
1	First	1	24
2	Business	1	43
2	Economy	1	49
2	First	3	55
3	Business	2	35
3	Economy	1	37
3	First	1	34

Num	Yield
Planes	
3	180309

[1]	[2]	actual_ price	actual_ sales	actual_ revenue
1	Business	900	39	35100
1	Economy	500	50	25000
1	First	1200	24	28800
2	Business	1100	43	47300
2	Economy	700	49	34300
2	First	1150	50	57500
3	Business	800	35	28000
3	Economy	480	37	17760
3	First	1500	34	51000

Yield
174760

As anticipated, the yield is smaller than in Figure 24.4 because there is no opportunity here to change prices based on actual demand from previous periods.

The following statements show the modifications to maximize yield based on actual demands:

```
drop S_con R_con_b;

con S_con_actual {period in PERIODS, class in CLASSES, option in OPTIONS}:
    S[period,class,option]
<= actual_demand[period,class,option] * P[period,class,option];

con R_con_b_actual {period in PERIODS, class in CLASSES, option in OPTIONS}:
    price[period,class,option] * S[period,class,option] - R[period,class,option]
<= price[period,class,option] * actual_demand[period,class,option]
    * (1 - P[period,class,option]);

solve;
```

```

for {period in PERIODS, class in CLASSES, option in OPTIONS}
  S[period,class,option] = round(S[period,class,option].sol);
print price_sol;
print {period in PERIODS, class in CLASSES, option in OPTIONS:
  S[period,class,option].sol > 0} S;
print NumPlanes Yield;

```

Since actual demand is considered directly in this formulation, the postprocessing steps do not need to reduce sales to actual demand:

```

for {period in PERIODS, class in CLASSES, option in OPTIONS} do;
  if P[period,class,option].sol > 0.5 then do;
    actual_price[period,class] = price_sol[period,class];
    actual_sales[period,class] = S[period,class,option];
    actual_revenue[period,class] =
      actual_price[period,class] * actual_sales[period,class];
    R[period,class,option] = actual_revenue[period,class];
  end;
end;
print actual_price actual_sales actual_revenue;
quit;

```

Figure 24.6 shows the output from the mixed integer linear programming solver for the problem based on actual demands.

Figure 24.6 Output from Mixed Integer Linear Programming Solver, Based on Actual Demands

Problem Summary	
Objective Sense	Maximization
Objective Function	Yield
Objective Type	Linear
Number of Variables	88
Bounded Above	0
Bounded Below	54
Bounded Below and Above	34
Free	0
Fixed	0
Binary	27
Integer	1
Number of Constraints	94
Linear LE (<=)	84
Linear EQ (=)	10
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	258

Figure 24.6 continued

Solution Summary				
Solver	MILP			
Objective Function	Yield			
Solution Status	Optimal			
Objective Value	193964			
Iterations	168			
Best Bound	.			
Nodes	1			
Relative Gap	0			
Absolute Gap	0			
Primal Infeasibility	6.394885E-14			
Bound Infeasibility	2.159375E-12			
Integer Infeasibility	1.814412E-15			
price_sol				
	Business	Economy	First	
1	900	500	1200	
2	1100	700	1300	
3	820	480	1500	
	[1]	[2]	[3] S	
1	Business	1	50	
1	Economy	1	50	
1	First	1	25	
2	Business	1	45	
2	Economy	1	50	
2	First	2	45	
3	Business	1	20	
3	Economy	1	36	
3	First	1	45	
Num				
	Planes	Yield		
	3	193964		
		actual_	actual_	actual_
[1]	[2]	price	sales	revenue
1	Business	900	50	45000
1	Economy	500	50	25000
1	First	1200	25	30000
2	Business	1100	45	49500
2	Economy	700	50	35000
2	First	1300	45	58500
3	Business	820	20	16400
3	Economy	480	36	17280
3	First	1500	45	67500

As anticipated, the yield is highest here because optimal prices and sales are determined with perfect knowledge of demand.

The stochastic programming formulation shown earlier represents a natural compromise between the deterministic formulations based on expected demand or actual demand.

Features Demonstrated

The following features are demonstrated in this example:

- problem type: mixed integer linear programming (stochastic programming with recourse)
- numeric and string index sets
- multiple input data sets
- CROSS set operator
- sets of tuples
- `.ub` variable suffix
- linearization of product of continuous variable and binary variable
- compact linearization
- IF-THEN expression
- using a variable suffix (such as `.sol`) in the declaration of a numeric parameter
- MAX aggregation operator
- ROUND function
- IF-THEN statement
- using a colon (:) to select members of a set
- FIX statement
- MIN function
- DROP statement
- multiple SOLVE statements

References

Liberti, Leo (2007), “Compact Linearization for Binary Quadratic Problems,” *4OR: A Quarterly Journal of Operations Research*, 5(3), 231–245.

Williams, H. Paul (1974), “Experiments in the Formulation of Integer Programming Problems,” *Mathematical Programming Studies*, 2, 180–197.

Williams, H. Paul (1999), *Model Building in Mathematical Programming*, West Sussex, England: John Wiley & Sons Ltd. Reproduced with permission of John Wiley & Sons Ltd.

Subject Index

- binary variables
 - modeling if-then constraints, 152, 173, 203, 236
 - product of, 125
- .body** constraint suffix, 27, 110, 222
- calculated numeric parameter, 83
- colon (:) operator
 - selecting members of a set, 6, 25, 55, 72, 97, 173, 222, 236, 245, 268, 294
- compact linearization, 127, 293
- concatenation
 - string, 162
- connected components, 266, 270
- constraint suffix
 - .body** suffix, 27, 110, 222
 - .dual** suffix, 39, 183, 222, 245
 - .lb** suffix, 112
 - modifying the right-hand side of a constraint, 112
 - .ub** suffix, 112, 222
- data set
 - multiple input, 4, 25, 42, 54, 82, 94, 107, 122, 161, 171, 192, 210, 220, 234, 244, 254, 289
 - multiple output, 12, 27, 44, 56, 84, 98, 180, 194, 257
- .dual** constraint suffix, 39, 183, 222, 245
- dynamic Leontief input-output model, 103
- fixed costs, 233
- generalized network flow with side constraints, 63
- if-then constraints
 - modeling with binary variables, 152, 173, 203, 236
- implicit slice, 55, 71, 152, 222, 236
- index set
 - creating an output data set, 173
 - in PRINT statement, 73
 - numeric, 2, 25, 32, 43, 51, 66, 80, 90, 104, 120, 132, 149, 159, 253, 284
 - sets indexed by other sets, 162, 201, 269
 - string, 2, 25, 32, 43, 51, 66, 80, 90, 104, 120, 149, 159, 253, 284
 - subset, 173
- input data set
 - multiple, 4, 25, 42, 54, 82, 94, 107, 122, 161, 171, 192, 210, 220, 234, 244, 254, 289
- L_1 and L_∞ norms, 157
- polynomial regression, 131
- .lb** constraint suffix
 - modifying the right-hand side of a constraint, 112
- .lb** variable suffix, 109
- linear programming, 1, 31, 49, 63, 89, 103, 131, 177, 207, 215, 251
- linearization
 - compact, 127, 293
 - product of binary variables, 127
 - product of continuous variable and binary variable, 292
 - ratio constraint, 7, 72, 73, 84
- loop
 - calling a solver, 256, 270
- mixed integer linear programming, 23, 41, 79, 119, 147, 157, 169, 177, 189, 199, 231, 263, 281
- modeling startup costs, 177, 189
- modifying the right-hand side of a constraint
 - .lb** constraint suffix, 112
 - .ub** constraint suffix, 112
- multiple objectives, 56, 104, 105, 109, 131, 132, 134, 158, 160, 207, 208, 211, 219
- network flow with side constraints, 215
- nonlinear programming
 - quadratic, 241
- numeric parameter
 - calculated, 83
- objective
 - multiple, 56, 104, 105, 109, 131, 132, 134, 158, 160, 207, 208, 211, 219
- output data set
 - creating from index set, 173
 - multiple, 12, 27, 44, 56, 84, 98, 180, 194, 257
- polynomial regression with L_1 and L_∞ norms, 131
- problem type
 - linear programming, 1, 31, 49, 63, 89, 103, 131, 177, 207, 215, 251
 - mixed integer linear programming, 23, 41, 79, 119, 147, 157, 169, 177, 189, 199, 231, 263, 281
 - nonlinear programming, 241
 - quadratic programming, 241
- product

- of binary variables, 125
 - of continuous variable and binary variable, 292
 - of decision variables, 245
- quadratic assignment problem, 119
- quadratic program, 241
- range constraint, 7, 72, 98
- ratio constraint, 7, 72, 73, 84
- reading data, 5, 35
 - dense, 5, 25, 36, 43, 55, 107, 123, 255
 - sparse, 35, 55, 71, 244
 - sparse (upper triangular), 124
 - two-dimensional, 5, 25, 35, 36, 43, 55, 71, 107, 123, 124, 244, 255
- routing
 - vehicle, 263
- row generation, 270
- selecting members of a set
 - colon (:) operator, 6, 25, 55, 72, 97, 173, 222, 236, 245, 268, 294
- sets indexed by other sets, 162, 201, 269
- sets of tuples, 55, 71, 125, 151, 201, 221, 235, 268, 291
- slice
 - implicit, 55, 71, 152, 222, 236
- .sol** variable suffix, 12, 203, 256, 270, 293
- solution
 - storing in numerical parameter, 256
- startup costs
 - modeling, 177, 189
- stochastic programming with recourse, 281
- storing a solution in a numeric parameter, 256
- string concatenation, 162
- suffix
 - constraint suffix, 27, 39, 110, 112, 183, 222, 245
 - variable suffix, 12, 25, 26, 55, 71, 73, 80, 83, 96, 109, 135, 163, 193, 203, 256, 270, 292, 293
- symmetry-breaking constraints, 269
- totally unimodular, 169, 173
- tuples
 - sets of, 55, 71, 125, 151, 201, 221, 235, 268, 291
- .ub** constraint suffix, 222
 - modifying the right-hand side of a constraint, 112
- .ub** variable suffix, 26, 55, 71, 80, 83, 96, 109, 193, 292
- variable suffix
 - in parameter declaration, 12, 25, 73, 80, 83, 135, 163, 256, 293
 - .lb** suffix, 109
 - .sol** suffix, 12, 203, 256, 270, 293
 - .ub** suffix, 26, 55, 71, 80, 83, 96, 109, 193, 292
- vehicle routing, 263
- writing data
 - dense, 98, 257
 - sparse, 99, 257
 - two-dimensional, 98, 99, 257

Syntax Index

- ABS function, 135, 163, 211
- AND aggregation operator, 270
- AND logical operator, 173
- BINARY option
 - VAR statement, 26
- CALL SYMPUT, 271
- CARD function, 163, 181, 193, 203, 270
- CLP procedure, 210
- colon (:) operator, 6, 25, 55, 72, 97, 173, 222, 236, 245, 268, 294
- CREATE DATA statement, 12, 27, 39, 44, 56, 73, 84, 98, 134, 135, 173, 181, 194, 256, 271
- CROSS set operator, 71, 201, 291
- DIFF set operator, 6, 25, 54, 71, 84, 97, 226, 269
- DO loop
 - calling a solver, 256
- DO UNTIL loop
 - calling a solver, 270
- DROP statement, 127, 297
- EXPAND statement, 7
- FILE statement, 203
- FINANCE function, 96
- FIX statement, 6, 25, 37, 43, 54, 95, 152, 183, 193, 236, 294
- IF-THEN/ELSE expression, 37, 43, 109, 181, 193, 236, 293
- IF-THEN/ELSE statement, 294
- IMPVAR statement, 6, 25, 37, 43, 83, 97, 109, 125, 134, 181, 192, 211, 236, 249
- IN expression, 173
- INDEX function, 72
- INIT option
 - VAR statement, 35, 124, 222, 236, 244, 269
- INTEGER option
 - VAR statement, 44
- MAX aggregation operator, 6, 25, 37, 43, 135, 163, 293
- MIN function, 294
- NOMISS option
 - READ DATA statement, 71
- OBJ option
- SOLVE statement, 56
- PRIMALIN option
 - SOLVE WITH MILP statement, 166
- PRINT statement, 110, 125, 164, 181, 194, 203, 212, 222, 237, 245, 256, 270, 294
 - with index set, 73
- PROBLEM statement, 110, 135
- PUT statement, 203
- READ DATA statement, 5, 25, 35, 43, 54, 71, 83, 95, 107, 123, 162, 172, 180, 192, 201, 210, 221, 235, 244, 255, 268, 291
 - NOMISS option, 71
- RELAXINT option
 - SOLVE WITH LP statement, 183
- ROUND function, 294
- set operator
 - CROSS, 71, 201, 291
 - DIFF, 6, 25, 54, 71, 84, 97, 226, 269
 - SETOF, 72, 163, 202, 226
 - UNION, 6, 25, 71, 95, 107, 152, 162, 210, 222, 236, 270
- SETOF set operator, 72, 163, 202, 226
- SGPLOT procedure, 142, 271
- SLICE expression, 56
- SOLVE statement
 - multiple, 294
 - OBJ option, 56
 - SOLVER= option, 12
 - WITH clause, 248
- SOLVE WITH LP statement
 - RELAXINT option, 183
- SOLVE WITH MILP statement
 - PRIMALIN option, 166
- SOLVER= option, 224, 247
 - SOLVE statement, 12
- SUBSTR function, 6, 25
- UNFIX statement, 185
- UNION set operator, 6, 25, 71, 95, 107, 152, 162, 210, 222, 236, 270
- USE PROBLEM statement, 110, 135
- VAR statement
 - BINARY option, 26
 - bounds, 6, 25, 37, 43, 54, 71, 83, 95
 - INIT option, 35, 124, 222, 236, 244, 269

INTEGER option, [44](#)

WITH clause

SOLVE statement, [248](#)

Your Turn

We welcome your feedback.

- If you have comments about this book, please send them to **`yourturn@sas.com`**. Include the full title and page numbers (if applicable).
- If you have comments about the software, please send them to **`suggest@sas.com`**.

SAS® Publishing Delivers!

Whether you are new to the work force or an experienced professional, you need to distinguish yourself in this rapidly changing and competitive job market. SAS® Publishing provides you with a wide range of resources to help you set yourself apart. Visit us online at support.sas.com/bookstore.

SAS® Press

Need to learn the basics? Struggling with a programming problem? You'll find the expert answers that you need in example-rich books from SAS Press. Written by experienced SAS professionals from around the world, SAS Press books deliver real-world insights on a broad range of topics for all skill levels.

support.sas.com/saspress

SAS® Documentation

To successfully implement applications using SAS software, companies in every industry and on every continent all turn to the one source for accurate, timely, and reliable information: SAS documentation. We currently produce the following types of reference documentation to improve your work experience:

- Online help that is built into the software.
- Tutorials that are integrated into the product.
- Reference documentation delivered in HTML and PDF – **free** on the Web.
- Hard-copy books.

support.sas.com/publishing

SAS® Publishing News

Subscribe to SAS Publishing News to receive up-to-date information about all new SAS titles, author podcasts, and new Web site features via e-mail. Complete instructions on how to subscribe, as well as access to past issues, are available at our Web site.

support.sas.com/spn



**THE
POWER
TO KNOW®**

