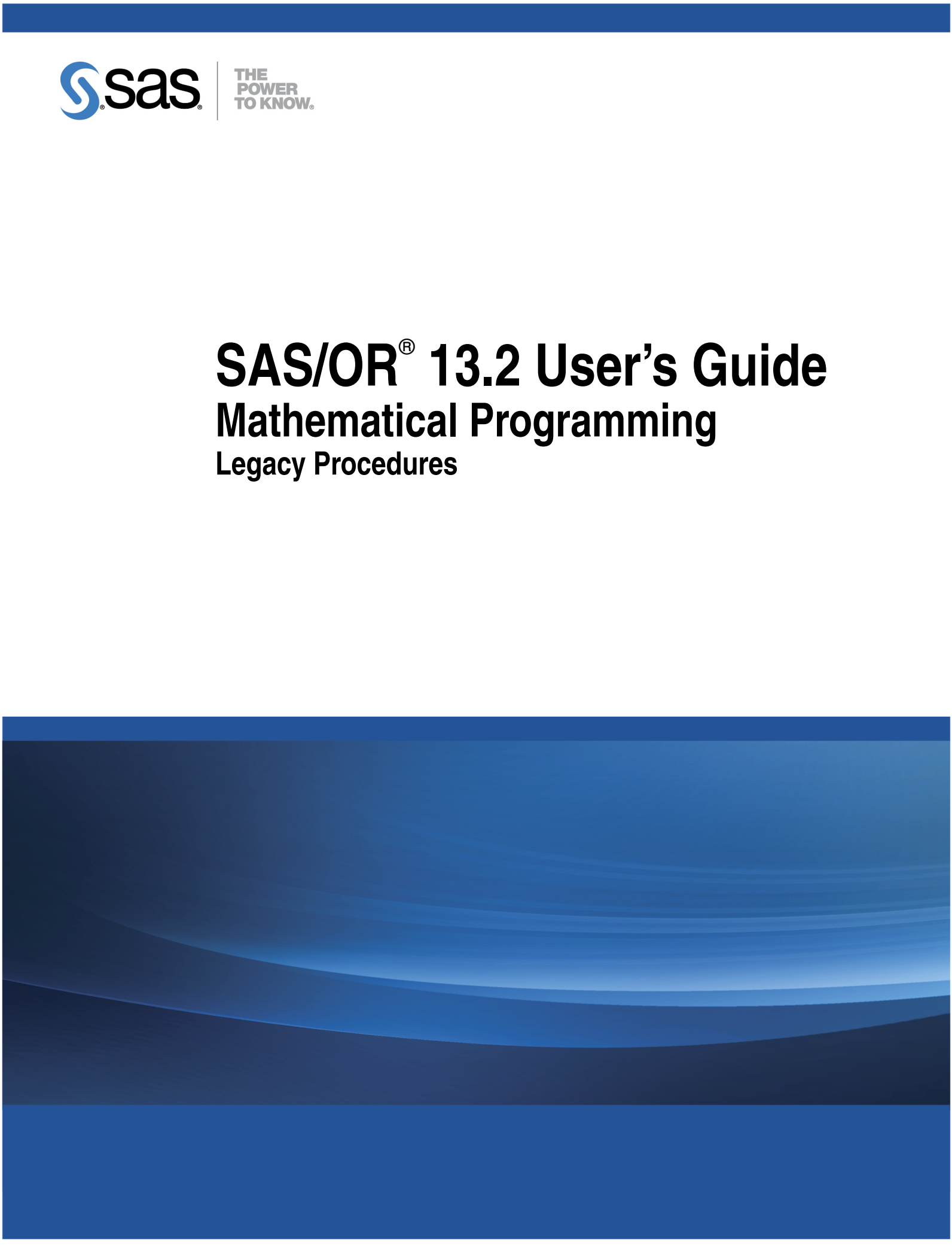


SAS/OR® 13.2 User's Guide

Mathematical Programming

Legacy Procedures

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2014. *SAS/OR® 13.2 User's Guide: Mathematical Programming Legacy Procedures*. Cary, NC: SAS Institute Inc.

SAS/OR® 13.2 User's Guide: Mathematical Programming Legacy Procedures

Copyright © 2014, SAS Institute Inc., Cary, NC, USA

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.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

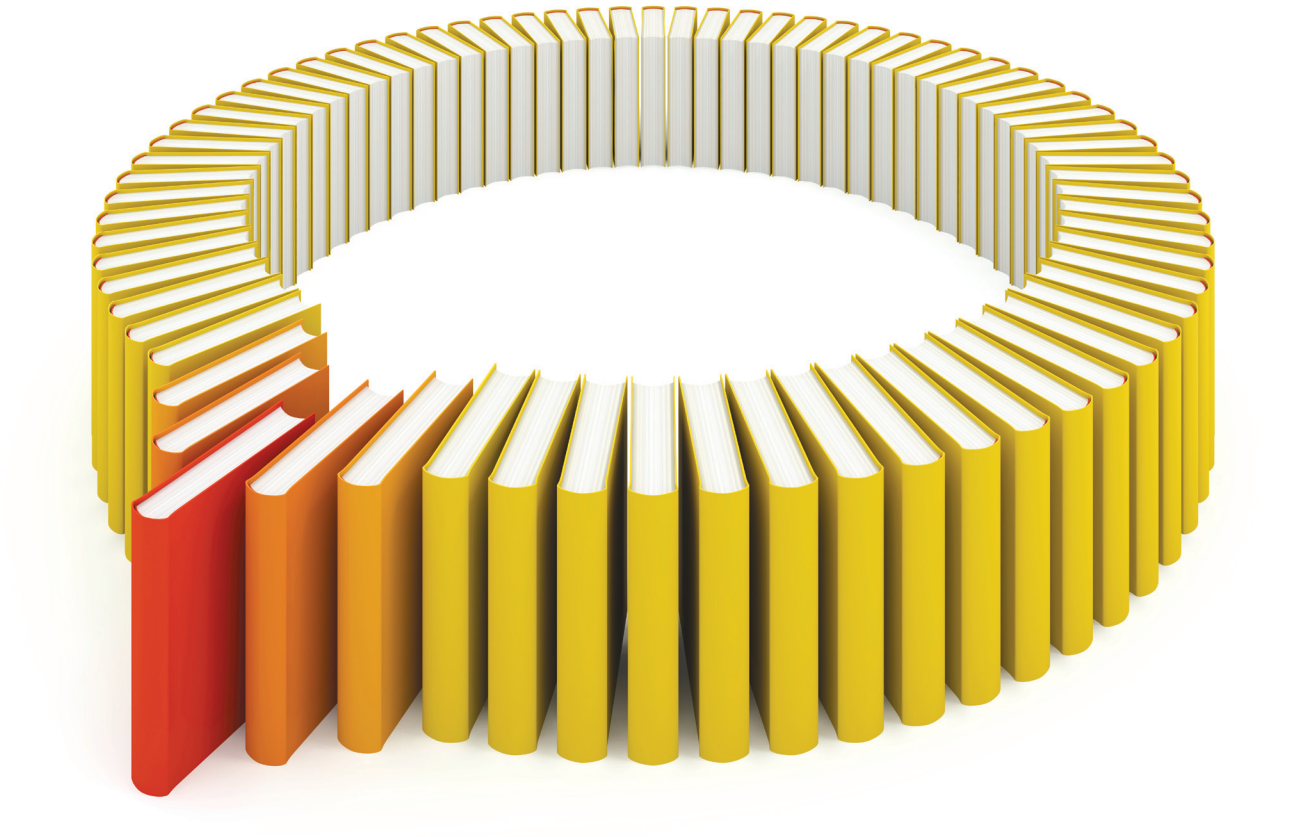
SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

August 2014

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our offerings, visit **support.sas.com/bookstore** 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 trademarks of their respective companies.



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

sas
THE POWER TO KNOW®

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 trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613

Contents

Chapter 1. What's New in SAS/OR 13.2 1
Chapter 2. Using This Book 7
Chapter 3. Introduction to Optimization 13
Chapter 4. The INTPOINT Procedure 37
Chapter 5. The LP Procedure 167
Chapter 6. The NETFLOW Procedure 301
Chapter 7. The NLP Procedure 543

Subject Index **685**

Syntax Index **694**

Acknowledgments

Credits

Documentation

Writing	Tao Huang, Ed Hughes, Trevor Kearney, Richard Liu, Michelle Opp, Rob Pratt, Girish Ramachandra, Ben-Hao Wang
Editing	Anne Baxter, Virginia Clark, Ed Huddleston, Donna Sawyer
Documentation Support	Tim Arnold, Natalie Baerlocher, Remya Chandran, Melanie Gratton, Richard Liu, Jianzhe Luo, Michelle Opp, Girish Ramachandra
Technical Review	Tonya Chapman, Donna Fulenwider, Bill Gjertsen, Tao Huang, Edward P. Hughes, John Jasperse, Charles B. Kelly, Radhika Kulkarni, Bengt Pederson, Rob Pratt

Software

The procedures in SAS/OR software were implemented by the Operations Research and Development Department. Substantial support was given to the project by other members of the Analytical Solutions Division. Core Development Division, Display Products Division, Graphics Division, and the Host Systems Division also contributed to this product.

In the following list, the name of the developer(s) currently supporting the procedure is listed.

INTPOINT	Trevor Kearney
LP	Ben-Hao Wang
NETFLOW	Trevor Kearney
NLP	Tao Huang

Support Groups

Software Testing	Bengt Pederson, Rob Pratt, Lois Zhu
Technical Support	Tonya Chapman

Acknowledgments

Many people have been instrumental in the development of SAS/OR software. The individuals acknowledged here have been especially helpful.

Richard Brockmeier	Union Electric Company
Ken Cruthers	Goodyear Tire & Rubber Company
Patricia Duffy	Auburn University
Richard A. Ehrhardt	University of North Carolina at Greensboro
Paul Hardy	Babcock & Wilcox
Don Henderson	ORI Consulting Group
Dave Jennings	Lockheed Martin
Vidyadhar G. Kulkarni	University of North Carolina at Chapel Hill
Wayne Maruska	Basin Electric Power Cooperative
Roger Perala	United Sugars Corporation
Bruce Reed	Auburn University
Charles Rissmiller	Lockheed Martin
David Rubin	University of North Carolina at Chapel Hill
John Stone	North Carolina State University
Keith R. Weiss	ICI Americas Inc.

The final responsibility for the SAS System lies with SAS Institute alone. We hope that you will always let us know your opinions about the SAS System and its documentation. It is through your participation that SAS software is continuously improved.

Chapter 1
What's New in SAS/OR 13.2

Contents

Overview **1**
Optimization Updates **2**
Solver Performance Improvements 2
PROC OPTMODEL: The CLP Solver (Experimental) 2
Covariance Matrix Output for Nonlinear Optimization 3
The DECOMP Algorithm: Block Detection 3
Changes in Status and Default Settings 4
SAS Simulation Studio 13.2 . . . . . **4**

Overview

SAS/OR 13.2 adds new optimization features that are designed to provide direct access to a wider range of algorithms, improve current methods, and enhance output. Highlights include:

- Several solvers improve their performance.
- PROC OPTMODEL adds the experimental constraint logic programming (CLP) solver.
- The nonlinear programming (NLP) solver adds output of the covariance matrix for the decision variables (parameter estimates).
- The decomposition (DECOMP) algorithm adds new block detection features.
- For the linear programming (LP) interior point solver, the crossover algorithm is applied by default.
- The network solver is at production status.
- The parallel implementation of the mixed integer linear programming (MILP) solver is at production status.

SAS Simulation Studio 13.2, a component of SAS/OR 13.2 for Windows environments, improves performance and adds controls on numeric precision and saved data. Highlights include:

- A new optional parallel mode executes design points and replications simultaneously on multiple computational cores.

- The new Data Trimmer block provides a central facility to control data collection. You can clear (or reset) collected data for all selected data collection blocks at one or more chosen times during the execution of the model.
- The Formula block adds a control on the precision of the numeric values that it produces.

Optimization Updates

Solver Performance Improvements

Several SAS/OR optimization solvers have improved their performance significantly from SAS/OR 13.1 to SAS/OR 13.2. The MILP solver is, on average, 75% faster in SAS/OR 13.2. For problems that require more than 1 second to solve, the MILP solver is 85% faster, and for problems that require more than 10 seconds to solve, it is 95% faster.

For the NLP solver, on average the active-set algorithm is 25% faster and the interior point algorithm is 12% faster in SAS/OR 13.2. For the LP solver, the interior point algorithm is 11% faster.

PROC OPTMODEL: The CLP Solver (Experimental)

PROC CLP for constraint logic programming has long been a valuable component of SAS/OR, providing methods for modeling and solving optimization problems and constraint satisfaction problems that often feature intricate logical constraints. In SAS/OR 13.2, these methods are also accessible from PROC OPTMODEL when you use the SOLVE WITH CLP statement. This enables you to include constraint programming as a component of a larger optimization process, in conjunction with other optimization modeling and solution methods.

The CLP solver can be used to find one solution, all solutions, or up to a specified number of solutions. In addition to linear constraints, the CLP solver supports the following constraint classes:

- All-different constraint: requires that among the specified set of variables, no two variables can take the same value. A single all-different constraint can replace a large number of binary not-equal-to constraints.
- Element constraint: creates a dependency between two variables, in which the value of the first variable indicates which value (from a specified list) the second variable takes. This dependency might correspond to a relationship, functional or otherwise, between the variables.
- Global cardinality constraint (GCC): specifies the minimum and maximum number of times each of a group of values can be assigned among a group of variables. One GCC statement can define several global cardinality constraints, each of which compactly expresses a relationship between a specific value and multiple decision variables. You can also use global cardinality constraints to define disjoint variable domains, by barring a set of variables from taking specified values in the domain (setting both the minimum and the maximum count for each barred value to zero).

- Lexicographic constraint: requires one array to be lexicographically less than (or less than or equal to) a second array of the same size. Lexicographic ordering, a generalization of alphabetical ordering to encompass numeric values, can be very useful in eliminating certain types of symmetry that can arise among solutions. Imposing a lexicographic ordering eliminates many of the mutually symmetric solutions, reducing the number of permissible solutions to the problem and in turn shortening the solution process.
- Pack constraint: defines a bin-packing constraint, in which items that have specified sizes must be placed in bins that have specified capacities. A number of real-world restrictions that are associated with resource allocation and consumption can be modeled using bin-packing constraints.
- Reify constraint: associates a binary control variable with a specified linear constraint. The control variable takes the value 1 if the constraint is satisfied and 0 otherwise. The importance of binary control variables in optimization modeling is well established.

Covariance Matrix Output for Nonlinear Optimization

Users of the legacy NLP procedure in SAS/OR include statisticians who use its nonlinear optimization capabilities to, for example, perform customized nonlinear regressions. In this context, PROC NLP's output of a covariance matrix of the decision variables (or parameter estimates) is essential. Other PROC NLP users also rely on its covariance matrix output. In SAS/OR 13.2, the NLP solver, accessible from PROC OPTMODEL, adds covariance matrix output to better enable these statistical SAS/OR users to take advantage of its more modern nonlinear optimization methods.

The NLP solver provides all six types of central-difference approximations of the covariance matrix that are provided by PROC NLP, with the same calculation adjustments for problems that have minimization or least squares objective functions. The options that control the calculation of the covariance matrix correspond to the options in PROC NLP, and, for the NLP solver, they appear as suboptions in the COVEST= option in the SOLVE statement, which requests that the NLP solver produce a covariance matrix. The COVOUT= suboption enables you to specify a parameter in PROC OPTMODEL that contains the covariance matrix. You can use this parameter in a CREATE DATA statement in PROC OPTMODEL to output the covariance matrix to a SAS data set.

The DECOMP Algorithm: Block Detection

The DECOMP algorithm, introduced in SAS/OR 12.1, enables you to exploit two types of special structure in a linear or mixed integer linear optimization problem. Block-angular structure occurs when the overall problem can be broken into a set of disjoint subproblems that contain mutually exclusive decision variables and constraints, along with a comparatively small set of spanning constraints that involve all or most of the decision variables in the problem (spanning across the subproblems). Block-diagonal structure occurs when the problem can be broken down completely into disjoint subproblems, with no spanning constraints. After the blocks of constraints that correspond to the subproblems have been identified, the DECOMP algorithm automatically coordinates the solution process for the subproblems and for the overall problem, often greatly reducing solution time.

SAS/OR 13.2 adds to the methods available for the critical task of identifying blocks in the constraint matrix and makes it easier to identify these blocks automatically. The METHOD= option in the DECOMP statement

specifies how blocks are identified. In SAS/OR 13.2, the METHOD=AUTO option, whose algorithm formerly permitted the automatic identification only of block-diagonal structure, invokes a new algorithm that can also identify block-angular structure. This enables you to use METHOD=AUTO to identify a much broader range of problem structures suitable for decomposition. You can still invoke the algorithm that formerly corresponded to METHOD=AUTO by specifying METHOD=CONCOMP.

Changes in Status and Default Settings

The crossover algorithm for the interior point LP solver was added in SAS/OR 9.3 and promoted to production status in SAS/OR 13.1. This algorithm converts a solution that is found by the interior point solver to an optimal basic solution, which is easier to interpret and implement. In SAS/OR 13.2, the crossover algorithm is used by default with the interior point LP solver.

The network solver, introduced in SAS/OR 12.1, provides direct access from PROC OPTMODEL to a set of 11 network diagnostic and optimization algorithms. This feature enables you to embed network analysis and optimization into larger solution processes; it is at production status in SAS/OR 13.2.

The parallel implementation of the MILP solver was introduced in SAS/OR 13.1. It enables optimization to be performed largely in parallel on multiple computational cores on a single machine, and it can significantly reduce solution time. On average, the parallel implementation on four cores is twice as fast as the serial implementation on a single core. This feature is at production status in SAS/OR 13.2.

SAS Simulation Studio 13.2

SAS Simulation Studio 13.2 provides a graphical environment for building and working with discrete-event simulation models. Its most prominent new feature is the optional parallel mode for executing a simulation model. In parallel mode, replications that correspond to selected design points are executed in parallel on multiple computational cores on a single machine. You select parallel mode by choosing the **Parallel Mode** option from the **Run** menu or by clicking the Parallel Mode icon on the toolbar. You can specify the maximum number of cores to use in parallel mode in the SAS Simulation Studio Configuration dialog box.

In parallel mode, the **Start**, **Augment**, **Pause**, and **Reset** options on the toolbar and on the **Run** menu operate just as they do otherwise. Certain run-time features, however, are disabled: the simulation clock and replication counter, animation, interactive graphics, and trace messages. These capabilities are usually used for debugging purposes, and the parallel mode is intended to reduce execution time for a completed, debugged model. Log messages are still produced; they now indicate the relevant design point and replication. A progress bar displays the percentage of replications (across all selected design points) that have completed execution. Currently executing design points are highlighted in red in the Experiment window.

SAS Simulation Studio 13.2 also adds the Data Trimmer block, which helps you centrally manage the saving of simulated data. You can use the Data Trimmer block to notify any data collection block in the same simulation model (including those located in compound blocks or submodels) to clear its accumulated data at a specified point during the model run. Multiple blocks can be selected in the Data Trimmer block dialog box and notified simultaneously. An input port on the Data Trimmer block receives a Boolean signal; a true Boolean value causes the Data Trimmer block to notify all its selected blocks to clear their collected data. You can use the Data Trimmer block to, for example, trim data that accumulate during a nonstationary

start-up period for the model, so that the simulated data that you save correspond to stationary operation of the system being modeled. The accumulated data can be in the form of individual data values or statistics that are calculated from simulated data.

Finally, in SAS Simulation Studio 13.2, the Formula block adds a control on the precision of its output numeric values. This is useful if the values that a Formula block produces need to be, for example, integers or two-digit decimals, because of how the values are interpreted and used elsewhere in the model. The integer value that you enter in the **Result Precision** field in the Formula block properties dialog box specifies the decimal precision of the numeric values that are produced. A value of zero for the **Result Precision** field rounds the numeric results to the nearest integer value, whereas a positive value rounds to the specified number of digits to the left of the decimal point, and a negative value rounds to the specified number of digits to the right of the decimal point.

Chapter 2
Using This Book

Contents

Purpose **7**
Organization **7**
Typographical Conventions **9**
Conventions for Examples **9**
Accessing the SAS/OR Sample Library 10
Online Documentation 10
Additional Documentation for SAS/OR Software **10**

Purpose

SAS/OR User's Guide: Mathematical Programming Legacy Procedures provides a complete reference for the older mathematical programming procedures in SAS/OR software. This book serves as the primary documentation for the INTPOINT, LP, NETFLOW, and NLP procedures. The OPTLP, OPTMILP, OPTMODEL, and OPTQP procedures, the various solvers used by PROC OPTMODEL, and the MPS-format SAS data set specification are documented in *SAS/OR User's Guide: Mathematical Programming*.

This chapter describes the organization of this book and the conventions that are used in the text and example code. To gain full benefit from using this book, you should familiarize yourself with the information presented in this section and refer to it when needed. The section "Additional Documentation for SAS/OR Software" on page 10 refers to other documents that contain related information.

Organization

Chapter 3, "Introduction to Optimization," contains a brief overview of the legacy mathematical programming procedures in SAS/OR software and highlights the need to transition in the near future from these legacy procedures to the newer OPTMODEL family of SAS/OR mathematical programming procedures. This chapter also provides an introduction to optimization and the use of the legacy optimization tools in the SAS System; it describes how to build input SAS data sets (directly or by synthesizing multiple data sources), use the legacy procedures to identify optimal solutions, and construct reports to communicate optimal decisions. Following this introductory chapter, the next four chapters describe the LP, INTPOINT, NETFLOW, and NLP procedures.

The following list summarizes the types of information provided for each procedure:

Overview provides a general description of what the procedure does. It outlines major capabilities of the procedure and lists all input and output data sets that are used with it.

Getting Started illustrates simple uses of the procedure in a few short examples. It provides introductory hands-on information for the procedure.

Syntax constitutes the major reference section for the syntax of the procedure. First, the statement syntax is summarized. Next, a functional summary table lists all the statements and options in the procedure, classified by function. In addition, the online version includes a Dictionary of Options, which provides an alphabetical list of all options. Following these tables, the PROC statement is described, and then all other statements are described in alphabetical order.

Details describes the features of the procedure, including algorithmic details and computational methods. It also explains how the various options interact with each other. This section describes input and output data sets in greater detail, with definitions of the output variables, and explains the format of printed output, if any.

Examples consists of examples that are designed to illustrate the use of the procedure. Each example includes a description of the problem and lists the options that are highlighted by the example. The example shows the data and the SAS statements needed, and includes the output that is produced. You can duplicate the examples by copying the statements and data and running the SAS program. The SAS Sample Library contains the code that is used to run the examples shown in this book; consult your SAS Software representative for specific information about the Sample Library.

References lists references that are relevant to the chapter.

Typographical Conventions

This book uses various type styles, as explained by the following list:

roman	is the standard type style used for most text.
UPPERCASE ROMAN	is used for SAS statements, options, and other SAS language elements when they appear in the text. However, you can enter these elements in your own SAS code in lowercase, uppercase, or a mixture of the two. This style is also used for identifying arguments and values (in the syntax specifications) that are literals (for example, to denote valid keywords for a specific option).
UPPERCASE BOLD	is used in the "Syntax" section to identify SAS keywords, such as the names of procedures, statements, and options.
VariableName	is used for the names of SAS variables and data sets when they appear in the text.
oblique	is used to indicate an option variable for which you must supply a value (for example, DUPLICATE=*dup* indicates that you must supply a value for *dup*).
italic	is used for terms that are defined in the text, for emphasis, and for publication titles.
monospace	is used to show examples of SAS statements. In most cases, this book uses lowercase type for SAS code. You can enter your own SAS code in lowercase, uppercase, or a mixture of the two.

Conventions for Examples

Most of the output shown in this book is produced with the following SAS System options:

```
options linesize=80 pagesize=60 nonumber nodate;
```

Accessing the SAS/OR Sample Library

The SAS/OR Sample Library includes many examples that illustrate the use of SAS/OR software, including the examples used in this documentation. To access these sample programs from the SAS windowing environment, select **Help** from the main menu and then select **Getting Started with SAS Software**. On the **Contents** tab, expand the **Learning to Use SAS**, **Sample SAS Programs**, and **SAS/OR** items. Then click **Samples**.

Online Documentation

This documentation is available online with the SAS System. To access SAS/OR documentation from the SAS windowing environment, select **Help** from the main menu and then select **SAS Help and Documentation**. On the **Contents** tab, expand the **SAS Products** and **SAS/OR** items. Then expand the book you want to view. You can search the documentation by using the **Search** tab.

You can also access the documentation by going to `http://support.sas.com/documentation`.

Additional Documentation for SAS/OR Software

In addition to *SAS/OR User's Guide: Mathematical Programming Legacy Procedures,* you might find the following documents helpful when using SAS/OR software:

SAS/OR User's Guide: Bill of Material Processing
: provides documentation for the BOM procedure and all bill of material postprocessing SAS macros. The BOM procedure and SAS macros enable you to generate different reports and to perform several transactions to maintain and update bills of material.

SAS/OR User's Guide: Constraint Programming
: provides documentation for the constraint programming procedure in SAS/OR software. This book serves as the primary documentation for the CLP procedure.

SAS/OR User's Guide: Local Search Optimization
: provides documentation for the local search optimization procedures in SAS/OR software. This book serves as the primary documentation for the GA procedure, which uses genetic algorithms to solve optimization problems, and the OPTLSO procedure, which performs parallel hybrid derivative-free optimization.

SAS/OR User's Guide: Mathematical Programming
: provides documentation for the mathematical programming procedures in SAS/OR software. This book serves as the primary documentation for the OPTLP, OPTMILP, OPTMODEL, and OPTQP procedures, the various solvers called by the OPTMODEL procedure, and the MPS-format SAS data set specification.

SAS/OR User's Guide: Mathematical Programming Examples

supplements the *SAS/OR User's Guide: Mathematical Programming* with additional examples that demonstrate best practices for building and solving linear programming, mixed integer linear programming, and quadratic programming problems. The problem statements are reproduced with permission from the book *Model Building in Mathematical Programming* by H. Paul Williams.

SAS/OR User's Guide: Network Optimization Algorithms

provides documentation for a set of algorithms that can be used to investigate the characteristics of networks and to solve network-oriented optimization problems. This book also documents PROC OPTNET, which invokes these algorithms and provides network-structured formats for input and output data.

SAS/OR User's Guide: Project Management

provides documentation for the project management procedures in SAS/OR software. This book serves as the primary documentation for the CPM, DTREE, GANTT, NETDRAW, and PM procedures, in addition to the PROJMAN Application, a graphical user interface for project management.

SAS Simulation Studio: User's Guide

provides documentation about using SAS Simulation Studio, a graphical application for creating and working with discrete-event simulation models. This book describes in detail how to build and run simulation models and how to interact with SAS software for analysis and with JMP software for experimental design and analysis.

Chapter 3
Introduction to Optimization

Contents

Important Information Regarding the Legacy Procedures . . . **13**
Migrating from the Legacy Procedures . . . 14
Overview . . . **14**
Linear Programming Problems . . . **16**
PROC LP . . . 16
PROC INTPOINT . . . 16
Mixed Integer Linear Programming Problems . . . **17**
PROC LP . . . 17
Network Problems . . . **17**
PROC NETFLOW . . . 17
Nonlinear Problems . . . **18**
The NLP Procedure . . . 18
Model Building . . . **18**
The LP Procedure . . . 18
The NETFLOW Procedure . . . 24
Matrix Generation . . . **27**
Exploiting Model Structure . . . **30**
Report Writing . . . **33**
The DATA Step . . . 33
Other Reporting Procedures . . . 34

Important Information Regarding the Legacy Procedures

This book documents four of the older SAS/OR mathematical programming procedures: the LP, INTPOINT, NETFLOW, and NLP procedures. These procedures are collectively referred to as the "legacy mathematical programming procedures." All of these procedures and their attendant mathematical optimization solvers were developed several years before the introduction of the newer OPTMODEL family of procedures and their corresponding linear, mixed integer linear, quadratic, and general nonlinear optimization solvers. These newer optimization procedures and solvers deliver significant improvements on the older procedures and solvers in several areas:

- clarity and flexibility in optimization modeling, including more versatile use of input data

- flexibility in tailoring the solution process to the model, synthesizing standard and customized optimization methods as needed
- scalability and speed in the provided linear, mixed integer linear, quadratic, and general nonlinear optimization solvers

For these reasons the focus of research and development for mathematical optimization in SAS/OR software has shifted to the newer procedures and solvers. The LP, INTPOINT, NETFLOW, and NLP procedures are no longer under active development although these legacy procedures are still fully supported. At some point in the near future, support for the legacy procedures will be discontinued. Accordingly all SAS/OR users are urged to plan now to migrate from these older procedures to the corresponding newer SAS/OR optimization procedures:

- for linear optimization, from the LP procedure to the OPTLP or OPTMODEL procedure
- for mixed integer linear optimization, from the LP procedure to the OPTMILP or OPTMODEL procedure
- from the NETFLOW procedure to the OPTLP or OPTMODEL procedure
- from the NLP procedure to the OPTMODEL procedure
- from the INTPOINT procedure to the OPTLP or OPTMODEL procedure

Migrating from the Legacy Procedures

The LP, INTPOINT, and NETFLOW procedures have recently added the MPSOUT= option which bypasses the procedure's solver and produces a SAS data set in a mathematical programming system (MPS) format that describes the problem that was to be solved with the older procedure. This data set can be supplied as an input data set to the OPTLP or OPTMILP procedure. It is preferable to model the problem using the algebraic modeling capabilities of PROC OPTMODEL, thus producing highly readable SAS code that transparently depicts the algebraic structure of the optimization model. You can find examples that demonstrate how to model linear and mixed integer linear problems in OPTMODEL in Chapter 7, "The Linear Programming Solver" (*SAS/OR User's Guide: Mathematical Programming*) and Chapter 8, "The Mixed Integer Linear Programming Solver" (*SAS/OR User's Guide: Mathematical Programming*).

Migration from the NLP procedure to the OPTMODEL procedure is not as simple as migration for linear optimization because of the more broadly defined structure of nonlinear optimization models. A brief guide to migration from PROC NLP to PROC OPTMODEL is included in the section "Rewriting NLP Models for PROC OPTMODEL" on page 634; it includes several examples.

Overview

Operations research tools are directed toward the solution of resource management and planning problems. Models in operations research are representations of the structure of a physical object or a conceptual or business process. Using the tools of operations research involves the following:

- defining a structural model of the system under investigation
- collecting the data for the model
- solving the model
- interpreting the results

SAS/OR software is a set of procedures for exploring models of distribution networks, production systems, resource allocation problems, and scheduling problems using the tools of operations research.

The following list suggests some of the application areas in which optimization-based decision support systems have been used. In practice, models often contain elements of several applications listed here.

- **Product-mix problems** find the mix of products that generates the largest return when several products compete for limited resources.
- **Blending problems** find the mix of ingredients to be used in a product so that it meets minimum standards at minimum cost.
- **Time-staged problems** are models whose structure repeats as a function of time. Production and inventory models are classic examples of time-staged problems. In each period, production plus inventory minus current demand equals inventory carried to the next period.
- **Scheduling problems** assign people to times, places, or tasks so as to optimize people's preferences or performance while satisfying the demands of the schedule.
- **Multiple objective problems** have multiple, possibly conflicting, objectives. Typically, the objectives are prioritized and the problems are solved sequentially in a priority order.
- **Capital budgeting and project selection problems** ask for the project or set of projects that yield the greatest return.
- **Location problems** seek the set of locations that meets the distribution needs at minimum cost.
- **Cutting stock problems** find the partition of raw material that minimizes waste and fulfills demand.

A problem is formalized with the construction of a model to represent it. These models, called mathematical programs, are represented in SAS data sets when they are to be solved using the legacy SAS/OR procedures. The solution of mathematical programs is called mathematical programming. Since mathematical programs are represented in SAS data sets, they can be saved, changed, and solved again. The legacy SAS/OR procedures also output SAS data sets that contain the solutions. These output data sets can then be used to produce customized reports. In addition, this structure enables you to build decision support systems by using operations research and other analytic tools in the SAS System as building blocks. All of these steps are even easier and solution of the mathematical programs is more efficient when you use PROC OPTMODEL and the rest of the newer SAS/OR mathematical programming procedures.

The basic optimization problem is that of minimizing or maximizing an objective function subject to constraints imposed on the variables of that function. The objective function and constraints can be linear or nonlinear; the constraints can be bound constraints, equality or inequality constraints, or integer constraints. Traditionally, optimization problems are divided into linear programming (LP; all functions and constraints are linear) and nonlinear programming (NLP).

The data that describe the model are supplied to an optimizer (such as one of the procedures described in this book), an optimizing algorithm is used to determine the optimal values for the decision variables so the objective is either maximized or minimized, the optimal values assigned to decision variables are on or between allowable bounds, and the constraints are obeyed. Determining the optimal values is the process called *optimization*.

This chapter describes how to use the legacy SAS/OR mathematical programming procedures to solve a wide variety of optimization problems. It describes various types of optimization problems, indicates which legacy SAS/OR procedure you can use, and demonstrates how to provide data, run the procedure, and obtain optimal solutions.

The next section broadly classifies the legacy SAS/OR procedures based on the types of mathematical programming problems they can solve.

Linear Programming Problems

PROC LP

The LP procedure solves linear programs with a primal simplex solver. It can perform several types of post-optimality analysis, including range analysis, sensitivity analysis, and parametric programming. The procedure can also be used interactively.

PROC LP requires a problem data set that contains the model. In addition, a primal data set can be used for warm-starting a problem that has been partially solved previously.

The problem data that describe the model can be in one of two formats: dense or sparse. The dense format represents the model as a rectangular coefficient matrix. The sparse format, on the other hand, represents only the nonzero elements of a rectangular coefficient matrix.

For more details about the LP procedure, see Chapter 5, "The LP Procedure."

PROC INTPOINT

The INTPOINT procedure solves linear programming problems by using the interior point algorithm.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to PROC INTPOINT.

For more details about the INTPOINT procedure, see Chapter 4, "The INTPOINT Procedure."

Mixed Integer Linear Programming Problems

PROC LP

The LP procedure solves mixed integer linear programming problems (MILPs) with a primal simplex solver. To solve a MILP, you first need to identify the integer variables. You can do this with a row in the input data set that has the keyword INTEGER for the type variable. It is important to note that integer variables must have upper bounds explicitly defined.

As with linear programs, you can specify MILP problem data using either sparse or dense format. For more details, see Chapter 5, "The LP Procedure."

Network Problems

PROC NETFLOW

The NETFLOW procedure solves network flow problems with linear side constraints by using either a network simplex algorithm or an interior point algorithm.

The Network Simplex Algorithm

PROC NETFLOW's network simplex algorithm solves pure network flow problems and network flow problems with linear side constraints. The procedure accepts the network specification in formats that are particularly suited to networks.

Network flow problems, such as finding the minimum cost flow in a network, benefit from model representation in a format that is specialized for network structures. The network is represented in two data sets: a node data set that names the nodes in the network and gives supply and demand information at them, and an arc data set that uses node names to define the arcs in the network and gives arc costs and capacities. In addition, a side-constraint data set is included that gives any side constraints that apply to the flow through the network. Examples of these are found later in this chapter.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to network flow models that have side constraints.

The Interior Point Algorithm

When you solve a constrained network problem, you can specify the INTPOINT option to indicate that the interior point algorithm is to be used. The input data are the same whether the network simplex or interior point method is used. The interior point method is often faster when problems have many side constraints.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to network flow models that have side constraints. For more information about the NETFLOW procedure, see Chapter 6, "The NETFLOW Procedure."

Nonlinear Problems

The NLP Procedure

The NLP procedure offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function subject to linear and nonlinear, equality and inequality, and lower and upper bound constraints. Problems of this type are found in many settings, ranging from optimal control to maximum likelihood estimation.

Nonlinear programs can be input into the NLP procedure in various ways. The objective, constraint, and derivative functions are specified using the programming statements of PROC NLP. In addition, information in SAS data sets can be used to define the structure of objectives and constraints and to specify constants that are used in objectives, constraints, and derivatives.

PROC NLP uses the following data sets to input various pieces of information:

- The DATA= data set enables you to specify data that is used to define one or more objective functions in a nonlinear programming problem.
- The INQUAD= data set contains the arrays that appear in a quadratic programming problem.
- The INEST= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, and simple boundary and general linear constraints.
- The MODEL= data set specifies a model (functions, constraints, derivatives) saved at a previous execution of the NLP procedure.

As an alternative to supplying data in SAS data sets, some or all data for the model can be specified using SAS programming statements. These statements are similar to the statements used in the SAS DATA step.

For more information about the NLP procedure, refer to Chapter 7, "The NLP Procedure."

Model Building

Model generation and maintenance are often difficult and expensive aspects of applying mathematical programming techniques. The flexible input formats for the optimization procedures in SAS/OR software simplify this task.

The LP Procedure

A small product-mix problem serves as a starting point for a discussion of different types of model formats that are supported in SAS/OR software.

A candy manufacturer makes two products: chocolates and toffee. What combination of chocolates and toffee should be produced in a day in order to maximize the company's profit? Chocolates contribute \$0.25 per pound to profit, and toffee contributes \$0.75 per pound. The decision variables are *chocolates* and *toffee*.

Four processes are used to manufacture the candy:

1. Process 1 combines and cooks the basic ingredients for both chocolates and toffee.
2. Process 2 adds colors and flavors to the toffee, then cools and shapes the confection.
3. Process 3 chops and mixes nuts and raisins, adds them to the chocolates, and then cools and cuts the bars.
4. Process 4 is packaging: chocolates are placed in individual paper shells; toffee is wrapped in cellophane packages.

During the day, there are 7.5 hours (27,000 seconds) available for each process.

Firm time standards have been established for each process. For Process 1, mixing and cooking take 15 seconds for each pound of chocolate, and 40 seconds for each pound of toffee. Process 2 takes 56.25 seconds per pound of toffee. For Process 3, each pound of chocolate requires 18.75 seconds of processing. In packaging, a pound of chocolates can be wrapped in 12 seconds, whereas a pound of toffee requires 50 seconds. These data are summarized as follows:

		Available Time	Required per Pound	
	Process	(sec)	chocolates (sec)	toffee (sec)
1	Cooking	27,000	15	40
2	Color/Flavor	27,000		56.25
3	Condiments	27,000	18.75	
4	Packaging	27,000	12	50

The objective is to maximize the company's total profit, which is represented as:

Maximize: $0.25(\textit{chocolates}) + 0.75(\textit{toffee})$

The production of the candy is limited by the time available for each process. The limits placed on production by Process 1 are expressed by the following inequality:

Process 1: $15(\textit{chocolates}) + 40(\textit{toffee}) \leq 27{,}000$

Process 1 can handle any combination of chocolates and toffee that satisfies this inequality.

The limits on production by other processes generate constraints described by the following inequalities:

Process 2: $56.25(\textit{toffee}) \leq 27{,}000$

Process 3: $18.75(\textit{chocolates}) \leq 27{,}000$

Process 4: $12(\textit{chocolates}) + 50(\textit{toffee}) \leq 27{,}000$

This sample linear program illustrates a product mix problem. The mix of products that maximizes the objective without violating the constraints is the solution. For input to PROC LP, this model can be represented in a SAS data set that uses either dense or sparse format.

Dense Format

The following DATA step creates a SAS data set for this product mix problem. Notice that the values of CHOCO and TOFFEE in the data set are the coefficients of those variables in the objective function and constraints. The variable _id_ contains a character string that names the rows in the data set. The variable _type_ is a character variable that contains keywords that describe the type of each row in the problem data set. The variable _rhs_ contains the right-hand-side values.

```
data factory;
   input _id_ $ CHOCO TOFFEE _type_ $ _rhs_;
   datalines;
object      0.25    0.75   MAX     .
process1   15.00   40.00   LE  27000
process2    0.00   56.25   LE  27000
process3   18.75    0.00   LE  27000
process4   12.00   50.00   LE  27000
;
```

To solve this problem by using PROC LP, specify the following:

```
proc lp data=factory;
run;
```

Sparse Format

Typically, mathematical programming models are sparse; that is, few of the coefficients in the constraint matrix are nonzero. For this reason the LP procedure also accepts data in a sparse format SAS data set. In this format only the nonzero coefficients for the constraints and objective function must be specified.

An example of a sparse format data set is shown here. The following data set contains the data from the product mix problem of the preceding section.

```
data sp_factory;
   format _type_ $8. _row_ $10. _col_ $10.;
   input _type_ $_row_ $ _col_ $ _coef_;
   datalines;
max      object     .            .
.        object     chocolate    .25
.        object     toffee       .75
le       process1   .            .
.        process1   chocolate    15
.        process1   toffee       40
.        process1   _RHS_        27000
le       process2   .            .
.        process2   toffee       56.25
.        process2   _RHS_        27000
le       process3   .            .
.        process3   chocolate    18.75
.        process3   _RHS_        27000
le       process4   .            .
.        process4   chocolate    12
.        process4   toffee       50
.        process4   _RHS_        27000
;
```

To solve this problem by using PROC LP, specify the following:

```
proc lp data=sp_factory sparsedata;
run;
```

The "Solution Summary" table (shown in Figure 3.1) contains information about the solution that was found, including whether the optimizer terminated successfully after finding the optimum.

PROC LP uses an iterative process to solve problems. First, the procedure finds a feasible solution that satisfies the constraints. Second, it finds an optimal solution from the set of feasible solutions. The "Solution Summary" table lists information about the optimization process such as the number of iterations, the infeasibilities of the solution, and the time required to solve the problem.

Figure 3.1 Solution Summary

The LP Procedure

Solution Summary	
Terminated Successfully	
Objective Value	475
Phase 1 Iterations	0
Phase 2 Iterations	3
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	6
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Separating the Data from the Model Structure

It is often desirable to keep the data separate from the structure of the model. This is useful for large models with numerous identifiable components. The data are best organized in rectangular tables that can be easily examined and modified. Then, before the problem is solved, the model is built using the stored data. This process of model building is known as *matrix generation*. In conjunction with the sparse format used by the legacy procedures, the SAS DATA step provides a good matrix generation language.

For example, consider the candy manufacturing example introduced previously. Suppose that, for the user interface, it is more convenient to organize the data so that each record describes the information related to each product (namely, the contribution to the objective function and the unit amount needed for each process). A DATA step for saving the data might look like this:

```
data manfg;
   format product $12.;
   input product $ object process1 - process4 ;
   datalines;
chocolate        .25    15  0.00 18.75    12
toffee           .75    40 56.25  0.00    50
licorice        1.00    29 30.00 20.00    20
jelly_beans      .85    10  0.00 30.00    10
_RHS_            .   27000 27000 27000 27000
;
```

Notice that there is a special record at the end that has product _RHS_. This record gives the amounts of time available for each of the processes. This information could have been stored in another data set. The next example illustrates a model in which the data are stored in separate data sets.

Building the model involves adding the data to the structure. There are as many ways to add data as there are programmers and problems. The following DATA step shows one way to use the candy data to build a sparse format model to solve the product mix problem:

```
data model;
   array process object process1-process4;
   format _type_ $8. _row_ $12. _col_ $12. ;
   keep _type_ _row_ _col_ _coef_;

   set manfg;          /* read the manufacturing data */

   /* build the object function */

   if _n_=1 then do;
      _type_='max'; _row_='object'; _col_=' '; _coef_=.;
      output;
   end;

   /* build the constraints */

   do over process;
      if _i_>1 then do;
         _type_='le'; _row_='process'||put(_i_-1,1.);
      end;
      else            _row_='object';
      _col_=product; _coef_=process;
      output;
   end;
run;
```

The sparse format data set is shown in Figure 3.2.

Figure 3.2 Sparse Data Format

Obs	_type_	_row_	_col_	_coef_
1	max	object		.
2	max	object	chocolate	0.25
3	le	process1	chocolate	15.00
4	le	process2	chocolate	0.00
5	le	process3	chocolate	18.75
6	le	process4	chocolate	12.00
7		object	toffee	0.75
8	le	process1	toffee	40.00
9	le	process2	toffee	56.25
10	le	process3	toffee	0.00
11	le	process4	toffee	50.00
12		object	licorice	1.00
13	le	process1	licorice	29.00
14	le	process2	licorice	30.00
15	le	process3	licorice	20.00
16	le	process4	licorice	20.00
17		object	jelly_beans	0.85
18	le	process1	jelly_beans	10.00
19	le	process2	jelly_beans	0.00
20	le	process3	jelly_beans	30.00
21	le	process4	jelly_beans	10.00
22		object	_RHS_	.
23	le	process1	_RHS_	27000.00
24	le	process2	_RHS_	27000.00
25	le	process3	_RHS_	27000.00
26	le	process4	_RHS_	27000.00

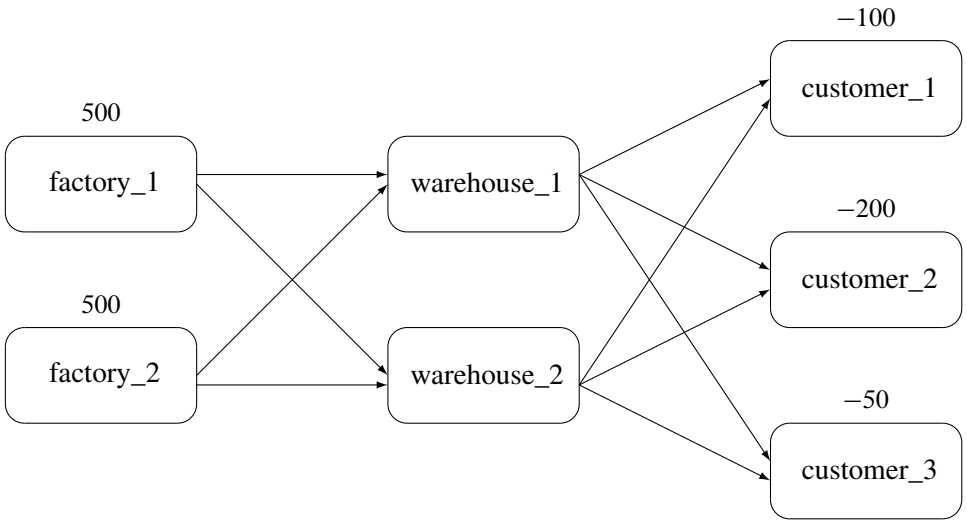
The model data set looks a little different from the sparse representation of the candy model shown earlier. It not only includes additional products (licorice and jelly beans), but it also defines the model in a different order. Since the sparse format is robust, the model can be generated in ways that are convenient for the DATA step program.

If the problem had more products, you could increase the size of the manfg data set to include the new product data. Also, if the problem had more than four processes, you could add the new process variables to the manfg data set and increase the size of the process array in the model data set. With these two simple changes and additional data, a product mix problem that has hundreds of processes and products can be modeled and solved.

The NETFLOW Procedure

Network flow problems can be described by specifying the nodes in the network and their supplies and demands, specifying the arcs in the network and their costs, capacities, and lower flow bounds. Consider the simple transshipment problem in Figure 3.3 as an illustration.

Figure 3.3 Transshipment Problem



Suppose the candy manufacturing company has two factories, two warehouses, and three customers for chocolate. The two factories each have a production capacity of 500 pounds per day. The three customers have demands of 100, 200, and 50 pounds per day, respectively.

The following data set describes the supplies (positive values for the supdem variable) and the demands (negative values for the supdem variable) for each of the customers and factories:

```
data nodes;
   format node $10. ;
   input node $ supdem;
   datalines;
customer_1   -100
customer_2   -200
customer_3    -50
factory_1     500
factory_2     500
;
```

Suppose that two warehouses are used to store the chocolate before shipment to the customers, and that there are different costs for shipping between each factory, warehouse, and customer. What is the minimum cost routing for supplying the customers?

Arcs are described in another data set. Each observation defines a new arc in the network and gives data about the arc. For example, there is an arc between the node factory_1 and the node warehouse_1. Each unit of flow on that arc costs 10. Although this example does not include it, lower and upper bounds on the flow across that arc can also be listed here.

```
data network;
   format from $12. to $12.;
   input from $ to $ cost ;
   datalines;
factory_1     warehouse_1  10
factory_2     warehouse_1   5
factory_1     warehouse_2   7
factory_2     warehouse_2   9
warehouse_1   customer_1    3
warehouse_1   customer_2    4
warehouse_1   customer_3    4
warehouse_2   customer_1    5
warehouse_2   customer_2    5
warehouse_2   customer_3    6
;
```

You can use PROC NETFLOW to find the minimum cost routing. This procedure takes the model as defined in the network and nodes data sets and finds the minimum cost flow.

```
proc netflow arcout=arc_sav
             arcdata=network nodedata=nodes;
   node node;      /* node data set information */
   supdem supdem;
   tail from;      /* arc data set information */
   head to;
   cost cost;
run;

proc print;
   var from to cost _capac_ _lo_ _supply_ _demand_
       _flow_ _fcost_ _rcost_;
   sum _fcost_;
run;
```

PROC NETFLOW produces the following messages in the SAS log:

```
NOTE: Number of nodes= 7 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 3 .
NOTE: Total supply= 1000 , total demand= 350 .
NOTE: Number of arcs= 10 .
NOTE: Number of iterations performed (neglecting any constraints)= 9 .
NOTE: Of these, 2 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 3050 .
NOTE: The data set WORK.ARC_SAV has 10 observations and 13 variables.
```

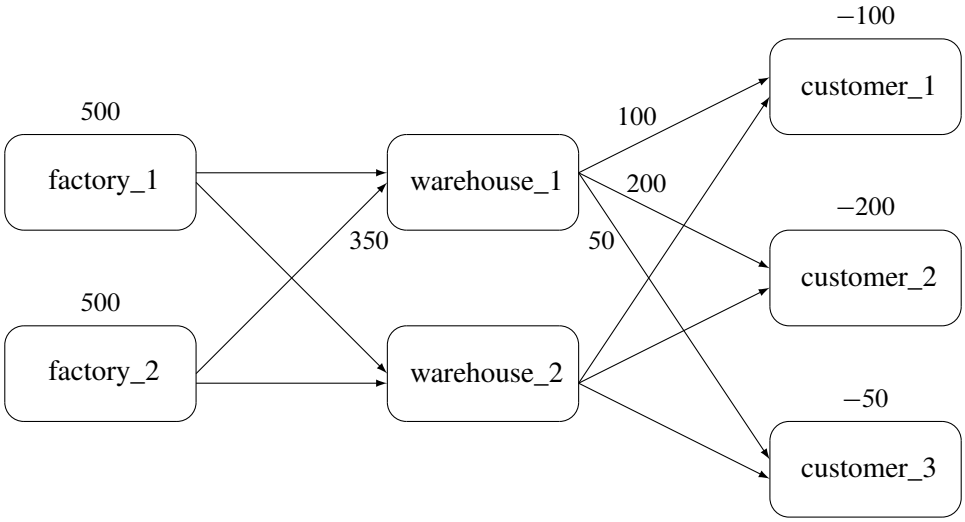
The solution (Figure 3.4) saved in the arc_sav data set shows the optimal amount of chocolate to send across each arc (the amount to ship from each factory to each warehouse and from each warehouse to each customer) in the network per day.

Figure 3.4 ARCOUT Data Set

Obs	from	to	cost	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1	3	99999999	0	.	100	100	300	.
2	warehouse_2	customer_1	5	99999999	0	.	100	0	0	4
3	warehouse_1	customer_2	4	99999999	0	.	200	200	800	.
4	warehouse_2	customer_2	5	99999999	0	.	200	0	0	3
5	warehouse_1	customer_3	4	99999999	0	.	50	50	200	.
6	warehouse_2	customer_3	6	99999999	0	.	50	0	0	4
7	factory_1	warehouse_1	10	99999999	0	500	.	0	0	5
8	factory_2	warehouse_1	5	99999999	0	500	.	350	1750	.
9	factory_1	warehouse_2	7	99999999	0	500	.	0	0	.
10	factory_2	warehouse_2	9	99999999	0	500	.	0	0	2
									3050	

Notice which arcs have positive flow (_FLOW_ is greater than 0). These arcs indicate the amount of chocolate that should be sent from factory_2 to warehouse_1 and from there to the three customers. The solution indicates no production at factory_1 and no use of warehouse_2.

Figure 3.5 Optimal Solution for the Transshipment Problem



Matrix Generation

It is desirable to keep data in separate tables, and then to automate model building and reporting. This example illustrates a problem that has elements of both a product mix problem and a blending problem. Suppose four kinds of ties are made: all silk, all polyester, a 50-50 polyester-cotton blend, and a 70-30 cotton-polyester blend.

The data include cost and supplies of raw material, selling price, minimum contract sales, maximum demand of the finished products, and the proportions of raw materials that go into each product. The objective is to find the product mix that maximizes profit.

The data are saved in three SAS data sets. The following program demonstrates one way for these data to be saved:

```
data material;
   format descpt $20.;
   input descpt $ cost supply;
   datalines;
silk_material              .21   25.8
polyester_material         .6    22.0
cotton_material            .9    13.6
;
```

```
data tie;
   format descpt $20.;
   input descpt $ price contract demand;
   datalines;
all_silk                 6.70    6.0    7.00
all_polyester            3.55   10.0   14.00
poly_cotton_blend        4.31   13.0   16.00
cotton_poly_blend        4.81    6.0    8.50
;

data manfg;
   format descpt $20.;
   input descpt $ silk poly cotton;
   datalines;
all_silk                  100     0      0
all_polyester               0   100      0
poly_cotton_blend           0    50     50
cotton_poly_blend           0    30     70
;
```

The following program takes the raw data from the three data sets and builds a linear program model in the data set called model. Although it is designed for the three-resource, four-product problem described here, it can be extended to include more resources and products. The model-building DATA step remains essentially the same; all that changes are the dimensions of loops and arrays. Of course, the data tables must expand to accommodate the new data. Note that data-driven model creation as outlined here is far easier and more direct with the algebraic modeling capabilities of PROC OPTMODEL, the foremost of the newer SAS/OR mathematical programming procedures.

```
data model;
   array  raw_mat {3} $ 20 ;
   array  raw_comp {3} silk poly cotton;
   length _type_ $ 8  _col_ $ 20  _row_ $ 20 _coef_ 8 ;
   keep   _type_      _col_       _row_      _coef_ ;

   /* define the objective, lower, and upper bound rows */

   _row_='profit'; _type_='max';     output;
   _row_='lower';  _type_='lowerbd'; output;
   _row_='upper';  _type_='upperbd'; output;
   _type_=' ';

   /* the object and upper rows for the raw materials */

   do i=1 to 3;
      set material;
      raw_mat[i]=descpt;  _col_=descpt;
      _row_='profit';    _coef_=-cost;  output;
      _row_='upper';     _coef_=supply; output;
   end;
```

```
   /* the object, upper, and lower rows for the products */

   do i=1 to 4;
      set tie;
      _col_=descpt;
      _row_='profit'; _coef_=price;    output;
      _row_='lower';  _coef_=contract; output;
      _row_='upper';  _coef_=demand;   output;
   end;

   /* the coefficient matrix for manufacturing */

   _type_='eq';
   do i=1 to 4;          /* loop for each raw material */
      set manfg;
      do j=1 to 3;       /* loop for each product      */

         _col_=descpt;   /* % of material in product   */
         _row_  = raw_mat[j];
         _coef_ = raw_comp[j]/100;
         output;

         _col_  = raw_mat[j];  _coef_ = -1;
         output;

         /* the right-hand side */

         if i=1 then do;
            _col_='_RHS_';
            _coef_=0;
            output;
         end;
      end;
      _type_='  ';
   end;
   stop;
run;
```

The model is solved using PROC LP, which saves the solution in the PRIMALOUT data set named solution. PROC PRINT displays the solution, shown in Figure 3.6.

```
proc lp sparsedata primalout=solution;

proc print;
      id _var_;
      var _lbound_--_r_cost_;
run;
```

Figure 3.6 Solution Data Set

The LP Procedure

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	168.708	.
2	silk_material	EQ	.	0	0	0.21
3	polyester_material	EQ	.	0	0	3.55
4	cotton_material	EQ	.	0	0	5.07

VAR	_LBOUND_	_VALUE_	_UBOUND_	_PRICE_	_R_COST_
all_polyester	10	11.800	14.0	3.55	0.000
all_silk	6	7.000	7.0	6.70	6.490
cotton_material	0	13.600	13.6	-0.90	4.170
cotton_poly_blend	6	8.500	8.5	4.81	0.196
polyester_material	0	22.000	22.0	-0.60	2.950
poly_cotton_blend	13	15.300	16.0	4.31	0.000
silk_material	0	7.000	25.8	-0.21	0.000
PHASE_1_OBJECTIVE	0	0.000	0.0	0.00	0.000
profit	0	168.708	1.7977E308	0.00	0.000

The solution shows that 11.8 units of polyester ties, 7 units of silk ties, 8.5 units of the cotton-polyester blend, and 15.3 units of the polyester-cotton blend should be produced. It also shows the amounts of raw materials that go into this product mix to generate a total profit of 168.708.

Exploiting Model Structure

Another example helps to illustrate how the model can be simplified by exploiting the structure in the model when using the NETFLOW procedure.

Recall the chocolate transshipment problem discussed previously. The solution required no production at factory_1 and no storage at warehouse_2. Suppose this solution, although optimal, is unacceptable. An additional constraint that requires the production at the two factories to be balanced is needed. Now, the production at the two factories can differ by, at most, 100 units. Such a constraint might look like this:

```
-100 <= (factory_1_warehouse_1 + factory_1_warehouse_2 -
         factory_2_warehouse_1 - factory_2_warehouse_2) <= 100
```

The network and supply and demand information are saved in the following two data sets:

```
data network;
   format from $12. to $12.;
   input  from $ to $ cost ;
   datalines;
factory_1    warehouse_1  10
factory_2    warehouse_1   5
```

```
factory_1    warehouse_2   7
factory_2    warehouse_2   9
warehouse_1  customer_1   3
warehouse_1  customer_2   4
warehouse_1  customer_3   4
warehouse_2  customer_1   5
warehouse_2  customer_2   5
warehouse_2  customer_3   6
;

data nodes;
   format node $12. ;
   input node $  supdem;
   datalines;
customer_1   -100
customer_2   -200
customer_3    -50
factory_1     500
factory_2     500
;
```

The factory-balancing constraint is not a part of the network. It is represented in the sparse format in a data set for side constraints.

```
data side_con;
   format _type_ $8. _row_ $8. _col_ $21. ;
   input  _type_  _row_  _col_  _coef_ ;
   datalines;
eq       balance   .                        .
.        balance  factory_1_warehouse_1     1
.        balance  factory_1_warehouse_2     1
.        balance  factory_2_warehouse_1    -1
.        balance  factory_2_warehouse_2    -1
.        balance  diff                     -1
lo       lowerbd  diff                   -100
up       upperbd  diff                    100
;
```

This data set contains an equality constraint that sets the value of DIFF to be the amount that factory 1 production exceeds factory 2 production. It also contains bounds on the DIFF variable. Note that the DIFF variable is a nonarc variable.

You can use the following call to PROC NETFLOW to solve the problem:

```
proc netflow
   conout=con_sav
   arcdata=network nodedata=nodes condata=side_con
   sparsecondata ;
   node node;
   supdem supdem;
   tail from;
   head to;
   cost cost;
run;
```

```
proc print;
   var from to _name_ cost _capac_ _lo_ _supply_ _demand_
       _flow_ _fcost_ _rcost_;
   sum _fcost_;
run;
```

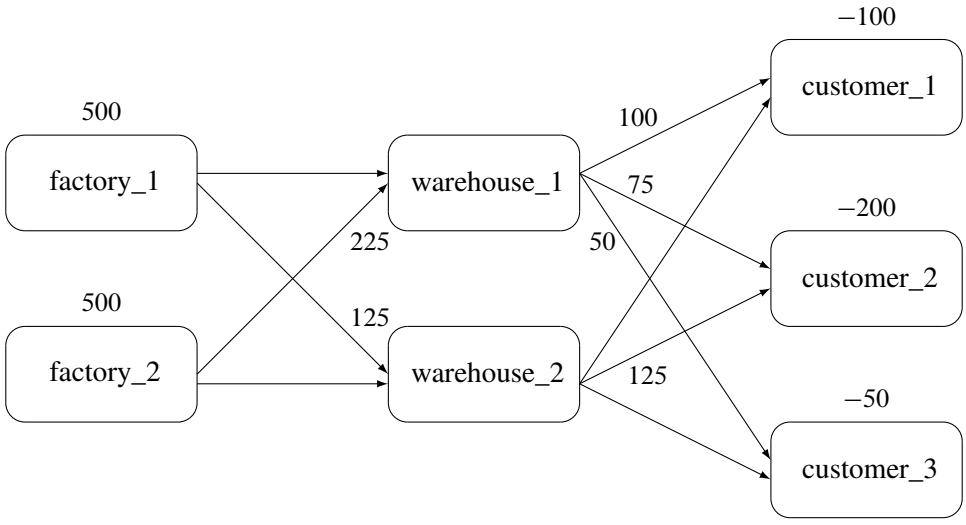
The solution is saved in the con_sav data set, as displayed in Figure 3.7.

Figure 3.7 CON_SAV Data Set

Obs	from	to	_NAME_	cost	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1		3	99999999	0	.	100	100	300	.
2	warehouse_2	customer_1		5	99999999	0	.	100	0	0	1.0
3	warehouse_1	customer_2		4	99999999	0	.	200	75	300	.
4	warehouse_2	customer_2		5	99999999	0	.	200	125	625	.
5	warehouse_1	customer_3		4	99999999	0	.	50	50	200	.
6	warehouse_2	customer_3		6	99999999	0	.	50	0	0	1.0
7	factory_1	warehouse_1		10	99999999	0	500	.	0	0	2.0
8	factory_2	warehouse_1		5	99999999	0	500	.	225	1125	.
9	factory_1	warehouse_2		7	99999999	0	500	.	125	875	.
10	factory_2	warehouse_2		9	99999999	0	500	.	0	0	5.0
11			diff	0	100	-100	.	.	-100	0	1.5
										3425	

Notice that the solution now has production balanced across the factories; the production at factory 2 exceeds that at factory 1 by 100 units.

Figure 3.8 Constrained Optimum for the Transshipment Problem



Report Writing

The reporting of the solution is also an important aspect of modeling. Since the optimization procedures save the solution in one or more SAS data sets, reports can be written using any of the tools in the SAS language.

The DATA Step

Use of the DATA step and PROC PRINT is the most common way to produce reports. For example, from the data set solution shown in Figure 3.6, a table that shows the revenue of the optimal production plan and a table of the cost of material can be produced with the following program:

```
data product(keep= _var_ _value_ _price_ revenue)
     material(keep=_var_ _value_ _price_ cost);
   set solution;
   if _price_>0 then do;
      revenue=_price_*_value_; output product;
   end;
   else if _price_<0 then do;
      _price_=-_price_;
      cost = _price_*_value_; output material;
   end;
run;

/* display the product report */

proc print data=product;
   id _var_;
   var _value_ _price_ revenue ;
   sum revenue;
   title 'Revenue Generated from Tie Sales';
run;

/* display the materials report */

proc print data=material;
   id _var_;
   var _value_ _price_ cost;
   sum cost;
   title 'Cost of Raw Materials';
run;
```

This DATA step reads the solution data set saved by PROC LP and segregates the records based on whether they correspond to materials or products—namely whether the contribution to profit is positive or negative. Each of these is then displayed to produce Figure 3.9.

Figure 3.9 Tie Problem: Revenues and Costs

Revenue Generated from Tie Sales

VAR	_VALUE_	_PRICE_	revenue
all_polyester	11.8	3.55	41.890
all_silk	7.0	6.70	46.900
cotton_poly_blend	8.5	4.81	40.885
poly_cotton_blend	15.3	4.31	65.943
			195.618

Cost of Raw Materials

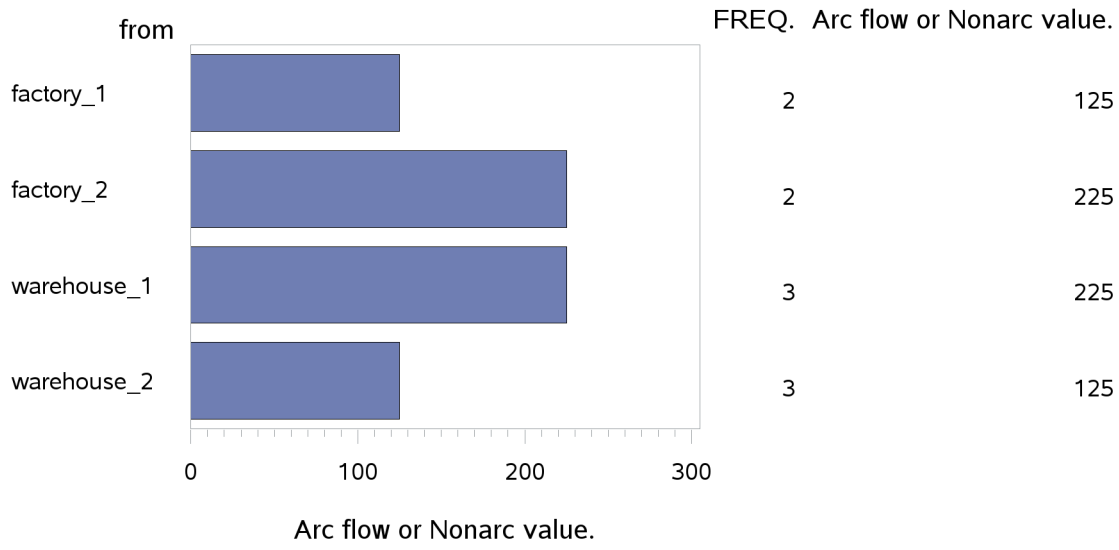
VAR	_VALUE_	_PRICE_	cost
cotton_material	13.6	0.90	12.24
polyester_material	22.0	0.60	13.20
silk_material	7.0	0.21	1.47
			26.91

Other Reporting Procedures

The GCHART procedure can be a useful tool for displaying the solution to mathematical programming models. The con_solv data set that contains the solution to the balanced transshipment problem can be effectively displayed using PROC GCHART. In Figure 3.10, the amount that is shipped from each factory and warehouse can be seen by submitting the following SAS statements:

```
title;
proc gchart data=con_sav;
   hbar from / sumvar=_flow_;
run;
```

Figure 3.10 Tie Problem: Throughputs

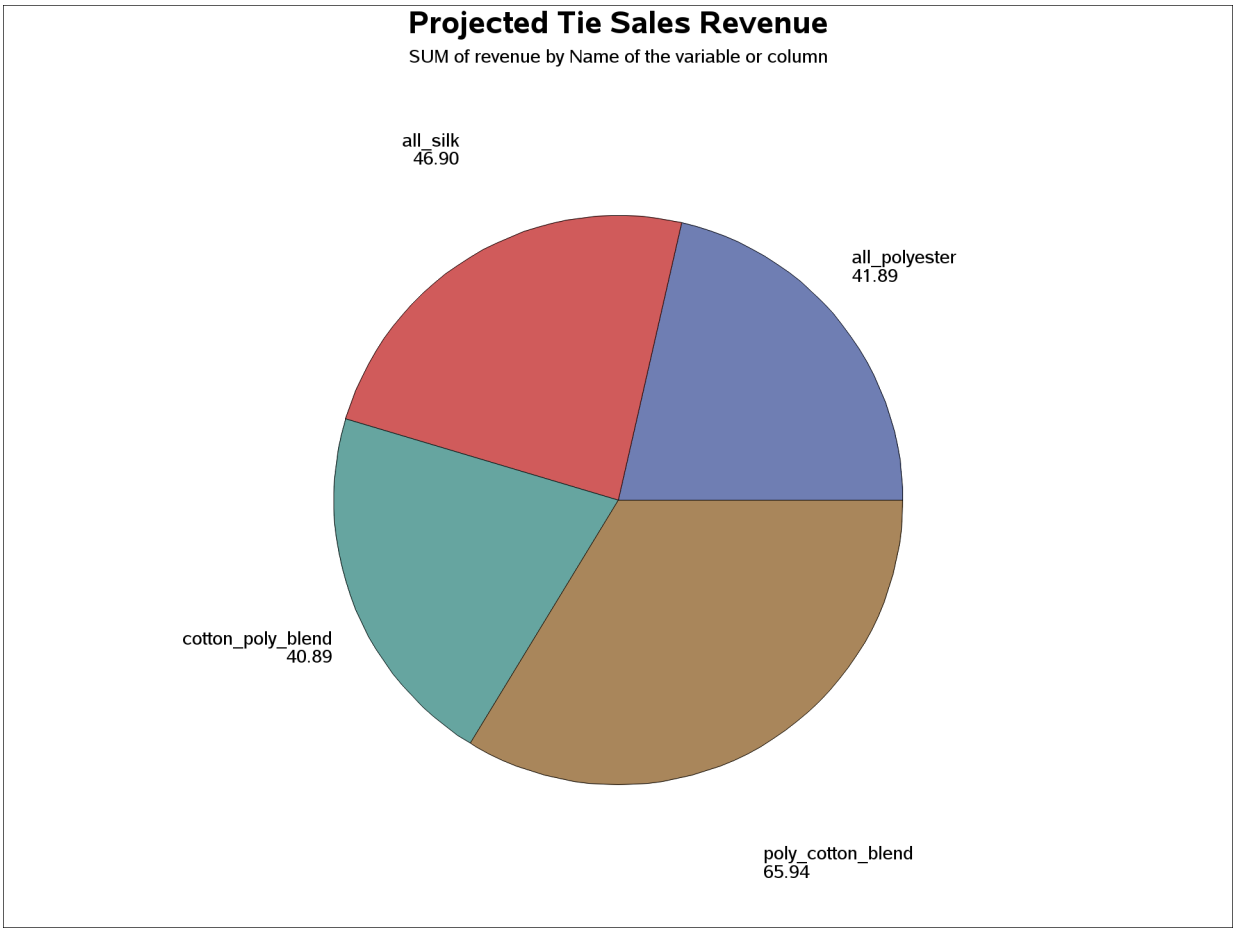


The horizontal bar chart is just one way of displaying the solution to a mathematical program. The solution to the tie product mix problem that was solved using PROC LP can also be illustrated using PROC GCHART.

```
proc gchart data=product;
   pie _var_ / sumvar=revenue;
title h=2.5 'Projected Tie Sales Revenue';
run;
```

The pie chart in Figure 3.11 shows the relative contribution of each product to total revenues.

Figure 3.11 Tie Problem: Projected Tie Sales Revenue



The TABULATE procedure can also help automate solution reporting. Several examples in Chapter 5, "The LP Procedure," illustrate its use.

Chapter 4
The INTPOINT Procedure

Contents

Overview: INTPOINT Procedure **38**
Mathematical Description of NPSC 39
Mathematical Description of LP 41
The Interior Point Algorithm 41
Network Models 48
Getting Started: INTPOINT Procedure **55**
NPSC Problems 55
LP Problems 62
Typical PROC INTPOINT Run 69
Syntax: INTPOINT Procedure **71**
Functional Summary 71
PROC INTPOINT Statement 73
CAPACITY Statement 92
COEF Statement 93
COLUMN Statement 93
COST Statement 94
DEMAND Statement 94
HEADNODE Statement 94
ID Statement 95
LO Statement 95
NAME Statement 95
NODE Statement 96
QUIT Statement 96
RHS Statement 96
ROW Statement 96
RUN Statement 97
SUPDEM Statement 97
SUPPLY Statement 97
TAILNODE Statement 97
TYPE Statement 98
VAR Statement 99
Details: INTPOINT Procedure **100**
Input Data Sets 100
Output Data Sets 109
Converting Any PROC INTPOINT Format to an MPS-Format SAS Data Set 111
Case Sensitivity 111

Loop Arcs . . . 112
Multiple Arcs . . . 112
Flow and Value Bounds . . . 112
Tightening Bounds and Side Constraints . . . 112
Reasons for Infeasibility . . . 113
Missing S Supply and Missing D Demand Values . . . 114
Balancing Total Supply and Total Demand . . . 118
How to Make the Data Read of PROC INTPOINT More Efficient . . . 119
Stopping Criteria . . . 123
Examples: INTPOINT Procedure . . . **126**
Example 4.1: Production, Inventory, Distribution Problem . . . 127
Example 4.2: Altering Arc Data . . . 134
Example 4.3: Adding Side Constraints . . . 139
Example 4.4: Using Constraints and More Alteration to Arc Data . . . 146
Example 4.5: Nonarc Variables in the Side Constraints . . . 151
Example 4.6: Solving an LP Problem with Data in MPS Format . . . 158
Example 4.7: Converting to an MPS-Format SAS Data Set . . . 160
Example 4.8: Migration to OPTMODEL: Production, Inventory, Distribution . . . 162
References . . . **166**

Overview: INTPOINT Procedure

The INTPOINT procedure solves the Network Program with Side Constraints (NPSC) problem (defined in the section "Mathematical Description of NPSC" on page 39) and the more general Linear Programming (LP) problem (defined in the section "Mathematical Description of LP" on page 41). NPSC and LP models can be used to describe a wide variety of real-world applications ranging from production, inventory, and distribution problems to financial applications.

Whether your problem is NPSC or LP, PROC INTPOINT uses the same optimization algorithm, the interior point algorithm. This algorithm is outlined in the section "The Interior Point Algorithm" on page 41.

While many of your problems may best be formulated as LP problems, there may be other instances when your problems are better formulated as NPSC problems. The section "Network Models" on page 48 describes typical models that have a network component and suggests reasons why NPSC may be preferable to LP. The section "NPSC Problems" on page 55 outlines how you supply data of any NPSC problem to PROC INTPOINT and call the procedure. After it reads the NPSC data, PROC INTPOINT converts the problem into an equivalent LP problem, performs interior point optimization, then converts the solution it finds back into a form you can use as the optimum to the original NPSC model.

If your model is an LP problem, the way you supply the data to PROC INTPOINT and run the procedure is described in the section "LP Problems" on page 62.

You can also solve LP problems by using the OPTLP procedure. The OPTLP procedure requires a linear program to be specified by using a SAS data set that adheres to the MPS format, a widely accepted format

in the optimization community. You can use the MPSOUT= option in the INTPOINT procedure to convert typical PROC INTPOINT format data sets into MPS-format SAS data sets.

The remainder of this chapter is organized as follows:

- The section "Typical PROC INTPOINT Run" on page 69 describes how to use this procedure.
- The section "Syntax: INTPOINT Procedure" on page 71 describes all the statements and options of PROC INTPOINT.
- The section "Functional Summary" on page 71 lists the statements and options that can be used to control PROC INTPOINT.
- The section "Details: INTPOINT Procedure" on page 100 contains detailed explanations, descriptions, and advice on the use and behavior of the procedure.
- PROC INTPOINT is demonstrated by solving several examples in the section "Examples: INTPOINT Procedure" on page 126.

Mathematical Description of NPSC

A network consists of a collection of nodes joined by a collection of arcs. The arcs connect nodes and convey flow of one or more commodities that are supplied at supply nodes and demanded at demand nodes in the network. Each arc has a cost per unit of flow, a flow capacity, and a lower flow bound associated with it. An important concept in network modeling is *conservation of flow*. Conservation of flow means that the total flow in arcs directed toward a node, plus the supply at the node, minus the demand at the node, equals the total flow in arcs directed away from the node.

Often all the details of a problem cannot be specified in a network model alone. In many of these cases, these details can be represented by the addition of side constraints to the model. Side constraints are linear functions of arc variables (variables containing flow through an arc) and nonarc variables (variables that are not part of the network). The data for a side constraint consist of coefficients of arcs and coefficients of nonarc variables, a constraint type (that is, $\leq$, =, or $\geq$) and a right-hand-side value (rhs). A nonarc variable has a name, an objective function coefficient analogous to an arc cost, an upper bound analogous to an arc capacity, and a lower bound analogous to an arc lower flow bound.

If a network component of NPSC is removed by merging arcs and nonarc variables into a single set of variables, and if the flow conservation constraints and side constraints are merged into a single set of constraints, the result is an LP problem. PROC INTPOINT will automatically transform an NPSC problem into an equivalent LP problem, perform the optimization, then transform the problem back into its original form. By doing this, PROC INTPOINT finds the flow through the network and the values of any nonarc variables that minimize the total cost of the solution. Flow conservation is met, flow through each arc is on or between the arc's lower flow bound and capacity, the value of each nonarc variable is on or between the nonarc's lower and upper bounds, and the side constraints are satisfied.

Note that, since many LPs have large embedded networks, PROC INTPOINT is an attractive alternative to the LP procedure in many cases. Rather than formulating all problems as LPs, network models remain conceptually easy since they are based on network diagrams that represent the problem pictorially. PROC INTPOINT accepts the network specification in a format that is particularly suited to networks. This not only

simplifies problem description but also aids in the interpretation of the solution. The conversion to and from the equivalent LP is done "behind the scenes" by the procedure.

If a network programming problem with side constraints has n nodes, a arcs, g nonarc variables, and k side constraints, then the formal statement of the problem solved by PROC INTPOINT is

$$\begin{array}{ll} \text{minimize} & c^T x + d^T z \\ \text{subject to} & Fx = b \\ & Hx + Qz \{\geq, =, \leq\} r \\ & l \leq x \leq u \\ & m \leq z \leq v \end{array}$$

where

- c is the $a \times 1$ arc variable objective function coefficient vector (the cost vector)
- x is the $a \times 1$ arc variable value vector (the flow vector)
- d is the $g \times 1$ nonarc variable objective function coefficient vector
- z is the $g \times 1$ nonarc variable value vector
- F is the $n \times a$ node-arc incidence matrix of the network, where

$$F_{i,j} = \begin{cases} -1, & \text{if arc } j \text{ is directed from node } i \\ 1, & \text{if arc } j \text{ is directed toward node } i \\ 0, & \text{otherwise} \end{cases}$$

- b is the $n \times 1$ node supply/demand vector, where

$$b_i = \begin{cases} s, & \text{if node } i \text{ has supply capability of } s \text{ units of flow} \\ -d, & \text{if node } i \text{ has demand of } d \text{ units of flow} \\ 0, & \text{if node } i \text{ is a transshipment node} \end{cases}$$

- H is the $k \times a$ side constraint coefficient matrix for arc variables, where $H_{i,j}$ is the coefficient of arc j in the ith side constraint
- Q is the $k \times g$ side constraint coefficient matrix for nonarc variables, where $Q_{i,j}$ is the coefficient of nonarc j in the ith side constraint
- r is the $k \times 1$ side constraint right-hand-side vector
- l is the $a \times 1$ arc lower flow bound vector
- u is the $a \times 1$ arc capacity vector
- m is the $g \times 1$ nonarc variable lower bound vector
- v is the $g \times 1$ nonarc variable upper bound vector

The INTPOINT procedure can also be used to solve an unconstrained network problem, that is, one in which H, Q, d, r, and z do not exist. It can also be used to solve a network problem with side constraints but no nonarc variables, in which case Q, d, and z do not exist.

Mathematical Description of LP

A linear programming (LP) problem has a linear objective function and a collection of linear constraints. PROC INTPOINT finds the values of variables that minimize the total cost of the solution. The value of each variable is on or between the variable's lower and upper bounds, and the constraints are satisfied.

If an LP has g variables and k constraints, then the formal statement of the problem solved by PROC INTPOINT is

$$\begin{array}{ll} \text{minimize} & d^T z \\ \text{subject to} & Qz \{\geq, =, \leq\} r \\ & m \leq z \leq v \end{array}$$

where

- d is the $g \times 1$ variable objective function coefficient vector
- z is the $g \times 1$ variable value vector
- Q is the $k \times g$ constraint coefficient matrix for the variables, where $Q_{i,j}$ is the coefficient of variable j in the ith constraint
- r is the $k \times 1$ side constraint right-hand-side vector
- m is the $g \times 1$ variable lower bound vector
- v is the $g \times 1$ variable upper bound vector

The Interior Point Algorithm

The simplex algorithm, developed shortly after World War II, was for many years the main method used to solve linear programming problems. Over the last fifteen years, however, the interior point algorithm has been developed. This algorithm also solves linear programming problems. From the start it showed great theoretical promise, and considerable research in the area resulted in practical implementations that performed competitively with the simplex algorithm. More recently, interior point algorithms have evolved to become superior to the simplex algorithm, in general, especially when the problems are large.

There are many variations of interior point algorithms. PROC INTPOINT uses the Primal-Dual with Predictor-Corrector algorithm. More information on this particular algorithm and related theory can be found in the texts by Roos, Terlaky, and Vial (1997), Wright (1997), and Ye (1996).

Interior Point Algorithmic Details

After preprocessing, the linear program to be solved is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

This is the *primal* problem. The matrices d, z, and Q of NPSC have been renamed c, x, and A, respectively, as these symbols are by convention used more, the problem to be solved is different from the original because of preprocessing, and there has been a change of primal variable to transform the LP into one whose variables have zero lower bounds. To simplify the algebra here, assume that variables have infinite upper bounds, and constraints are equalities. (Interior point algorithms do efficiently handle finite upper bounds, and it is easy to introduce primal slack variables to change inequalities into equalities.) The problem has n variables; i is a variable number; k is an iteration number, and if used as a subscript or superscript it denotes "of iteration k".

There exists an equivalent problem, the *dual* problem, stated as

$$\begin{array}{ll} \text{maximize} & b^T y \\ \text{subject to} & A^T y + s = c \\ & s \geq 0 \end{array}$$

where y are dual variables, and s are dual constraint slacks.

The interior point algorithm solves the system of equations to satisfy the Karush-Kuhn-Tucker (KKT) conditions for optimality:

$$\begin{aligned} & Ax = b \\ & A^T y + s = c \\ & XSe = 0 \\ & x \geq 0 \\ & s \geq 0 \end{aligned}$$

where

$$\begin{aligned} & S = \text{diag}(s) \text{ (that is, } S_{i,j} = s_i \text{ if } i = j, S_{i,j} = 0 \text{ otherwise)} \\ & X = \text{diag}(x) \\ & e_i = 1 \; \forall i \end{aligned}$$

These are the conditions for feasibility, with the *complementarity* condition $XSe = 0$ added. Complementarity forces the optimal objectives of the primal and dual to be equal, $c^T x_{opt} = b^T y_{opt}$, as

$$\begin{aligned} & 0 = x_{opt}^T s_{opt} = s_{opt}^T x_{opt} = (c - A^T y_{opt})^T x_{opt} = \\ & c^T x_{opt} - y_{opt}^T (A x_{opt}) = c^T x_{opt} - b^T y_{opt} \end{aligned}$$

Before the optimum is reached, a solution (x, y, s) may not satisfy the KKT conditions:

- Primal constraints may be violated, $infeas_c = b - Ax \neq 0$.
- Dual constraints may be violated, $infeas_d = c - A^T y - s \neq 0$.
- Complementarity may not be satisfied, $x^T s = c^T x - b^T y \neq 0$. This is called the *duality gap*.

The interior point algorithm works by using Newton's method to find a direction to move $(\Delta x^k, \Delta y^k, \Delta s^k)$ from the current solution (x^k, y^k, s^k) toward a better solution:

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where α is the *step length* and is assigned a value as large as possible but not so large that an x_i^{k+1} or s_i^{k+1} is "too close" to zero. The direction in which to move is found using

$$
\begin{aligned}
A\Delta x^k &= infeas_c \\
A^T\Delta y^k + \Delta s^k &= infeas_d \\
S^k\Delta x^k + X^k\Delta s^k &= -X^k S^k e
\end{aligned}
$$

To greatly improve performance, the third equation is changed to

$$S^k\Delta x^k + X^k\Delta s^k = -X^k S^k e + \sigma_k\mu_k e$$

where $\mu_k = (x^k)^T s^k/n$, the average complementarity, and $0 \leq \sigma_k \leq 1$.

The effect now is to find a direction in which to move to reduce infeasibilities and to reduce the complementarity toward zero, but if any $x_i^k s_i^k$ is too close to zero, it is "nudged out" to μ, and any $x_i^k s_i^k$ that is larger than μ is "nudged into" μ. A σ_k close to or equal to 0.0 biases a direction toward the optimum, and a value of σ_k close to or equal to 1.0 "centers" the direction toward a point where all pairwise products $x_i^k s_i^k = \mu$. Such points make up the *central path* in the interior. Although centering directions make little, if any, progress in reducing μ and moving the solution closer to the optimum, substantial progress toward the optimum can usually be made in the next iteration.

The central path is crucial to why the interior point algorithm is so efficient. As μ is decreased, this path "guides" the algorithm to the optimum through the interior of feasible space. Without centering, the algorithm would find a series of solutions near each other close to the boundary of feasible space. Step lengths along the direction would be small and many more iterations would probably be required to reach the optimum.

That in a nutshell is the primal-dual interior point algorithm. Varieties of the algorithm differ in the way α and σ_k are chosen and the direction adjusted during each iteration. A wealth of information can be found in the texts by Roos, Terlaky, and Vial (1997), Wright (1997), and Ye (1996).

The calculation of the direction is the most time-consuming step of the interior point algorithm. Assume the kth iteration is being performed, so the subscript and superscript k can be dropped from the algebra:

$$
\begin{aligned}
A\Delta x &= infeas_c \\
A^T\Delta y + \Delta s &= infeas_d \\
S\Delta x + X\Delta s &= -XSe + \sigma\mu e
\end{aligned}
$$

Rearranging the second equation,

$$\Delta s = infeas_d - A^T\Delta y$$

Rearranging the third equation,

$$
\begin{aligned}
\Delta s &= X^{-1}(-S\Delta x - XSe + \sigma\mu e) \\
\Delta s &= -\Theta\Delta x - Se + X^{-1}\sigma\mu e
\end{aligned}
$$

where $\Theta = SX^{-1}$.

Equating these two expressions for Δs and rearranging,

$$
\begin{aligned}
&-\Theta\Delta x - Se + X^{-1}\sigma\mu e = infeas_d - A^T\Delta y \\
&-\Theta\Delta x = Se - X^{-1}\sigma\mu e + infeas_d - A^T\Delta y \\
&\Delta x = \Theta^{-1}(-Se + X^{-1}\sigma\mu e - infeas_d + A^T\Delta y) \\
&\Delta x = \rho + \Theta^{-1}A^T\Delta y
\end{aligned}
$$

where $\rho = \Theta^{-1}(-Se + X^{-1}\sigma\mu e - infeas_d)$.

Substituting into the first direction equation,

$$
\begin{aligned}
&A\Delta x = infeas_c \\
&A(\rho + \Theta^{-1}A^T\Delta y) = infeas_c \\
&A\Theta^{-1}A^T\Delta y = infeas_c - A\rho \\
&\Delta y = (A\Theta^{-1}A^T)^{-1}(infeas_c - A\rho)
\end{aligned}
$$

Θ, ρ, Δy, Δx, and Δs are calculated in that order. The hardest term is the factorization of the $(A\Theta^{-1}A^T)$ matrix to determine Δy. Fortunately, although the *values* of $(A\Theta^{-1}A^T)$ are different for each iteration, the *locations* of the nonzeros in this matrix remain fixed; the nonzero locations are the same as those in the matrix (AA^T). This is because $\Theta^{-1} = XS^{-1}$ is a diagonal matrix that has the effect of merely scaling the columns of (AA^T).

The fact that the nonzeros in $A\Theta^{-1}A^T$ have a constant pattern is exploited by all interior point algorithms and is a major reason for their excellent performance. Before iterations begin, AA^T is examined and its rows and columns are symmetrically permuted so that during Cholesky factorization, the number of *fill-ins* created is smaller. A list of arithmetic operations to perform the factorization is saved in concise computer data structures (working with memory locations rather than actual numerical values). This is called *symbolic factorization*. During iterations, when memory has been initialized with numerical values, the operations list is performed sequentially. Determining how the factorization should be performed again and again is unnecessary.

The Primal-Dual Predictor-Corrector Interior Point Algorithm

The variant of the interior point algorithm implemented in PROC INTPOINT is a Primal-Dual Predictor-Corrector interior point algorithm. At first, Newton's method is used to find a direction $(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff})$ to move, but calculated as if μ is zero, that is, as a step with no centering, known as an *affine* step:

$$
\begin{aligned}
&A\Delta x^k_{aff} = infeas_c \\
&A^T\Delta y^k_{aff} + \Delta s^k_{aff} = infeas_d \\
&S^k\Delta x^k_{aff} + X^k\Delta s^k_{aff} = -X^kS^ke \\
&(x^k_{aff}, y^k_{aff}, s^k_{aff}) = (x^k, y^k, s^k) + \alpha(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff})
\end{aligned}
$$

where α is the *step length* as before.

Complementarity $x^T s$ is calculated at $(x^k_{aff}, y^k_{aff}, s^k_{aff})$ and compared with the complementarity at the starting point (x^k, y^k, s^k), and the success of the affine step is gauged. If the affine step was successful in reducing the complementarity by a substantial amount, the need for centering is not great, and σ_k in the following linear system is assigned a value close to zero. If, however, the affine step was unsuccessful, centering would be beneficial, and σ_k in the following linear system is assigned a value closer to 1.0. The value of σ_k is therefore adaptively altered depending on the progress made toward the optimum.

A second linear system is solved to determine a centering vector $(\Delta x^k_c, \Delta y^k_c, \Delta s^k_c)$ from $(x^k_{aff}, y^k_{aff}, s^k_{aff})$:

$$
\begin{aligned}
&A\Delta x^k_c = 0\\
&A^T \Delta y^k_c + \Delta s^k_c = 0\\
&S^k \Delta x^k_c + X^k \Delta s^k_c = -X^k_{aff} S^k_{aff} e + \sigma_k \mu_k e
\end{aligned}
$$

Then

$$(\Delta x^k, \Delta y^k, \Delta s^k) = (\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff}) + (\Delta x^k_c, \Delta y^k_c, \Delta s^k_c)$$

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where, as before, α is the *step length* assigned a value as large as possible but not so large that an x_i^{k+1} or s_i^{k+1} is "too close" to zero.

Although the Predictor-Corrector variant entails solving two linear systems instead of one, fewer iterations are usually required to reach the optimum. The additional overhead of calculating the second linear system is small, as the factorization of the $(A\Theta^{-1}A^T)$ matrix has already been performed to solve the first linear system.

Interior Point: Upper Bounds

If the LP had upper bounds ($0 \le x \le u$ where u is the upper bound vector), then the primal and dual problems, the duality gap, and the KKT conditions would have to be expanded.

The primal linear program to be solved is

$$
\begin{aligned}
\text{minimize} \quad & c^T x\\
\text{subject to} \quad & Ax = b\\
& 0 \le x \le u
\end{aligned}
$$

where $0 \le x \le u$ is split into $x \ge 0$ and $x \le u$. Let z be primal slack so that $x + z = u$, and associate dual variables w with these constraints. The interior point algorithm solves the system of equations to satisfy the Karush-Kuhn-Tucker (KKT) conditions for optimality:

$$
\begin{aligned}
&Ax = b\\
&x + z = u\\
&A^T y + s - w = c\\
&XSe = 0\\
&ZWe = 0\\
&x, s, z, w \ge 0
\end{aligned}
$$

These are the conditions for feasibility, with the *complementarity* conditions $XSe = 0$ and $ZWe = 0$ added. Complementarity forces the optimal objectives of the primal and dual to be equal, $c^T x_{opt} = b^T y_{opt} - u^T w_{opt}$, as

$$0 = z_{opt}^T w_{opt} = (u - x_{opt})^T w_{opt} = u^T w_{opt} - x_{opt}^T w_{opt}$$
$$0 = x_{opt}^T s_{opt} = s_{opt}^T x_{opt} = (c - A^T y_{opt} + w_{opt})^T x_{opt} =$$
$$c^T x_{opt} - y_{opt}^T (A x_{opt}) + w_{opt}^T x_{opt} = c^T x_{opt} - b^T y_{opt} + u^T w_{opt}$$

Before the optimum is reached, a solution (x, y, s, z, w) might not satisfy the KKT conditions:

- Primal bound constraints may be violated, $infeas_b = u - x - z \neq 0$.
- Primal constraints may be violated, $infeas_c = b - Ax \neq 0$.
- Dual constraints may be violated, $infeas_d = c - A^T y - s + w \neq 0$.
- Complementarity conditions may not be satisfied, $x^T s \neq 0$ or $z^T w \neq 0$.

The calculations of the interior point algorithm can easily be derived in a fashion similar to calculations for when an LP has no upper bounds. See the paper by Lustig, Marsten, and Shanno (1992).

In some iteration *k*, the *affine* step system that must be solved is

$$\Delta x_{aff} + \Delta z_{aff} = infeas_b$$
$$A \Delta x_{aff} = infeas_c$$
$$A^T \Delta y_{aff} + \Delta s_{aff} - \Delta w_{aff} = infeas_d$$
$$S \Delta x_{aff} + X \Delta s_{aff} = -XSe$$
$$Z \Delta w_{aff} + W \Delta z_{aff} = -ZWe$$

Therefore, the computations involved in solving the affine step are

$$\Theta = SX^{-1} + WZ^{-1}$$
$$\rho = \Theta^{-1}(infeas_d + (S - W)e - Z^{-1}W\ infeas_b)$$
$$\Delta y_{aff} = (A\Theta^{-1}A^T)^{-1}(infeas_c + A\rho)$$
$$\Delta x_{aff} = \Theta^{-1}A^T \Delta y_{aff} - \rho$$
$$\Delta z_{aff} = infeas_b - \Delta x_{aff}$$
$$\Delta w_{aff} = -We - Z^{-1}W \Delta z_{aff}$$
$$\Delta s_{aff} = -Se - X^{-1}S \Delta x_{aff}$$

$$(x_{aff}, y_{aff}, s_{aff}, z_{aff}, w_{aff}) = (x, y, s, z, w) +$$
$$\alpha(\Delta x_{aff}, \Delta y_{aff}, \Delta s_{aff}, \Delta z_{aff}, \Delta w_{aff})$$

and α is the *step length* as before.

A second linear system is solved to determine a centering vector $(\Delta x_c, \Delta y_c, \Delta s_c, \Delta z_c, \Delta w_c)$ from $(x_{aff}, y_{aff}, s_{aff}, z_{aff}, w_{aff})$:

$$\begin{aligned}
&\Delta x_c + \Delta z_c = 0\\
&A\Delta x_c = 0\\
&A^T\Delta y_c + \Delta s_c - \Delta w_c = 0\\
&S\Delta x_c + X\Delta s_c = -X_{aff}S_{aff}e + \sigma\mu e\\
&Z\Delta w_c + W\Delta z_c = -Z_{aff}W_{aff}e + \sigma\mu e
\end{aligned}$$

where

$$\begin{aligned}
&\zeta_{start} = x^T s + z^T w, \text{ complementarity at the start of the iteration}\\
&\zeta_{aff} = x_{aff}^T s_{aff} + z_{aff}^T w_{aff}, \text{ the affine complementarity}\\
&\mu = \zeta_{aff}/2n, \text{ the average complementarity}\\
&\sigma = (\zeta_{aff}/\zeta_{start})^3
\end{aligned}$$

Therefore, the computations involved in solving the centering step are

$$\begin{aligned}
&\rho = \Theta^{-1}(\sigma\mu(X^{-1} - Z^{-1})e - X^{-1}X_{aff}S_{aff}e + Z^{-1}Z_{aff}W_{aff}e)\\
&\Delta y_c = (A\Theta^{-1}A^T)^{-1}A\rho\\
&\Delta x_c = \Theta^{-1}A^T\Delta y_c - \rho\\
&\Delta z_c = -\Delta x_c\\
&\Delta w_c = \sigma\mu Z^{-1}e - Z^{-1}Z_{aff}W_{aff}e - Z^{-1}W_{aff}\Delta z_c\\
&\Delta s_c = \sigma\mu X^{-1}e - X^{-1}X_{aff}S_{aff}e - X^{-1}S_{aff}\Delta x_c
\end{aligned}$$

Then

$$\begin{aligned}
&(\Delta x, \Delta y, \Delta s, \Delta z, \Delta w) =\\
&\qquad (\Delta x_{aff}, \Delta y_{aff}, \Delta s_{aff}, \Delta z_{aff}, \Delta w_{aff})\\
&\qquad +(\Delta x_c, \Delta y_c, \Delta s_c, \Delta z_c, \Delta w_c)
\end{aligned}$$

$$\begin{aligned}
&(x^{k+1}, y^{k+1}, s^{k+1}, z^{k+1}, w^{k+1}) =\\
&\qquad (x^k, y^k, s^k, z^k, w^k)\\
&\qquad +\alpha(\Delta x, \Delta y, \Delta s, \Delta z, \Delta w)
\end{aligned}$$

where, as before, α is the *step length* assigned a value as large as possible but not so large that an x_i^{k+1}, s_i^{k+1}, z_i^{k+1}, or w_i^{k+1} is "too close" to zero.

The algebra in this section has been simplified by assuming that *all* variables have finite upper bounds. If the number of variables with finite upper bounds $n_u < n$, you need to change the algebra to reflect that the Z and W matrices have dimension $n_u \times 1$ or $n_u \times n_u$. Other computations need slight modification. For example, the average complementarity is

$$\mu = x_{aff}^T s_{aff}/n + z_{aff}^T w_{aff}/n_u$$

An important point is that any upper bounds can be handled by specializing the algorithm and *not* by generating the constraints $x \leq u$ and adding these to the main primal constraints $Ax = b$.

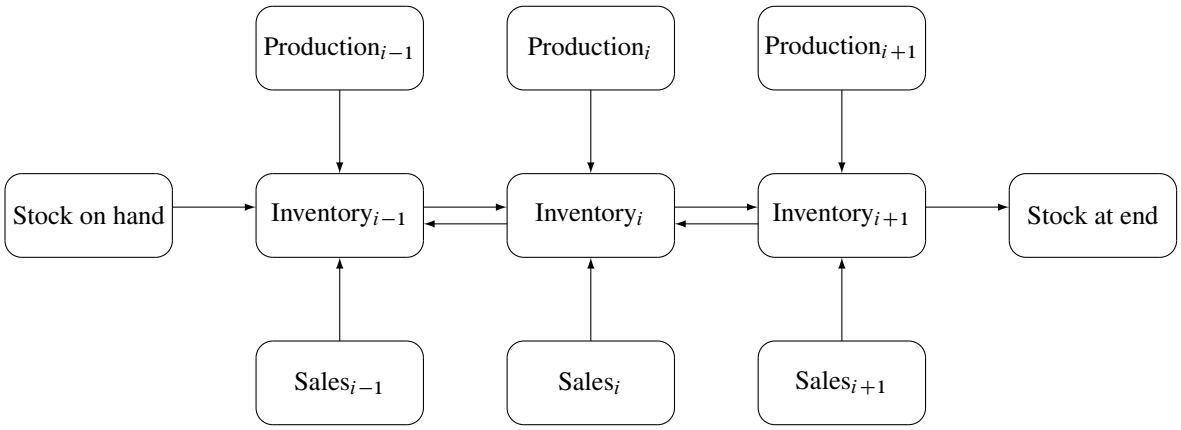
Network Models

The following are descriptions of some typical NPSC models.

Production, Inventory, and Distribution (Supply Chain) Problems

One common class of network models is the production-inventory-distribution or supply-chain problem. The diagram in Figure 4.1 illustrates this problem. The subscripts on the Production, Inventory, and Sales nodes indicate the time period. By replicating sections of the model, the notion of time can be included.

Figure 4.1 Production-Inventory-Distribution Problem



In this type of model, the nodes can represent a wide variety of facilities. Several examples are suppliers, spot markets, importers, farmers, manufacturers, factories, parts of a plant, production lines, waste disposal facilities, workstations, warehouses, coolstores, depots, wholesalers, export markets, ports, rail junctions, airports, road intersections, cities, regions, shops, customers, and consumers. The diversity of this selection demonstrates how rich the potential applications of this model are.

Depending upon the interpretation of the nodes, the objectives of the modeling exercise can vary widely. Some common types of objectives are

- to reduce collection or purchase costs of raw materials

- to reduce inventory holding or backorder costs. Warehouses and other storage facilities sometimes have capacities, and there can be limits on the amount of goods that can be placed on backorder.
- to decide where facilities should be located and what the capacity of these should be. Network models have been used to help decide where factories, hospitals, ambulance and fire stations, oil and water wells, and schools should be sited.
- to determine the assignment of resources (machines, production capability, workforce) to tasks, schedules, classes, or files
- to determine the optimal distribution of goods or services. This usually means minimizing transportation costs and reducing transit time or distances covered.
- to find the shortest path from one location to another
- to ensure that demands (for example, production requirements, market demands, contractual obligations) are met
- to maximize profits from the sale of products or the charge for services
- to maximize production by identifying bottlenecks

Some specific applications are

- car distribution models. These help determine which models and numbers of cars should be manufactured in which factories and where to distribute cars from these factories to zones in the United States in order to meet customer demand at least cost.
- models in the timber industry. These help determine when to plant and mill forests, schedule production of pulp, paper, and wood products, and distribute products for sale or export.
- military applications. The nodes can be theaters, bases, ammunition dumps, logistical suppliers, or radar installations. Some models are used to find the best ways to mobilize personnel and supplies and to evacuate the wounded in the least amount of time.
- communications applications. The nodes can be telephone exchanges, transmission lines, satellite links, and consumers. In a model of an electrical grid, the nodes can be transformers, powerstations, watersheds, reservoirs, dams, and consumers. The effect of high loads or outages might be of concern.

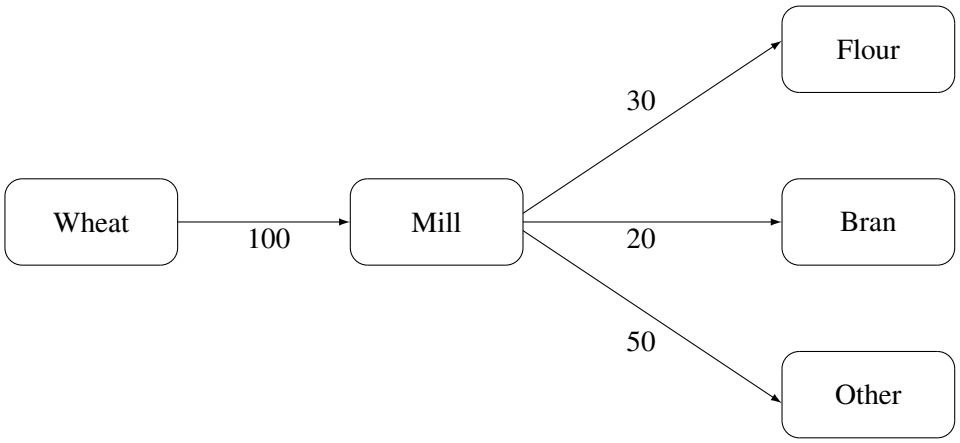
Proportionality Constraints

In many models, you have the characteristic that a flow through an arc must be proportional to the flow through another arc. Side constraints are often necessary to model that situation. Such constraints are called *proportionality constraints* and are useful in models where production is subject to refining or modification into different materials. The amount of each output, or any waste, evaporation, or reduction can be specified as a proportion of input.

Typically, the arcs near the supply nodes carry raw materials and the arcs near the demand nodes carry refined products. For example, in a model of the milling industry, the flow through some arcs may represent quantities of wheat. After the wheat is processed, the flow through other arcs might be flour. For others it might be bran. The side constraints model the relationship between the amount of flour or bran produced as

a proportion of the amount of wheat milled. Some of the wheat can end up as neither flour, bran, nor any useful product, so this waste is drained away via arcs to a waste node.

Figure 4.2 Proportionality Constraints



In order for arcs to be specified in side constraints, they must be named. By default, PROC INTPOINT names arcs using the names of the nodes at the head and tail of the arc. An arc is named with its tail node name followed by an underscore and its head node name. For example, an arc from node *from* to node *to* is called *from_to*.

Consider the network fragment in Figure 4.2. The arc Wheat_Mill conveys the wheat milled. The cost of flow on this arc is the milling cost. The capacity of this arc is the capacity of the mill. The lower flow bound on this arc is the minimum quantity that must be milled for the mill to operate economically. The constraints

$$0.3\,\mathsf{Wheat_Mill} - \mathsf{Mill_Flour} = 0.0$$

$$0.2\,\mathsf{Wheat_Mill} - \mathsf{Mill_Bran} = 0.0$$

force every unit of wheat that is milled to produce 0.3 units of flour and 0.2 units of bran. Note that it is not necessary to specify the constraint

$$0.5\,\mathsf{Wheat_Mill} - \mathsf{Mill_Other} = 0.0$$

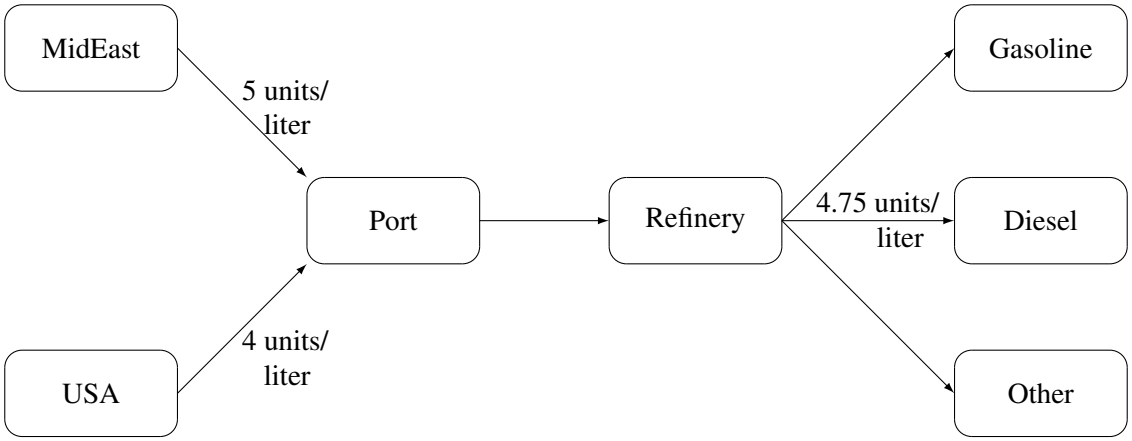
since flow conservation implies that any flow that does not traverse through Mill_Flour or Mill_Bran must be conveyed through Mill_Other. And, computationally, it is better if this constraint is not specified, since there is one less side constraint and fewer problems with numerical precision. Notice that the sum of the proportions must equal 1.0 exactly; otherwise, flow conservation is violated.

Blending Constraints

Blending or quality constraints can also influence the recipes or proportions of ingredients that are mixed. For example, different raw materials can have different properties. In an application of the oil industry, the amount of products that are obtained could be different for each type of crude oil. Furthermore, fuel might have a minimum octane requirement or limited sulphur or lead content, so that a blending of crudes is needed to produce the product.

The network fragment in Figure 4.3 shows an example of this.

Figure 4.3 Blending Constraints



The arcs MidEast_Port and USA_Port convey crude oil from the two sources. The arc Port_Refinery represents refining while the arcs Refinery_Gasoline and Refinery_Diesel carry the gas and diesel produced. The proportionality constraints

$$0.4\,\text{Port_Refinery} - \text{Refinery_Gasoline} = 0.0$$

$$0.2\,\text{Port_Refinery} - \text{Refinery_Diesel} = 0.0$$

capture the restrictions for producing gasoline and diesel from crude. Suppose that only crude from the Middle East is used, then the resulting diesel would contain 5 units of sulphur per liter. If only crude from the U.S.A. is used, the resulting diesel would contain 4 units of sulphur per liter. Diesel can have at most 4.75 units of sulphur per liter. Some crude from the U.S.A. must be used if Middle East crude is used in order to meet the 4.75 sulphur per liter limit. The side constraint to model this requirement is

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 4.75\,\text{Port_Refinery} \leq 0.0$$

Since Port_Refinery = MidEast_Port + USA_Port, flow conservation allows this constraint to be simplified to

$$1\,\text{MidEast_Port} - 3\,\text{USA_Port} \leq 0.0$$

If, for example, 120 units of crude from the Middle East is used, then at least 40 units of crude from the U.S.A. must be used. The preceding constraint is simplified because you assume that the sulphur concentration of diesel is proportional to the sulphur concentration of the crude mix. If this is not the case, the relation

$$0.2\,\text{Port_Refinery} = \text{Refinery_Diesel}$$

is used to obtain

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 4.75\,(1.0/0.2\,\text{Refinery_Diesel}) \leq 0.0$$

which equals

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 23.75\,\text{Refinery_Diesel} \leq 0.0$$

An example similar to this oil industry problem is solved in the section "Introductory NPSC Example" on page 57.

Multicommodity Problems

Side constraints are also used in models in which there are capacities on transportation or some other shared resource, or there are limits on overall production or demand in multicommodity, multidivisional, or multiperiod problems. Each commodity, division, or period can have a separate network coupled to one main system by the side constraints. Side constraints are used to combine the outputs of subdivisions of a problem (either commodities, outputs in distinct time periods, or different process streams) to meet overall demands or to limit overall production or expenditures. This method is more desirable than doing separate *local* optimizations for individual commodity, process, or time networks and then trying to establish relationships between each when determining an overall policy if the *global* constraint is not satisfied. Of course, to make models more realistic, side constraints may be necessary in the local problems.

Figure 4.4 Multicommodity Problem

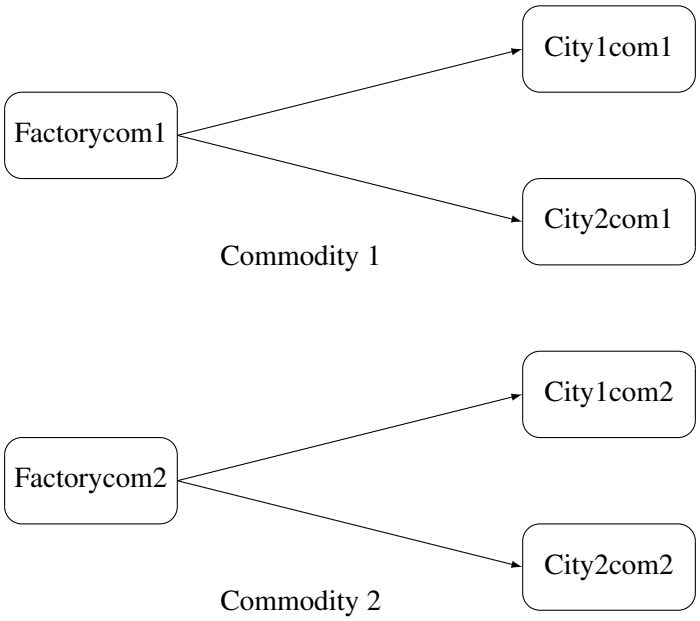


Figure 4.4 shows two network fragments. They represent identical production and distribution sites of two different commodities. Suffix *com1* represents commodity 1 and suffix *com2* represents commodity 2. The nodes Factorycom1 and Factorycom2 model the same factory, and nodes City1com1 and City1com2 model the same location, city 1. Similarly, City2com1 and City2com2 are the same location, city 2. Suppose that commodity 1 occupies 2 cubic meters, commodity 2 occupies 3 cubic meters, the truck dispatched to city 1 has a capacity of 200 cubic meters, and the truck dispatched to city 2 has a capacity of 250 cubic meters. How much of each commodity can be loaded onto each truck? The side constraints for this case are

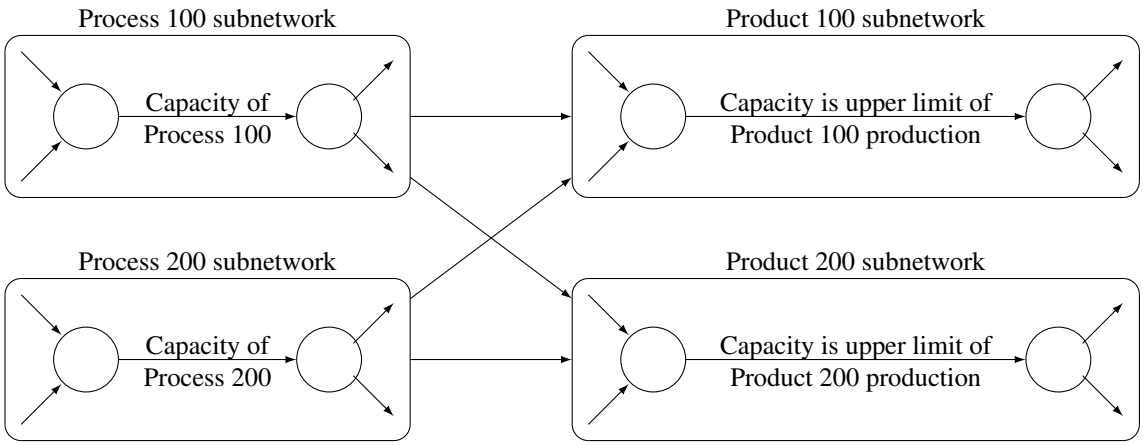
$$2\,\text{Factorycom1_City1com1} + 3\,\text{Factorycom2_City1com2} \leq 200$$

$$2\,\text{Factorycom1_City2com1} + 3\,\text{Factorycom2_City2com2} \leq 250$$

Large Modeling Strategy

In many cases, the flow through an arc might actually represent the flow or movement of a commodity from place to place or from time period to time period. However, sometimes an arc is included in the network as a method of capturing some aspect of the problem that you would not normally think of as part of a network model. There is no commodity movement associated with that arc. For example, in a multiprocess, multiproduct model (Figure 4.5), there might be subnetworks for each process and each product. The subnetworks can be joined together by a set of arcs that have flows that represent the amount of product *j* produced by process *i*. To model an upper-limit constraint on the total amount of product *j* that can be produced, direct all arcs carrying product *j* to a single node and from there through a single arc. The capacity of this arc is the upper limit of product *j* production. It is preferable to model this structure in the network rather than to include it in the side constraints because the efficiency of the optimizer may be less affected by a reasonable increase in the size of the network rather than increasing the number or complicating side constraints.

Figure 4.5 Multiprocess, Multiproduct Example



When starting a project, it is often a good strategy to use a small network formulation and then use that model as a framework upon which to add detail. For example, in the multiprocess, multiproduct model, you might start with the network depicted in Figure 4.5. Then, for example, the process subnetwork can be enhanced to include the distribution of products. Other phases of the operation could be included by adding more subnetworks. Initially, these subnetworks can be single nodes, but in subsequent studies they can be expanded to include greater detail.

Advantages of Network Models over LP Models

Many linear programming problems have large embedded network structures. Such problems often result when modeling manufacturing processes, transportation or distribution networks, or resource allocation, or when deciding where to locate facilities. Often, some commodity is to be moved from place to place, so the more natural formulation in many applications is that of a constrained network rather than a linear program.

Using a network diagram to visualize a problem makes it possible to capture the important relationships in an easily understood picture form. The network diagram aids the communication between model builder and model user, making it easier to comprehend how the model is structured, how it can be changed, and how results can be interpreted.

If a network structure is embedded in a linear program, the problem is an NPSC (see the section "Mathematical Description of NPSC" on page 39). When the network part of the problem is large compared to the nonnetwork part, especially if the number of side constraints is small, it is worthwhile to exploit this structure to describe the model. Rather than generating the data for the flow conservation constraints, generate instead the data for the nodes and arcs of the network.

Flow Conservation Constraints

The constraints $Fx = b$ in NPSC (see the section "Mathematical Description of NPSC" on page 39) are referred to as the nodal flow conservation constraints. These constraints algebraically state that the sum of the flow through arcs directed toward a node plus that node's supply, if any, equals the sum of the flow through arcs directed away from that node plus that node's demand, if any. The flow conservation constraints are implicit in the network model and should not be specified explicitly in side constraint data when using PROC INTPOINT to solve NPSC problems.

Nonarc Variables

Nonarc variables can be used to simplify side constraints. For example, if a sum of flows appears in many constraints, it may be worthwhile to equate this expression with a nonarc variable and use this in the other constraints. This keeps the constraint coefficient matrix sparse. By assigning a nonarc variable a nonzero objective function, it is then possible to incur a cost for using resources above some lowest feasible limit. Similarly, a profit (a negative objective function coefficient value) can be made if all available resources are not used.

In some models, nonarc variables are used in constraints to absorb excess resources or supply needed resources. Then, either the excess resource can be used or the needed resource can be supplied to another component of the model.

For example, consider a multicommodity problem of making television sets that have either 19- or 25-inch screens. In their manufacture, three and four chips, respectively, are used. Production occurs at two factories during March and April. The supplier of chips can supply only 2,600 chips to factory 1 and 3,750 chips to factory 2 each month. The names of arcs are in the form Prod*n_s_m*, where *n* is the factory number, *s* is the screen size, and *m* is the month. For example, Prod1_25_Apr is the arc that conveys the number of 25-inch TVs produced in factory 1 during April. You might have to determine similar systematic naming schemes for your application.

As described, the constraints are

$$3\,\mathrm{Prod1_19_Mar} + 4\,\mathrm{Prod1_25_Mar} \leq 2600$$

$$3\,\mathrm{Prod2_19_Mar} + 4\,\mathrm{Prod2_25_Mar} \leq 3750$$

$$3\,\mathrm{Prod1_19_Apr} + 4\,\mathrm{Prod1_25_Apr} \leq 2600$$

$$3\,\mathrm{Prod2_19_Apr} + 4\,\mathrm{Prod2_25_Apr} \leq 3750$$

If there are chips that could be obtained for use in March but not used for production in March, why not keep these unused chips until April? Furthermore, if the March excess chips at factory 1 could be used either at factory 1 or factory 2 in April, the model becomes

$$3\,\text{Prod1_19_Mar} + 4\,\text{Prod1_25_Mar} + \text{F1_Unused_Mar} = 2600$$

$$3\,\text{Prod2_19_Mar} + 4\,\text{Prod2_25_Mar} + \text{F2_Unused_Mar} = 3750$$

$$3\,\text{Prod1_19_Apr} + 4\,\text{Prod1_25_Apr} - \text{F1_Kept_Since_Mar} = 2600$$

$$3\,\text{Prod2_19_Apr} + 4\,\text{Prod2_25_Apr} - \text{F2_Kept_Since_Mar} = 3750$$

$$\text{F1_Unused_Mar} + \text{F2_Unused_Mar} \text{ (continued)}$$

$$- \text{F1_Kept_Since_Mar} - \text{F2_Kept_Since_Mar} \geq 0.0$$

where F1_Kept_Since_Mar is the number of chips used during April at factory 1 that were obtained in March at either factory 1 or factory 2, and F2_Kept_Since_Mar is the number of chips used during April at factory 2 that were obtained in March. The last constraint ensures that the number of chips used during April that were obtained in March does not exceed the number of chips not used in March. There may be a cost to hold chips in inventory. This can be modeled having a positive objective function coefficient for the nonarc variables F1_Kept_Since_Mar and F2_Kept_Since_Mar. Moreover, nonarc variable upper bounds represent an upper limit on the number of chips that can be held in inventory between March and April.

See Example 4.1 through Example 4.5, which use this TV problem. The use of nonarc variables as described previously is illustrated.

Getting Started: INTPOINT Procedure

NPSC Problems

To solve NPSC problems using PROC INTPOINT, you save a representation of the network and the side constraints in three SAS data sets. These data sets are then passed to PROC INTPOINT for solution. There are various forms that a problem's data can take. You can use any one or a combination of several of these forms.

The NODEDATA= data set contains the names of the supply and demand nodes and the supply or demand associated with each. These are the elements in the column vector *b* in the NPSC problem (see the section "Mathematical Description of NPSC" on page 39).

The ARCDATA= data set contains information about the variables of the problem. Usually these are arcs, but there can also be data related to nonarc variables in the ARCDATA= data set.

An arc is identified by the names of its tail node (where it originates) and head node (where it is directed). Each observation can be used to identify an arc in the network and, optionally, the cost per flow unit across

the arc, the arc's capacity, lower flow bound, and name. These data are associated with the matrix F and the vectors c, l, and u in the NPSC problem (see the section "Mathematical Description of NPSC" on page 39).

NOTE: Although F is a node-arc incidence matrix, it is specified in the ARCDATA= data set by arc definitions. Do not explicitly specify these flow conservation constraints as constraints of the problem.

In addition, the ARCDATA= data set can be used to specify information about nonarc variables, including objective function coefficients, lower and upper value bounds, and names. These data are the elements of the vectors d, m, and v in the NPSC problem (see the section "Mathematical Description of NPSC" on page 39). Data for an arc or nonarc variable can be given in more than one observation.

Supply and demand data also can be specified in the ARCDATA= data set. In such a case, the NODEDATA= data set may not be needed.

The CONDATA= data set describes the side constraints and their right-hand sides. These data are elements of the matrices H and Q and the vector r. Constraint types are also specified in the CONDATA= data set. You can include in this data set upper bound values or capacities, lower flow or value bounds, and costs or objective function coefficients. It is possible to give all information about some or all nonarc variables in the CONDATA= data set.

An arc is identified in this data set by its name. If you specify an arc's name in the ARCDATA= data set, then this name is used to associate data in the CONDATA= data set with that arc. Each arc also has a default name that is the name of the tail and head node of the arc concatenated together and separated by an underscore character; tail_head, for example.

If you use the dense side constraint input format (described in the section "CONDATA= Data Set" on page 101), and want to use the default arc names, these arc names are names of SAS variables in the VAR list of the CONDATA= data set.

If you use the sparse side constraint input format (see the section "CONDATA= Data Set" on page 101) and want to use the default arc names, these arc names are values of the COLUMN list variable of the CONDATA= data set.

PROC INTPOINT reads the data from the NODEDATA= data set, the ARCDATA= data set, and the CONDATA= data set. Error checking is performed, and the model is converted into an equivalent LP. This LP is preprocessed. Preprocessing is optional but highly recommended. Preprocessing analyzes the model and tries to determine before optimization whether variables can be "fixed" to their optimal values. Knowing that, the model can be modified and these variables dropped out. It can be determined that some constraints are redundant. Sometimes, preprocessing succeeds in reducing the size of the problem, thereby making the subsequent optimization easier and faster.

The optimal solution to the equivalent LP is then found. This LP is converted back to the original NPSC problem, and the optimum for this is derived from the optimum of the equivalent LP. If the problem was preprocessed, the model is now post-processed, where fixed variables are reintroduced. The solution can be saved in the CONOUT= data set.

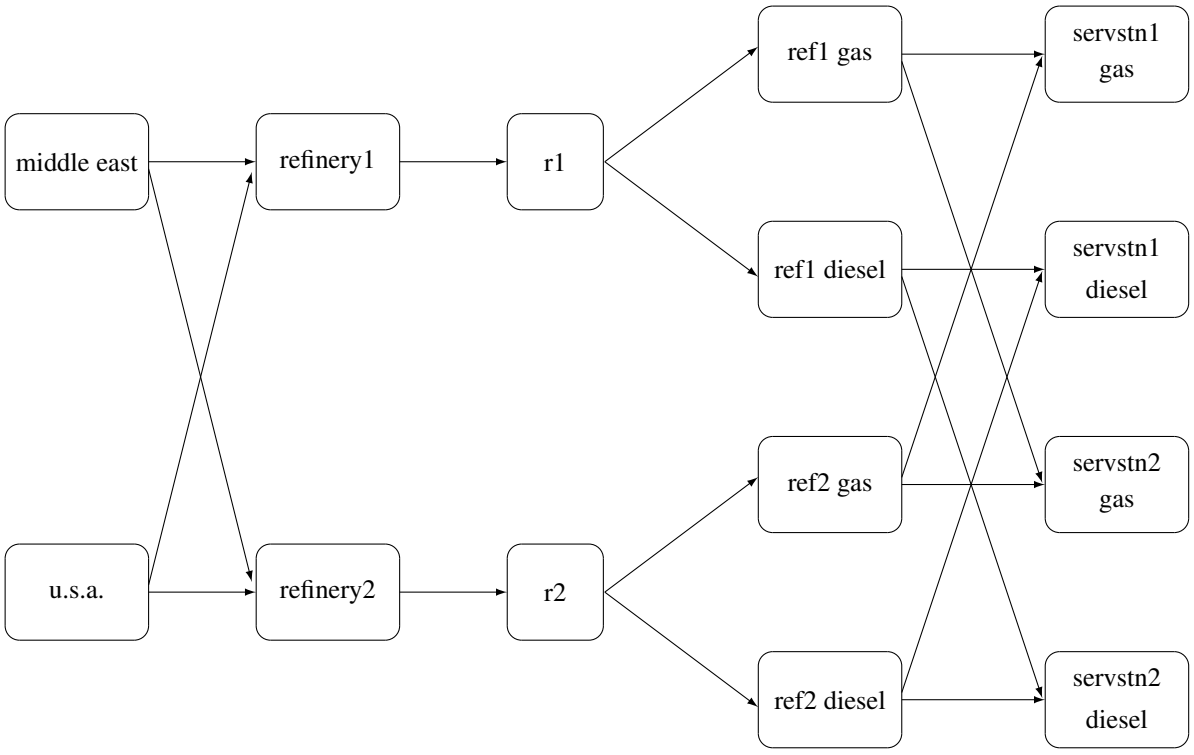
Introductory NPSC Example

Consider the following transshipment problem for an oil company. Crude oil is shipped to refineries where it is processed into gasoline and diesel fuel. The gasoline and diesel fuel are then distributed to service stations. At each stage, there are shipping, processing, and distribution costs. Also, there are lower flow bounds and capacities.

In addition, there are two sets of side constraints. The first set is that two times the crude from the Middle East cannot exceed the throughput of a refinery plus 15 units. (The phrase "plus 15 units" that finishes the last sentence is used to enable some side constraints in this example to have a nonzero rhs.) The second set of constraints are necessary to model the situation that one unit of crude mix processed at a refinery yields three-fourths of a unit of gasoline and one-fourth of a unit of diesel fuel.

Because there are two products that are not independent in the way in which they flow through the network, an NPSC is an appropriate model for this example (see Figure 4.6). The side constraints are used to model the limitations on the amount of Middle Eastern crude that can be processed by each refinery and the conversion proportions of crude to gasoline and diesel fuel.

Figure 4.6 Oil Industry Example



To solve this problem with PROC INTPOINT, save a representation of the model in three SAS data sets. In the NODEDATA= data set, you name the supply and demand nodes and give the associated supplies and demands. To distinguish demand nodes from supply nodes, specify demands as negative quantities. For the oil example, the NODEDATA= data set can be saved as follows:

```
title  'Oil Industry Example';
title3 'Setting Up Nodedata = Noded For PROC INTPOINT';
data noded;
   input   _node_&$15. _sd_;
   datalines;
middle east       100
u.s.a.             80
servstn1 gas      -95
servstn1 diesel   -30
servstn2 gas      -40
servstn2 diesel   -15
;
```

The ARCDATA= data set contains the rest of the information about the network. Each observation in the data set identifies an arc in the network and gives the cost per flow unit across the arc, the capacities of the arc, the lower bound on flow across the arc, and the name of the arc.

```
title3 'Setting Up Arcdata = Arcd1 For PROC INTPOINT';
data arcd1;
   input _from_&$11. _to_&$15. _cost_ _capac_ _lo_ _name_ $;
   datalines;
middle east    refinery 1        63     95   20    m_e_ref1
middle east    refinery 2        81     80   10    m_e_ref2
u.s.a.         refinery 1        55      .    .    .
u.s.a.         refinery 2        49      .    .    .
refinery 1     r1               200    175   50    thruput1
refinery 2     r2               220    100   35    thruput2
r1             ref1 gas           .    140    .    r1_gas
r1             ref1 diesel        .     75    .    .
r2             ref2 gas           .    100    .    r2_gas
r2             ref2 diesel        .     75    .    .
ref1 gas       servstn1 gas      15     70    .    .
ref1 gas       servstn2 gas      22     60    .    .
ref1 diesel    servstn1 diesel   18      .    .    .
ref1 diesel    servstn2 diesel   17      .    .    .
ref2 gas       servstn1 gas      17     35    5    .
ref2 gas       servstn2 gas      31      .    .    .
ref2 diesel    servstn1 diesel   36      .    .    .
ref2 diesel    servstn2 diesel   23      .    .    .
;
```

Finally, the CONDATA= data set contains the side constraints for the model:

```
title3 'Setting Up Condata = Cond1 For PROC INTPOINT';
data cond1;
   input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
         _type_ $ _rhs_;
   datalines;
-2  .  1 .  . . >= -15
 . -2  . .  1 . GE -15
 .  . -3 4  . . EQ   0
 .  .  . . -3 4  =   0
;
```

Note that the SAS variable names in the CONDATA= data set are the names of arcs given in the ARCDATA= data set. These are the arcs that have nonzero constraint coefficients in side constraints. For example, the proportionality constraint that specifies that one unit of crude at each refinery yields three-fourths of a unit of gasoline and one-fourth of a unit of diesel fuel is given for refinery 1 in the third observation and for refinery 2 in the last observation. The third observation requires that each unit of flow on the arc thruput1 equals three-fourths of a unit of flow on the arc r1_gas. Because all crude processed at refinery 1 flows through thruput1 and all gasoline produced at refinery 1 flows through r1_gas, the constraint models the situation. It proceeds similarly for refinery 2 in the last observation.

To find the minimum cost flow through the network that satisfies the supplies, demands, and side constraints, invoke PROC INTPOINT as follows:

```
proc intpoint
   bytes=1000000
   nodedata=noded        /* the supply and demand data */
   arcdata=arcd1         /* the arc descriptions       */
   condata=cond1         /* the side constraints       */
   conout=solution;      /* the solution data set      */
   run;
```

The following messages, which appear on the SAS log, summarize the model as read by PROC INTPOINT and note the progress toward a solution.

```
NOTE: Number of nodes= 14 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 180 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of side constraint coefficients= 8 .
NOTE: The following messages relate to the equivalent Linear Programming
      problem solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 16 .
NOTE: Number of >= constraints= 2 .
NOTE: Number of constraint coefficients= 44 .
NOTE: Number of variables= 18 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 3.
NOTE: After preprocessing, number of >= constraints= 2.
NOTE: The preprocessor eliminated 13 constraints from the problem.
NOTE: The preprocessor eliminated 33 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 5.
NOTE: The preprocessor eliminated 13 variables from the problem.
NOTE: 4 columns, 0 rows and 4 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 10 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 5 factor nodes make up 1 supernodes
NOTE: There are 0 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
NOTE: Bound feasibility attained by iteration 1.
NOTE: Dual feasibility attained by iteration 1.
NOTE: Constraint feasibility attained by iteration 1.
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 6
      iterations.
NOTE: Optimum reached.
NOTE: Objective= 50875.
NOTE: The data set WORK.SOLUTION has 18 observations and 10 variables.
NOTE: There were 18 observations read from the data set WORK.ARCD1.
NOTE: There were 6 observations read from the data set WORK.NODED.
NOTE: There were 4 observations read from the data set WORK.COND1.
```

The first set of messages shows the size of the problem. The next set of messages provides statistics on the size of the equivalent LP problem. The number of variables may not equal the number of arcs if the problem has nonarc variables. This example has none. To convert a network to the equivalent LP problem, a flow conservation constraint must be created for each node (including an excess or bypass node, if required). This explains why the number of equality constraints and the number of constraint coefficients differ from the number of equality side constraints and the number of coefficients in all side constraints.

If the preprocessor was successful in decreasing the problem size, some messages will report how well it did. In this example, the model size was cut approximately in half!

The next set of messages describes aspects of the interior point algorithm. Of particular interest are those concerned with the Cholesky factorization of AA^T where A is the coefficient matrix of the final LP. It is crucial to preorder the rows and columns of this matrix to prevent *fill-in* and reduce the number of row operations to undertake the factorization. See the section "Interior Point Algorithmic Details" on page 41 for a more extensive explanation.

Unlike PROC LP, which displays the solution and other information as output, PROC INTPOINT saves the optimum in the output SAS data set that you specify. For this example, the solution is saved in the SOLUTION data set. It can be displayed with the PRINT procedure as

```
title3 'Optimum';
proc print data=solution;
   var _from_ _to_ _cost_ _capac_ _lo_ _name_
       _supply_ _demand_ _flow_ _fcost_;
   sum _fcost_;
   run;
```

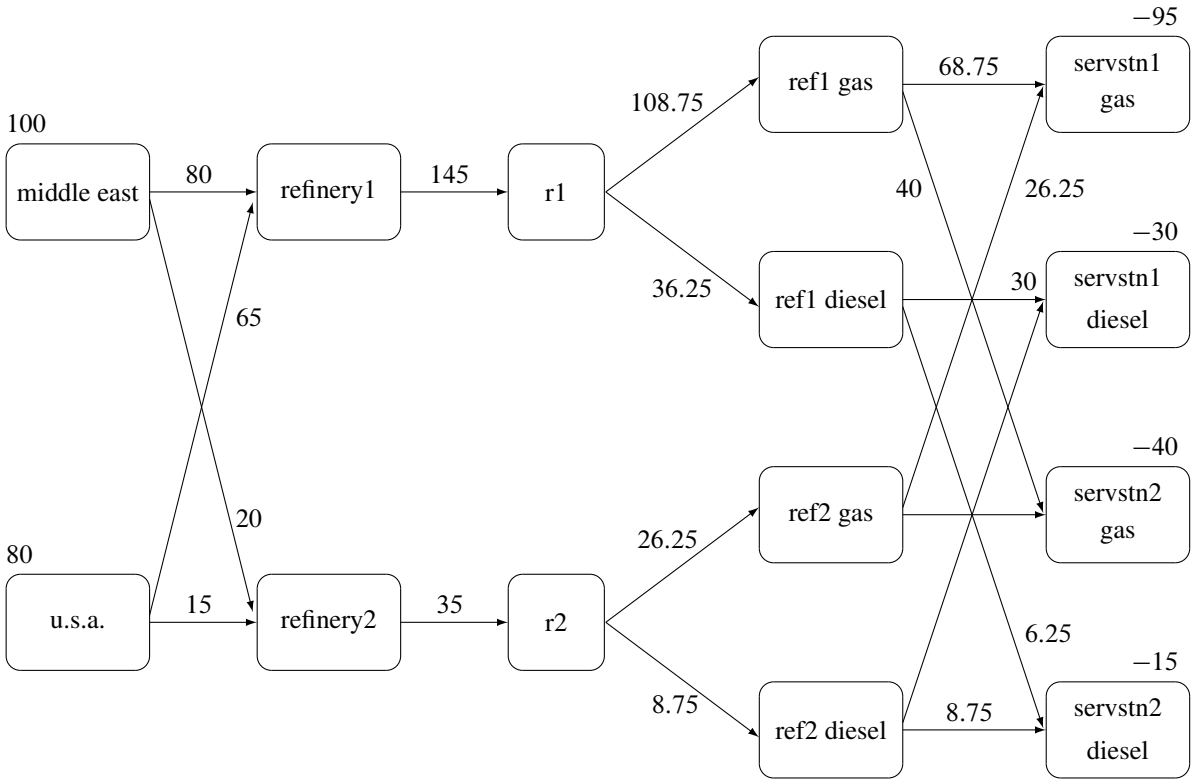
Figure 4.7 CONOUT=SOLUTION

Optimum

Obs	_from_	_to_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	refinery 1	r1	200	175	50	thruput1	.	.	145.000	29000.00
2	refinery 2	r2	220	100	35	thruput2	.	.	35.000	7700.00
3	r1	ref1 diesel	0	75	0		.	.	36.250	0.00
4	r1	ref1 gas	0	140	0	r1_gas	.	.	108.750	0.00
5	r2	ref2 diesel	0	75	0		.	.	8.750	0.00
6	r2	ref2 gas	0	100	0	r2_gas	.	.	26.250	0.00
7	middle east	refinery 1	63	95	20	m_e_ref1	100	.	80.000	5040.00
8	u.s.a.	refinery 1	55	99999999	0		80	.	65.000	3575.00
9	middle east	refinery 2	81	80	10	m_e_ref2	100	.	20.000	1620.00
10	u.s.a.	refinery 2	49	99999999	0		80	.	15.000	735.00
11	ref1 diesel	servstn1 diesel	18	99999999	0		.	30	30.000	540.00
12	ref2 diesel	servstn1 diesel	36	99999999	0		.	30	0.000	0.00
13	ref1 gas	servstn1 gas	15	70	0		.	95	68.750	1031.25
14	ref2 gas	servstn1 gas	17	35	5		.	95	26.250	446.25
15	ref1 diesel	servstn2 diesel	17	99999999	0		.	15	6.250	106.25
16	ref2 diesel	servstn2 diesel	23	99999999	0		.	15	8.750	201.25
17	ref1 gas	servstn2 gas	22	60	0		.	40	40.000	880.00
18	ref2 gas	servstn2 gas	31	99999999	0		.	40	0.000	0.00
										50875.00

Notice that, in CONOUT=SOLUTION (Figure 4.7), the optimal flow through each arc in the network is given in the variable named _FLOW_, and the cost of flow through each arc is given in the variable _FCOST_.

Figure 4.8 Oil Industry Solution



LP Problems

Data for an LP problem resembles the data for side constraints and nonarc variables supplied to PROC INTPOINT when solving an NPSC problem. It is also very similar to the data required by the LP procedure.

To solve LP problems using PROC INTPOINT, you save a representation of the LP variables and the constraints in one or two SAS data sets. These data sets are then passed to PROC INTPOINT for solution. There are various forms that a problem's data can take. You can use any one or a combination of several of these forms.

The ARCDATA= data set contains information about the LP variables of the problem. Although this data set is called ARCDATA, it contains data for no arcs. Instead, all data in this data set are related to LP variables. This data set has no SAS variables containing values that are node names.

The ARCDATA= data set can be used to specify information about LP variables, including objective function coefficients, lower and upper value bounds, and names. These data are the elements of the vectors d, m, and v in problem (LP). Data for an LP variable can be given in more than one observation.

The CONDATA= data set describes the constraints and their right-hand sides. These data are elements of the matrix Q and the vector r.

Constraint types are also specified in the CONDATA= data set. You can include in this data set LP variable data such as upper bound values, lower value bounds, and objective function coefficients. It is possible to give all information about some or all LP variables in the CONDATA= data set.

Because PROC INTPOINT evolved from PROC NETFLOW, another procedure in SAS/OR software that was originally designed to solve models with networks, the ARCDATA= data set is always expected. If the ARCDATA= data set is not specified, by default the last data set created before PROC INTPOINT is invoked is assumed to be the ARCDATA= data set. However, these characteristics of PROC INTPOINT are not helpful when an LP problem is being solved and all data are provided in a single data set specified by the CONDATA= data set, and that data set is not the last data set created before PROC INTPOINT starts. In this case, you must specify that the ARCDATA= data set and the CONDATA= data set are both equal to the input data set. PROC INTPOINT then knows that an LP problem is to be solved and that the data reside in one data set.

An LP variable is identified in this data set by its name. If you specify an LP variable's name in the ARCDATA= data set, then this name is used to associate data in the CONDATA= data set with that LP variable.

If you use the dense constraint input format (described in the section "CONDATA= Data Set" on page 101), these LP variable names are names of SAS variables in the VAR list of the CONDATA= data set.

If you use the sparse constraint input format (described in the section "CONDATA= Data Set" on page 101), these LP variable names are values of the SAS variables in the COLUMN list of the CONDATA= data set.

PROC INTPOINT reads the data from the ARCDATA= data set (if there is one) and the CONDATA= data set. Error checking is performed, and the LP is preprocessed. Preprocessing is optional but highly recommended. The preprocessor analyzes the model and tries to determine before optimization whether LP variables can be "fixed" to their optimal values. Knowing that, the model can be modified and these LP variables dropped out. Some constraints may be found to be redundant. Sometimes, preprocessing succeeds in reducing the size of the problem, thereby making the subsequent optimization easier and faster.

The optimal solution is then found for the resulting LP. If the problem was preprocessed, the model is now post-processed, where fixed LP variables are reintroduced. The solution can be saved in the CONOUT= data set.

Introductory LP Example

Consider the linear programming problem in the section "An Introductory Example" on page 171. The SAS data set in that section is created the same way here:

```
title  'Linear Programming Example';
title3 'Setting Up Condata = dcon1 For PROC INTPOINT';
data dcon1;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2
         _type_ $ _rhs_;
   datalines;
profit            -175 -165 -205  0  0  0 300 300 max      .
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq      0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq      0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq      0
recipe_1             0    0    0  0 .3 .7  -1   0  eq      0
recipe_2             0    0    0 .2  0 .8   0  -1  eq      0
available          110  165   80  .  .  .   .   . upperbd .
;
```

To solve this problem, use

```
proc intpoint
   bytes=1000000
   condata=dcon1
   conout=solutn1;
   run;
```

Note how it is possible to use an input SAS data set of PROC LP and, without requiring any changes to be made to the data set, to use that as an input data set for PROC INTPOINT.

The following messages that appear on the SAS log summarize the model as read by PROC INTPOINT and note the progress toward a solution

```
NOTE: Number of variables= 8 .
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 5 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 18 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 0.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 5 constraints from the problem.
NOTE: The preprocessor eliminated 18 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 0.
NOTE: The preprocessor eliminated 8 variables from the problem.
NOTE: The optimum has been determined by the Preprocessor.
NOTE: Objective= 1544.
NOTE: The data set WORK.SOLUTN1 has 8 observations and 6 variables.
NOTE: There were 7 observations read from the data set WORK.DCON1.
```

Notice that the preprocessor succeeded in fixing *all* LP variables to their optimal values, eliminating the need to do any actual optimization.

Unlike PROC LP, which displays the solution and other information as output, PROC INTPOINT saves the optimum in the output SAS data set you specify. For this example, the solution is saved in the SOLUTION data set. It can be displayed with PROC PRINT as

```
title3 'LP Optimum';
proc print data=solutn1;
   var _name_ _objfn_ _upperbd _lowerbd _value_ _fcost_;
   sum _fcost_;
   run;
```

Notice that in the CONOUT=SOLUTION (Figure 4.9) the optimal value through each variable in the LP is given in the variable named _VALUE_, and that the cost of value for each variable is given in the variable _FCOST_.

Figure 4.9 CONOUT=SOLUTN1

LP Optimum

Obs	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_FCOST_
1	a_heavy	-165	165	0	0.00	0
2	a_light	-175	110	0	110.00	-19250
3	brega	-205	80	0	80.00	-16400
4	heatingo	0	99999999	0	77.30	0
5	jet_1	300	99999999	0	60.65	18195
6	jet_2	300	99999999	0	63.33	18999
7	naphthai	0	99999999	0	21.80	0
8	naphthal	0	99999999	0	7.45	0
						1544

The same model can be specified in the sparse format as in the following scon2 data set. This format enables you to omit the zero coefficients.

```
title3 'Setting Up Condata = scon2 For PROC INTPOINT';
data scon2;
   format _type_ $8. _col_ $8. _row_ $16.;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
max       .           profit                .
eq        .           napha_l_conv          .
eq        .           napha_i_conv          .
eq        .           heating_oil_conv      .
eq        .           recipe_1              .
eq        .           recipe_2              .
upperbd   .           available             .
.         a_light     profit             -175
.         a_light     napha_l_conv       .035
.         a_light     napha_i_conv       .100
.         a_light     heating_oil_conv   .390
.         a_light     available           110
.         a_heavy     profit             -165
.         a_heavy     napha_l_conv       .030
.         a_heavy     napha_i_conv       .075
.         a_heavy     heating_oil_conv   .300
.         a_heavy     available           165
.         brega       profit             -205
.         brega       napha_l_conv       .045
.         brega       napha_i_conv       .135
.         brega       heating_oil_conv   .430
.         brega       available           80
.         naphthal    napha_l_conv        -1
.         naphthal    recipe_2            .2
.         naphthai    napha_i_conv        -1
.         naphthai    recipe_1            .3
.         heatingo    heating_oil_conv    -1
.         heatingo    recipe_1            .7
.         heatingo    recipe_2            .8
.         jet_1       profit             300
```

```
.         jet_1          recipe_1                 -1
.         jet_2          profit                  300
.         jet_2          recipe_2                 -1
;
```

To find the minimum cost solution, invoke PROC INTPOINT (note the SPARSECONDATA option which must be specified) as follows:

```
proc intpoint
   bytes=1000000
   sparsecondata
   condata=scon2
   conout=solutn2;
   run;
```

A data set that can be used as the ARCDATA= data set can be initialized as follows:

```
data vars3;
   input _name_ $ profit available;
   datalines;
a_heavy  -165 165
a_light  -175 110
brega    -205  80
heatingo    0   .
jet_1     300   .
jet_2     300   .
naphthai    0   .
naphthal    0   .
;
```

The following CONDATA= data set is the original dense format CONDATA= dcon1 data set after the LP variable's nonconstraint information has been removed. (You could have left some or all of that information in CONDATA as PROC INTPOINT "merges" data, but doing that and checking for consistency takes time.)

```
data dcon3;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2
         _type_ $ _rhs_;
   datalines;
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq     0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq     0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq     0
recipe_1             0    0    0  0 .3 .7  -1   0  eq     0
recipe_2             0    0    0 .2  0 .8   0  -1  eq     0
;
```

NOTE: You must now specify the MAXIMIZE option; otherwise, PROC INTPOINT will optimize to the minimum (which, incidentally, has a total objective = -3539.25). You must indicate that the SAS variable profit in the ARCDATA=vars3 data set has values that are objective function coefficients, by specifying the OBJFN statement. The UPPERBD must be specified as the SAS variable available that has as values upper bounds:

```
proc intpoint
     maximize        /* ***** necessary ***** */
     bytes=1000000
     arcdata=vars3
     condata=dcon3
     conout=solutn3;
   objfn profit;
   upperbd available;
   run;
```

The ARCDATA=vars3 data set can become more concise by noting that the model variables heatingo, naphthai, and naphthal have zero objective function coefficients (the default) and default upper bounds, so those observations need not be present:

```
data vars4;
   input _name_ $ profit available;
   datalines;
a_heavy  -165 165
a_light  -175 110
brega    -205  80
jet_1     300   .
jet_2     300   .
;
```

The CONDATA=dcon3 data set can become more concise by noting that all the constraints have the same type (eq) and zero (the default) rhs values. This model is a good candidate for using the DEFCONTYPE= option.

The DEFCONTYPE= option can be useful not only when *all* constraints have the same type as is the case here, but also when *most* constraints have the same type and you want to change the default type from $\leq$ to $=$ or $\geq$. The essential constraint type data in the CONDATA= data set is that which overrides the DEFCONTYPE= type you specified.

```
data dcon4;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2;
   datalines;
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0
heating_o_conv    .390 .300 .430  0  0 -1   0   0
recipe_1             0    0    0  0 .3 .7  -1   0
recipe_2             0    0    0 .2  0 .8   0  -1
;

proc intpoint
     maximize defcontype=eq
     bytes=1000000
     arcdata=vars3
     condata=dcon3
     conout=solutn3;
   objfn profit;
   upperbd available;
   run;
```

Here are several different ways of using the ARCDATA= data set and a sparse format CONDATA= data set for this LP. The following CONDATA= data set is the result of removing the profit and available data from the original sparse format CONDATA=scon2 data set.

```
data scon5;
   format _type_ $8. _col_ $8. _row_ $16. ;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
eq       .             napha_l_conv              .
eq       .             napha_i_conv              .
eq       .             heating_oil_conv          .
eq       .             recipe_1                  .
eq       .             recipe_2                  .
.        a_light       napha_l_conv           .035
.        a_light       napha_i_conv           .100
.        a_light       heating_oil_conv       .390
.        a_heavy       napha_l_conv           .030
.        a_heavy       napha_i_conv           .075
.        a_heavy       heating_oil_conv       .300
.        brega         napha_l_conv           .045
.        brega         napha_i_conv           .135
.        brega         heating_oil_conv       .430
.        naphthal      napha_l_conv             -1
.        naphthal      recipe_2                  .2
.        naphthai      napha_i_conv             -1
.        naphthai      recipe_1                  .3
.        heatingo      heating_oil_conv         -1
.        heatingo      recipe_1                  .7
.        heatingo      recipe_2                  .8
.        jet_1         recipe_1                 -1
.        jet_2         recipe_2                 -1
;

proc intpoint
     maximize
     bytes=1000000
     sparsecondata
     arcdata=vars3      /* or arcdata=vars4 */
     condata=scon5
     conout=solutn5;
   objfn profit;
   upperbd available;
   run;
```

The CONDATA=scon5 data set can become more concise by noting that all the constraints have the same type (eq) and zero (the default) rhs values. Use the DEFCONTYPE= option again. Once the first five observations of the CONDATA=scon5 data set are removed, the _type_ variable has values that are missing in all of the remaining observations. Therefore, this variable can be removed.

```
data scon6;
   input _col_ $ _row_&$16. _coef_;
   datalines;
a_light  napha_l_conv           .035
a_light  napha_i_conv           .100
```

```
a_light  heating_oil_conv       .390
a_heavy  napha_l_conv           .030
a_heavy  napha_i_conv           .075
a_heavy  heating_oil_conv       .300
brega    napha_l_conv           .045
brega    napha_i_conv           .135
brega    heating_oil_conv       .430
naphthal napha_l_conv            -1
naphthal recipe_2                 .2
naphthai napha_i_conv            -1
naphthai recipe_1                 .3
heatingo heating_oil_conv        -1
heatingo recipe_1                 .7
heatingo recipe_2                 .8
jet_1    recipe_1                -1
jet_2    recipe_2                -1
;

proc intpoint
     maximize
     bytes=1000000
     defcontype=eq
     sparsecondata
     arcdata=vars4
     condata=scon6
     conout=solutn6;
   objfn profit;
   upperbd available;
   run;
```

Typical PROC INTPOINT Run

You start PROC INTPOINT by giving the PROC INTPOINT statement. You can specify many options in the PROC INTPOINT statement to control the procedure, or you can rely on default settings and specify very few options. However, there are some options you must specify:

- You must specify the BYTES= parameter indicating the size of the working memory that the procedure is allowed to use. This option has no default.
- In many instances (and certainly when solving NPSC problems), you need to specify the ARCDATA= data set. This option has a default (which is the SAS data set that was created last before PROC INTPOINT began running), but that may need to be overridden.
- The CONDATA= data set must also be specified if the problem is NPSC and has side constraints, or if it is an LP problem.
- When solving a network problem, you have to specify the NODEDATA= data set, if some model data are given in such a data set.

Some options, while optional, are frequently required. To have the optimal solution output to a SAS data set, you have to specify the CONOUT= data set. You may want to indicate reasons why optimization should stop

(for example, you can indicate the maximum number of iterations that can be performed), or you might want to alter stopping criteria so that optimization does not stop prematurely. Some options enable you to control other aspects of the interior point algorithm. Specifying certain values for these options can reduce the time it takes to solve a problem.

The SAS variable lists should be given next. If you have SAS variables in the input data sets that have special names (for example, a SAS variable in the ARCDATA= data set named _TAIL_ that has tail nodes of arcs as values), it may not be necessary to have many or any variable lists. If you do not specify a TAIL variable list, PROC INTPOINT will search the ARCDATA= data set for a SAS variable named _TAIL_.

What usually follows is a RUN statement, which indicates that all information that you, the user, need to supply to PROC INTPOINT has been given, and the procedure is to start running. This also happens if you specify a statement in your SAS program that PROC INTPOINT does not recognize as one of its own, the next DATA step or procedure.

The QUIT statement indicates that PROC INTPOINT must immediately finish.

For example, a PROC INTPOINT run might look something like this:

```
proc intpoint
    bytes=   /* working memory size */
    arcdata= /* data set */
    condata= /* data set */
    /* other options */
;
variable list specifications;  /* if necessary */
run;     /* start running, read data, */
         /* and do the optimization.  */
```

Syntax: INTPOINT Procedure

Below are statements used in PROC INTPOINT, listed in alphabetical order as they appear in the text that follows.

PROC INTPOINT *options* ;
CAPACITY *variable* ;
COEF *variables* ;
COLUMN *variable* ;
COST *variable* ;
DEMAND *variable* ;
HEADNODE *variable* ;
ID *variables* ;
LO *variable* ;
NAME *variable* ;
NODE *variable* ;
QUIT *;* ;
RHS *variable* ;
ROW *variables* ;
RUN *;* ;
SUPDEM *variable* ;
SUPPLY *variable* ;
TAILNODE *variable* ;
TYPE *variable* ;
VAR *variables* ;

Functional Summary

Table 4.1 outlines the options that can be specified in the INTPOINT procedure. All options are specified in the PROC INTPOINT statement.

Table 4.1 Functional Summary

Description	Statement	Option
Input Data Set Options:		
Arcs input data set	PROC INTPOINT	ARCDATA=
Nodes input data set	PROC INTPOINT	NODEDATA=
Constraint input data set	PROC INTPOINT	CONDATA=
Output Data Set Options:		
Constrained solution data set	PROC INTPOINT	CONOUT=
Convert sparse or dense format input data set into MPS-format output data set	PROC INTPOINT	MPSOUT=
Data Set Read Options:		
CONDATA has sparse data format	PROC INTPOINT	SPARSECONDATA

Description	Statement	Option
Default constraint type	PROC INTPOINT	DEFCONTYPE=
Special COLUMN variable value	PROC INTPOINT	TYPEOBS=
Special COLUMN variable value	PROC INTPOINT	RHSOBS=
Used to interpret arc and variable names	PROC INTPOINT	NAMECTRL=
No nonarc data in ARCDATA	PROC INTPOINT	ARCS_ONLY_ARCDATA
Data for an arc found once in ARCDATA	PROC INTPOINT	ARC_SINGLE_OBS
Data for a constraint found once in CONDATA	PROC INTPOINT	CON_SINGLE_OBS
Data for a coefficient found once in CONDATA	PROC INTPOINT	NON_REPLIC=
Data are grouped, exploited during data read	PROC INTPOINT	GROUPED=
Problem Size Specification Options:		
Approximate number of nodes	PROC INTPOINT	NNODES=
Approximate number of arcs	PROC INTPOINT	NARCS=
Approximate number of variables	PROC INTPOINT	NNAS=
Approximate number of coefficients	PROC INTPOINT	NCOEFS=
Approximate number of constraints	PROC INTPOINT	NCONS=
Network Options:		
Default arc cost, objective function coefficient	PROC INTPOINT	DEFCOST=
Default arc capacity, variable upper bound	PROC INTPOINT	DEFCAPACITY=
Default arc flow and variable lower bound	PROC INTPOINT	DEFMINFLOW=
Network's only supply node	PROC INTPOINT	SOURCE=
SOURCE's supply capability	PROC INTPOINT	SUPPLY=
Network's only demand node	PROC INTPOINT	SINK=
SINK's demand	PROC INTPOINT	DEMAND=
Convey excess supply/demand through network	PROC INTPOINT	THRUNET
Find max flow between SOURCE and SINK	PROC INTPOINT	MAXFLOW
Cost of bypass arc, MAXFLOW problem	PROC INTPOINT	BYPASSDIVIDE=
Find shortest path from SOURCE to SINK	PROC INTPOINT	SHORTPATH
Interior Point Algorithm Options:		
Factorization method	PROC INTPOINT	FACT_METHOD=
Allowed amount of dual infeasibility	PROC INTPOINT	TOLDINF=
Allowed amount of primal infeasibility	PROC INTPOINT	TOLPINF=
Allowed total amount of dual infeasibility	PROC INTPOINT	TOLTOTDINF=
Allowed total amount of primal infeasibility	PROC INTPOINT	TOLTOTPINF=
Cut-off tolerance for Cholesky factorization	PROC INTPOINT	CHOLTINYTOL=
Density threshold for Cholesky processing	PROC INTPOINT	DENSETHR=
Step-length multiplier	PROC INTPOINT	PDSTEPMULT=
Preprocessing type	PROC INTPOINT	PRSLTYPE=
Print optimization progress on SAS log	PROC INTPOINT	PRINTLEVEL2=
Ratio test zero tolerance	PROC INTPOINT	RTTOL=

Description	Statement	Option
Interior Point Algorithm Stopping Criteria:		
maximum number of interior point iterations	PROC INTPOINT	MAXITERB=
primal-dual (duality) gap tolerance	PROC INTPOINT	PDGAPTOL=
Stop because of complementarity	PROC INTPOINT	STOP_C=
Stop because of duality gap	PROC INTPOINT	STOP_DG=
Stop because of $infeas_b$	PROC INTPOINT	STOP_IB=
Stop because of $infeas_c$	PROC INTPOINT	STOP_IC=
Stop because of $infeas_d$	PROC INTPOINT	STOP_ID=
Stop because of complementarity	PROC INTPOINT	AND_STOP_C=
Stop because of duality gap	PROC INTPOINT	AND_STOP_DG=
Stop because of $infeas_b$	PROC INTPOINT	AND_STOP_IB=
Stop because of $infeas_c$	PROC INTPOINT	AND_STOP_IC=
Stop because of $infeas_d$	PROC INTPOINT	AND_STOP_ID=
Stop because of complementarity	PROC INTPOINT	KEEPGOING_C=
Stop because of duality gap	PROC INTPOINT	KEEPGOING_DG=
Stop because of $infeas_b$	PROC INTPOINT	KEEPGOING_IB=
Stop because of $infeas_c$	PROC INTPOINT	KEEPGOING_IC=
Stop because of $infeas_d$	PROC INTPOINT	KEEPGOING_ID=
Stop because of complementarity	PROC INTPOINT	AND_KEEPGOING_C=
Stop because of duality gap	PROC INTPOINT	AND_KEEPGOING_DG=
Stop because of $infeas_b$	PROC INTPOINT	AND_KEEPGOING_IB=
Stop because of $infeas_c$	PROC INTPOINT	AND_KEEPGOING_IC=
Stop because of $infeas_d$	PROC INTPOINT	AND_KEEPGOING_ID=
Memory Control Options:		
Issue memory usage messages to SAS log	PROC INTPOINT	MEMREP
Number of bytes to use for main memory	PROC INTPOINT	BYTES=
Miscellaneous Options:		
Infinity value	PROC INTPOINT	INFINITY=
Maximization instead of minimization	PROC INTPOINT	MAXIMIZE
Zero tolerance - optimization	PROC INTPOINT	ZERO2=
Zero tolerance - real number comparisons	PROC INTPOINT	ZEROTOL=
Suppress similar SAS log messages	PROC INTPOINT	VERBOSE=
Scale problem data	PROC INTPOINT	SCALE=
Write optimization time to SAS log	PROC INTPOINT	OPTIM_TIMER

PROC INTPOINT Statement

PROC INTPOINT *options* **;**

This statement invokes the procedure. The following options can be specified in the PROC INTPOINT statement.

Data Set Options

This section briefly describes all the input and output data sets used by PROC INTPOINT. The ARCDATA= data set, the NODEDATA= data set, and the CONDATA= data set can contain SAS variables that have special names, for instance _CAPAC_, _COST_, and _HEAD_. PROC INTPOINT looks for such variables if you do not give explicit variable list specifications. If a SAS variable with a special name is found and that SAS variable is not in another variable list specification, PROC INTPOINT determines that values of the SAS variable are to be interpreted in a special way. By using SAS variables that have special names, you may not need to have any variable list specifications.

ARCDATA=*SAS-data-set*
names the data set that contains arc and, optionally, nonarc variable information and nodal supply/demand data. The ARCDATA= data set must be specified in all PROC INTPOINT statements when solving NPSC problems.

If your problem is an LP, the ARCDATA= data set is optional. You can specify LP variable information such as objective function coefficients, and lower and upper bounds.

CONDATA=*SAS-data-set*
names the data set that contains the side constraint data. The data set can also contain other data such as arc costs, capacities, lower flow bounds, nonarc variable upper and lower bounds, and objective function coefficients. PROC INTPOINT needs a CONDATA= data set to solve a constrained problem. See the section "CONDATA= Data Set" on page 101 for more information.

If your problem is an LP, this data set contains the constraint data, and can also contain other data such as objective function coefficients, and lower and upper bounds. PROC INTPOINT needs a CONDATA= data set to solve an LP.

CONOUT=*SAS-data-set*
COUT=*SAS-data-set*
names the output data set that receives an optimal solution. See the section "CONOUT= Data Set" on page 109 for more information.

If PROC INTPOINT is outputting observations to the output data set and you want this to stop, press the keys used to stop SAS procedures.

MPSOUT=*SAS-data-set*
names the SAS data set that contains converted sparse or dense format input data in MPS format. Invoking this option directs the INTPOINT procedure to halt before attempting optimization. For more information about the MPSOUT= option, see the section "Converting Any PROC INTPOINT Format to an MPS-Format SAS Data Set" on page 111. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

NODEDATA=*SAS-data-set*
names the data set that contains the node supply and demand specifications. You do not need observations in the NODEDATA= data set for transshipment nodes. (Transshipment nodes neither supply nor demand flow.) All nodes are assumed to be transshipment nodes unless supply or demand data indicate otherwise. It is acceptable for some arcs to be directed toward supply nodes or away from demand nodes.

This data set is used only when you are solving network problems (not when solving LP problems), in which case the use of the NODEDATA= data set is optional provided that, if the NODEDATA= data set is not used, supply and demand details are specified by other means. Other means include using the MAXFLOW or SHORTPATH option, SUPPLY or DEMAND variable list (or both) in the ARCDATA= data set, and the SOURCE=, SUPPLY=, SINK=, or DEMAND= option in the PROC INTPOINT statement.

General Options

The following is a list of options you can use with PROC INTPOINT. The options are listed in alphabetical order.

ARCS_ONLY_ARCDATA

indicates that data for arcs only are in the ARCDATA= data set. When PROC INTPOINT reads the data in the ARCDATA= data set, memory would not be wasted to receive data for nonarc variables. The read might then be performed faster. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

ARC_SINGLE_OBS

indicates that for all arcs and nonarc variables, data for each arc or nonarc variable is found in only one observation of the ARCDATA= data set. When reading the data in the ARCDATA= data set, PROC INTPOINT knows that the data in an observation is for an arc or a nonarc variable that has not had data previously read and that needs to be checked for consistency. The read might then be performed faster.

When solving an LP, specifying the ARC_SINGLE_OBS option indicates that for all LP variables, data for each LP variable is found in only one observation of the ARCDATA= data set. When reading the data in the ARCDATA= data set, PROC INTPOINT knows that the data in an observation is for an LP variable that has not had data previously read and that needs to be checked for consistency. The read might then be performed faster.

If you specify ARC_SINGLE_OBS, PROC INTPOINT automatically works as if GROUPED=ARCDATA is also specified.

See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

BYPASSDIVIDE=*b*

BYPASSDIV=*b*

BPD=*b*

should be used only when the MAXFLOW option has been specified; that is, PROC INTPOINT is solving a maximal flow problem. PROC INTPOINT prepares to solve maximal flow problems by setting up a bypass arc. This arc is directed from the SOURCE= to the SINK= and will eventually convey flow equal to INFINITY minus the maximal flow through the network. The cost of the bypass arc must be great enough to drive flow through the network, rather than through the bypass arc. Also, the cost of the bypass arc must be greater than the eventual total cost of the maximal flow, which can be nonzero if some network arcs have nonzero costs. The cost of the bypass is set to the value of the INFINITY= option. Valid values for the BYPASSDIVIDE= option must be greater than or equal to 1.1.

If there are no nonzero costs of arcs in the MAXFLOW problem, the cost of the bypass arc is set to 1.0 (-1.0 if maximizing) if you do not specify the BYPASSDIVIDE= option. The default value for the BYPASSDIVIDE= option (in the presence of nonzero arc costs) is 100.0.

BYTES=*b*

indicates the size of the main working memory (in bytes) that PROC INTPOINT will allocate. Specifying this option is mandatory. The working memory is used to store all the arrays and buffers used by PROC INTPOINT. If this memory has a size smaller than what is required to store all arrays and buffers, PROC INTPOINT uses various schemes that page information between auxiliary memory (often your machine's disk) and RAM.

For small problems, specify BYTES=100000. For large problems (those with hundreds of thousands or millions of variables), BYTES=1000000 might do. For solving problems of that size, if you are running on a machine with an inadequate amount of RAM, PROC INTPOINT's performance will suffer since it will be forced to page or to rely on virtual memory.

If you specify the MEMREP option, PROC INTPOINT will issue messages on the SAS log informing you of its memory usage; that is, how much memory is required to prevent paging, and details about the amount of paging that must be performed, if applicable.

CON_SINGLE_OBS

improves how the CONDATA= data set is read. How it works depends on whether the CONDATA has a dense or sparse format.

If the CONDATA= data set has the dense format, specifying CON_SINGLE_OBS indicates that, for each constraint, data for each can be found in only one observation of the CONDATA= data set.

If the CONDATA= data set has a sparse format, and data for each arc, nonarc variable, or LP variable can be found in only one observation of the CONDATA, then specify the CON_SINGLE_OBS option. If there are *n* SAS variables in the ROW and COEF list, then each arc or nonarc can have at most *n* constraint coefficients in the model. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

DEFCAPACITY=*c*

DC=*c*

requests that the default arc capacity and the default nonarc variable value upper bound (or for LP problems, the default LP variable value upper bound) be *c*. If this option is not specified, then DEFCAPACITY= INFINITY.

DEFCONTYPE=*c*

DEFTYPE=*c*

DCT=*c*

specifies the default constraint type. This default constraint type is either *less than or equal to* or is the type indicated by DEFCONTYPE=*c*. Valid values for this option are

LE, le, or <=	for *less than or equal to*
EQ, eq, or =	for *equal to*
GE, ge, or >=	for *greater than or equal to*

The values do not need to be enclosed in quotes.

DEFCOST=*c*

requests that the default arc cost and the default nonarc variable objective function coefficient (or for an LP, the default LP variable objective function coefficient) be *c*. If this option is not specified, then DEFCOST=0.0.

DEFMINFLOW=*m*

DMF=*m*

requests that the default lower flow bound through arcs and the default lower value bound of nonarc variables (or for an LP, the default lower value bound of LP variables) be *m*. If a value is not specified, then DEFMINFLOW=0.0.

DEMAND=*d*

specifies the demand at the SINK node specified by the SINK= option. The DEMAND= option should be used only if the SINK= option is given in the PROC INTPOINT statement and neither the SHORTPATH option nor the MAXFLOW option is specified. If you are solving a minimum cost network problem and the SINK= option is used to identify the sink node, and the DEMAND= option is not specified, then the demand at the sink node is made equal to the network's total supply.

GROUPED=*grouped*

PROC INTPOINT can take a much shorter time to read data if the data have been grouped prior to the PROC INTPOINT call. This enables PROC INTPOINT to conclude that, for instance, a new NAME list variable value seen in the ARCDATA= data set grouped by the values of the NAME list variable before PROC INTPOINT was called is new. PROC INTPOINT does not need to check that the NAME has been read in a previous observation. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

- GROUPED=ARCDATA indicates that the ARCDATA= data set has been grouped by values of the NAME list variable. If _NAME_ is the name of the NAME list variable, you could use

  ```
  proc sort data=arcdata; by _name_;
  ```

  prior to calling PROC INTPOINT. Technically, you do not have to sort the data, only to ensure that all similar values of the NAME list variable are grouped together. If you specify the ARCS_ONLY_ARCDATA option, PROC INTPOINT automatically works as if GROUPED=ARCDATA is also specified.

- GROUPED=CONDATA indicates that the CONDATA= data set has been grouped.

  If the CONDATA= data set has a dense format, GROUPED=CONDATA indicates that the CONDATA= data set has been grouped by values of the ROW list variable. If _ROW_ is the name of the ROW list variable, you could use

  ```
  proc sort data=condata; by _row_;
  ```

  prior to calling PROC INTPOINT. Technically, you do not have to sort the data, only to ensure that all similar values of the ROW list variable are grouped together. If you specify the CON_SINGLE_OBS option, or if there is no ROW list variable, PROC INTPOINT automatically works as if GROUPED=CONDATA has been specified.

If the CONDATA= data set has the sparse format, GROUPED=CONDATA indicates that CONDATA has been grouped by values of the COLUMN list variable. If _COL_ is the name of the COLUMN list variable, you could use

```
proc sort data=condata; by _col_;
```

prior to calling PROC INTPOINT. Technically, you do not have to sort the data, only to ensure that all similar values of the COLUMN list variable are grouped together.

- GROUPED=BOTH indicates that both GROUPED=ARCDATA and GROUPED=CONDATA are TRUE.
- GROUPED=NONE indicates that the data sets have not been grouped, that is, neither GROUPED=ARCDATA nor GROUPED=CONDATA is TRUE. This is the default, but it is much better if GROUPED=ARCDATA, or GROUPED=CONDATA, or GROUPED=BOTH.

A data set like

```
... _XXXXX_ ....
      bbb
      bbb
      aaa
      ccc
      ccc
```

is a candidate for the GROUPED= option. Similar values are grouped together. When PROC INTPOINT is reading the ith observation, either the value of the _XXXXX_ variable is the same as the $(i-1)$st (that is, the previous observation's) _XXXXX_ value, or it is a new _XXXXX_ value not seen in any previous observation. This also means that if the ith _XXXXX_ value is different from the $(i-1)$st _XXXXX_ value, the value of the $(i-1)$st _XXXXX_ variable will not be seen in any observations $i, i+1, \ldots$.

INFINITY=*i*

INF=*i*

is the largest number used by PROC INTPOINT in computations. A number too small can adversely affect the solution process. You should avoid specifying an enormous value for the INFINITY= option because numerical roundoff errors can result. If a value is not specified, then INFINITY=99999999. The INFINITY= option cannot be assigned a value less than 9999.

MAXFLOW

MF

specifies that PROC INTPOINT solve a maximum flow problem. In this case, the PROC INTPOINT procedure finds the maximum flow from the node specified by the SOURCE= option to the node specified by the SINK= option. PROC INTPOINT automatically assigns an INFINITY= option supply to the SOURCE= option node and the SINK= option is assigned the INFINITY= option demand. In this way, the MAXFLOW option sets up a maximum flow problem as an equivalent minimum cost problem.

You can use the MAXFLOW option when solving any flow problem (not necessarily a maximum flow problem) when the network has one supply node (with infinite supply) and one demand node (with infinite demand). The MAXFLOW option can be used in conjunction with all other options (except SHORTPATH, SUPPLY=, and DEMAND=) and capabilities of PROC INTPOINT.

MAXIMIZE

MAX

specifies that PROC INTPOINT find the maximum cost flow through the network. If both the MAXIMIZE and the SHORTPATH options are specified, the solution obtained is the longest path between the SOURCE= and SINK= nodes. Similarly, MAXIMIZE and MAXFLOW together cause PROC INTPOINT to find the minimum flow between these two nodes; this is zero if there are no nonzero lower flow bounds. If solving an LP, specifying the MAXIMIZE option is necessary if you want the maximal optimal solution found instead of the minimal optimum.

MEMREP

indicates that information on the memory usage and paging schemes (if necessary) is reported by PROC INTPOINT on the SAS log.

NAMECTRL=*i*

is used to interpret arc and nonarc variable names in the CONDATA= data set. In the ARCDATA= data set, an arc is identified by its tail and head node. In the CONDATA= data set, arcs are identified by names. You can give a name to an arc by having a NAME list specification that indicates a SAS variable in the ARCDATA= data set that has names of arcs as values.

PROC INTPOINT requires that arcs that have information about them in the CONDATA= data set have names, but arcs that do not have information about them in the CONDATA= data set can also have names. Unlike a nonarc variable whose name uniquely identifies it, an arc can have several different names. An arc has a default name in the form *tail_head*, that is, the name of the arc's tail node followed by an underscore and the name of the arc's head node.

In the CONDATA= data set, if the dense data format is used (described in the section "CONDATA= Data Set" on page 101), a name of an arc or a nonarc variable is the *name* of a SAS variable listed in the VAR list specification. If the sparse data format of the CONDATA= data set is used, a name of an arc or a nonarc variable is a *value* of the SAS variable listed in the COLUMN list specification.

The NAMECTRL= option is used when a name of an arc or a nonarc variable in the CONDATA= data set (either a VAR list variable name or a value of the COLUMN list variable) is in the form *tail_head* and there exists an arc with these end nodes. If *tail_head* has not already been tagged as belonging to an arc or nonarc variable in the ARCDATA= data set, PROC INTPOINT needs to know whether *tail_head* is the name of the arc or the name of a nonarc variable.

If you specify NAMECTRL=1, a name that is not defined in the ARCDATA= data set is assumed to be the name of a nonarc variable. NAMECTRL=2 treats *tail_head* as the name of the arc with these endnodes, provided no other name is used to associate data in the CONDATA= data set with this arc. If the arc does have other names that appear in the CONDATA= data set, *tail_head* is assumed to be the name of a nonarc variable. If you specify NAMECTRL=3, *tail_head* is assumed to be a name of the arc with these end nodes, whether the arc has other names or not. The default value of NAMECTRL is 3.

If the dense format is used for the CONDATA= data set, there are two circumstances that affect how this data set is read:

1. if you are running SAS Version 6, or a previous version to that, or if you are running SAS Version 7 onward and you specify

```
options validvarname=v6;
```

in your SAS session. Let's refer to this as *case 1*.

2. if you are running SAS Version 7 onward and you do not specify

```
options validvarname=v6;
```

in your SAS session. Let's refer to this as *case 2*.

For *case 1*, the SAS System converts SAS variable names in a SAS program to uppercase. The VAR list variable names are uppercased. Because of this, PROC INTPOINT automatically uppercases names of arcs and nonarc variables or LP variables (the values of the NAME list variable) in the ARCDATA= data set. The names of arcs and nonarc variables or LP variables (the values of the NAME list variable) appear uppercased in the CONOUT= data set.

Also, if the dense format is used for the CONDATA= data set, be careful with default arc names (names in the form tailnode_headnode). Node names (values in the TAILNODE and HEADNODE list variables) in the ARCDATA= data set are not automatically uppercased by PROC INTPOINT. Consider the following statements:

```
data arcdata;
   input _from_ $ _to_ $ _name $ ;
   datalines;
from to1 .
from to2 arc2
TAIL TO3 .
;
data densecon;
   input from_to1 from_to2 arc2 tail_to3;
   datalines;
2 3 3 5
;
proc intpoint
   arcdata=arcdata condata=densecon;
   run;
```

The SAS System does not uppercase character string values within SAS data sets. PROC INTPOINT never uppercases node names, so the arcs in observations 1, 2, and 3 in the preceding ARCDATA= data set have the default names from_to1, from_to2, and TAIL_TO3, respectively. When the dense format of the CONDATA= data set is used, PROC INTPOINT does uppercase values of the NAME list variable, so the name of the arc in the second observation of the ARCDATA= data set is ARC2. Thus, the second arc has two names: its default from_to2 and the other that was specified ARC2.

As the SAS System uppercases program code, you must think of the input statement

```
input from_to1 from_to2 arc2 tail_to3;
```

as really being

```
INPUT FROM_TO1 FROM_TO2 ARC2 TAIL_TO3;
```

The SAS variables named FROM_TO1 and FROM_TO2 are *not* associated with any of the arcs in the preceding ARCDATA= data set. The values FROM_TO1 and FROM_TO2 are different from all of the arc names from_to1, from_to2, TAIL_TO3, and ARC2. FROM_TO1 and FROM_TO2 could end up being the names of two nonarc variables.

The SAS variable named ARC2 is the name of the second arc in the ARCDATA= data set, even though the name specified in the ARCDATA= data set looks like arc2. The SAS variable named TAIL_TO3 is the default name of the third arc in the ARCDATA= data set.

For *case 2*, the SAS System does not convert SAS variable names in a SAS program to uppercase. The VAR list variable names are not uppercased. PROC INTPOINT does not automatically uppercase names of arcs and nonarc variables or LP variables (the values of the NAME list variable) in the ARCDATA= data set. PROC INTPOINT does not uppercase any SAS variable names, data set values, or indeed anything. Therefore, PROC INTPOINT respects case, and characters in the data if compared must have the right case if you mean them to be the same. Note how the input statement in the DATA step that initialized the data set densecon below is specified in the following code:

```
data arcdata;
   input _from_ $ _to_ $ _name $ ;
   datalines;
from to1 .
from to2 arc2
TAIL TO3 .
;
data densecon;
   input from_to1 from_to2 arc2 TAIL_TO3;
   datalines;
2 3 3 5
;
proc intpoint
   arcdata=arcdata condata=densecon;
   run;
```

NARCS=*n*

specifies the approximate number of arcs. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

NCOEFS=*n*

specifies the approximate number of constraint coefficients. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

NCONS=*n*

specifies the approximate number of constraints. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

NNAS=*n*

specifies the approximate number of nonarc variables. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

NNODES=*n*
specifies the approximate number of nodes. See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

NON_REPLIC=*non_replic*
prevents PROC INTPOINT from doing unnecessary checks of data previously read.

- NON_REPLIC=COEFS indicates that each constraint coefficient is specified *once* in the CONDATA= data set.
- NON_REPLIC=NONE indicates that constraint coefficients can be specified more than once in the CONDATA= data set. NON_REPLIC=NONE is the default.

See the section "How to Make the Data Read of PROC INTPOINT More Efficient" on page 119.

OPTIM_TIMER
indicates that the procedure is to issue a message to the SAS log giving the CPU time spent doing optimization. This includes the time spent preprocessing, performing optimization, and postprocessing. Not counted in that time is the rest of the procedure execution, which includes reading the data and creating output SAS data sets.

The time spent optimizing can be small compared to the total CPU time used by the procedure. This is especially true when the problem is quite small (e.g., fewer than 10,000 variables).

RHSOBS=*charstr*
specifies the keyword that identifies a right-hand-side observation when using the sparse format for data in the CONDATA= data set. The keyword is expected as a value of the SAS variable in the CONDATA= data set named in the COLUMN list specification. The default value of the RHSOBS= option is _RHS_ or _rhs_. If *charstr* is not a valid SAS variable name, enclose it in quotes.

SCALE=*scale*
indicates that the NPSC side constraints or the LP constraints are to be scaled. Scaling is useful when some coefficients are either much larger or much smaller than other coefficients. Scaling might make all coefficients have values that have a smaller range, and this can make computations more stable numerically. Try the SCALE= option if PROC INTPOINT is unable to solve a problem because of numerical instability. Specify

- SCALE=ROW, SCALE=CON, or SCALE=CONSTRAINT if you want the largest absolute value of coefficients in each constraint to be about 1.0
- SCALE=COL, SCALE=COLUMN, or SCALE=NONARC if you want NPSC nonarc variable columns or LP variable columns to be scaled so that the absolute value of the largest constraint coefficient of that variable is near to 1
- SCALE=BOTH if you want the largest absolute value of coefficients in each constraint, and the absolute value of the largest constraint coefficient of an NPSC nonarc variable or LP variable to be near to 1. This is the default.
- SCALE=NONE if no scaling should be done

SHORTPATH

SP

specifies that PROC INTPOINT solve a shortest path problem. The INTPOINT procedure finds the shortest path between the nodes specified in the SOURCE= option and the SINK= option. The costs of arcs are their *lengths*. PROC INTPOINT automatically assigns a supply of one flow unit to the SOURCE= node, and the SINK= node is assigned to have a one flow unit demand. In this way, the SHORTPATH option sets up a shortest path problem as an equivalent minimum cost problem.

If a network has one supply node (with supply of one unit) and one demand node (with demand of one unit), you could specify the SHORTPATH option, with the SOURCE= and SINK= nodes, even if the problem is not a shortest path problem. You then should not provide any supply or demand data in the NODEDATA= data set or the ARCDATA= data set.

SINK=*sinkname*

SINKNODE=*sinkname*

identifies the demand node. The SINK= option is useful when you specify the MAXFLOW option or the SHORTPATH option and you need to specify toward which node the shortest path or maximum flow is directed. The SINK= option also can be used when a minimum cost problem has only one demand node. Rather than having this information in the ARCDATA= data set or the NODEDATA= data set, use the SINK= option with an accompanying DEMAND= specification for this node. The SINK= option must be the name of a head node of at least one arc; thus, it must have a character value. If the value of the SINK= option is not a valid SAS character variable name (if, for example, it contains embedded blanks), it must be enclosed in quotes.

SOURCE=*sourcename*

SOURCENODE=*sourcename*

identifies a supply node. The SOURCE= option is useful when you specify the MAXFLOW or the SHORTPATH option and need to specify from which node the shortest path or maximum flow originates. The SOURCE= option also can be used when a minimum cost problem has only one supply node. Rather than having this information in the ARCDATA= data set or the NODEDATA= data set, use the SOURCE= option with an accompanying SUPPLY= amount of supply at this node. The SOURCE= option must be the name of a tail node of at least one arc; thus, it must have a character value. If the value of the SOURCE= option is not a valid SAS character variable name (if, for example, it contains embedded blanks), it must be enclosed in quotes.

SPARSECONDATA

SCDATA

indicates that the CONDATA= data set has data in the sparse data format. Otherwise, it is assumed that the data are in the dense format.

NOTE: If the SPARSECONDATA option is not specified, and you are running SAS software Version 6 or you have specified

```
options validvarname=v6;
```

all NAME list variable values in the ARCDATA= data set are uppercased. See the section "Case Sensitivity" on page 111.

SUPPLY=*s*

specifies the supply at the source node specified by the SOURCE= option. The SUPPLY= option should be used only if the SOURCE= option is given in the PROC INTPOINT statement and neither the SHORTPATH option nor the MAXFLOW option is specified. If you are solving a minimum cost network problem and the SOURCE= option is used to identify the source node and the SUPPLY= option is not specified, then by default the supply at the source node is made equal to the network's total demand.

THRUNET

tells PROC INTPOINT to force through the network any excess supply (the amount by which total supply exceeds total demand) or any excess demand (the amount by which total demand exceeds total supply) as is required. If a network problem has unequal total supply and total demand and the THRUNET option is not specified, PROC INTPOINT drains away the excess supply or excess demand in an optimal manner. The consequences of specifying or not specifying THRUNET are discussed in the section "Balancing Total Supply and Total Demand" on page 118.

TYPEOBS=*charstr*

specifies the keyword that identifies a type observation when using the sparse format for data in the CONDATA= data set. The keyword is expected as a value of the SAS variable in the CONDATA= data set named in the COLUMN list specification. The default value of the TYPEOBS= option is _TYPE_ or _type_. If *charstr* is not a valid SAS variable name, enclose it in quotes.

VERBOSE=*v*

limits the number of similar messages that are displayed on the SAS log.

For example, when reading the ARCDATA= data set, PROC INTPOINT might have cause to issue the following message many times:

```
ERROR: The HEAD list variable value in obs i in ARCDATA is
       missing and the TAIL list variable value of this obs
       is nonmissing. This is an incomplete arc specification.
```

If there are many observations that have this fault, messages that are similar are issued for only the first VERBOSE= such observations. After the ARCDATA= data set has been read, PROC INTPOINT will issue the message

```
NOTE: More messages similar to the ones immediately above
      could have been issued but were suppressed as
      VERBOSE=v.
```

If observations in the ARCDATA= data set have this error, PROC INTPOINT stops and you have to fix the data. Imagine that this error is only a warning and PROC INTPOINT proceeded to other operations such as reading the CONDATA= data set. If PROC INTPOINT finds there are numerous errors when reading that data set, the number of messages issued to the SAS log are also limited by the VERBOSE= option.

When PROC INTPOINT finishes and messages have been suppressed, the message

```
NOTE: To see all messages, specify VERBOSE=vmin.
```

is issued. The value of *vmin* is the smallest value that should be specified for the VERBOSE= option so that *all* messages are displayed if PROC INTPOINT is run again with the same data and everything else (except VERBOSE=*vmin*) unchanged.

The default value for the VERBOSE= option is 12.

ZERO2=*z*

Z2=*z*

specifies the zero tolerance level used when determining whether the final solution has been reached. ZERO2= is also used when outputting the solution to the CONOUT= data set. Values within *z* of zero are set to 0.0, where *z* is the value of the ZERO2= option. Flows close to the lower flow bound or capacity of arcs are reassigned those exact values. If there are nonarc variables, values close to the lower or upper value bound of nonarc variables are reassigned those exact values. When solving an LP problem, values close to the lower or upper value bound of LP variables are reassigned those exact values.

The ZERO2= option works when determining whether optimality has been reached or whether an element in the vector $(\Delta x^k, \Delta y^k, \Delta s^k)$ is less than or greater than zero. It is crucial to know that when determining the maximal value for the step length α in the formula

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

See the description of the PDSTEPMULT= option for more details on this computation.

Two values are deemed to be close if one is within *z* of the other. The default value for the ZERO2= option is 0.000001. Any value specified for the ZERO2= option that is < 0.0 or > 0.0001 is not valid.

ZEROTOL=*z*

specifies the zero tolerance used when PROC INTPOINT must compare any real number with another real number, or zero. For example, if x and y are real numbers, then for x to be considered greater than y, x must be at least $y + z$. The ZEROTOL= option is used throughout any PROC INTPOINT run.

ZEROTOL=*z* controls the way PROC INTPOINT performs all double precision comparisons; that is, whether a double precision number is equal to, not equal to, greater than (or equal to), or less than (or equal to) zero or some other double precision number. A double precision number is deemed to be the same as another such value if the absolute difference between them is less than or equal to the value of the ZEROTOL= option.

The default value for the ZEROTOL= option is 1.0E−14. You can specify the ZEROTOL= option in the INTPOINT statement. Valid values for the ZEROTOL= option must be $>$ 0.0 and $<$ 0.0001. Do not specify a value too close to zero as this defeats the purpose of the ZEROTOL= option. Neither should the value be too large, as comparisons might be incorrectly performed.

Interior Point Algorithm Options

FACT_METHOD=*f*

enables you to choose the type of algorithm used to factorize and solve the main linear systems at each iteration of the interior point algorithm.

FACT_METHOD=LEFT_LOOKING is new for SAS 9.1.2. It uses algorithms described in George, Liu, and Ng (2001). Left looking is one of the main methods used to perform Cholesky optimization and, along with some recently developed implementation approaches, can be faster and require less memory than other algorithms.

Specify FACT_METHOD=USE_OLD if you want the procedure to use the only factorization available prior to SAS 9.1.2.

TOLDINF=*t*

RTOLDINF=*t*

specifies the allowed amount of dual infeasibility. In the section "Interior Point Algorithmic Details" on page 41, the vector $infeas_d$ is defined. If all elements of this vector are $\leq t$, the solution is considered dual feasible. $infeas_d$ is replaced by a zero vector, making computations faster. This option is the dual equivalent to the TOLPINF= option. Increasing the value of the TOLDINF= option too much can lead to instability, but a modest increase can give the algorithm added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

TOLPINF=*t*

RTOLPINF=*t*

specifies the allowed amount of primal infeasibility. This option is the primal equivalent to the TOLDINF= option. In the section "Interior Point: Upper Bounds" on page 45, the vector $infeas_b$ is defined. In the section "Interior Point Algorithmic Details" on page 41, the vector $infeas_c$ is defined. If all elements in these vectors are $\leq t$, the solution is considered primal feasible. $infeas_b$ and $infeas_c$ are replaced by zero vectors, making computations faster. Increasing the value of the TOLPINF= option too much can lead to instability, but a modest increase can give the algorithm added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

TOLTOTDINF=*t*

RTOLTOTDINF=*t*

specifies the allowed total amount of dual infeasibility. In the section "Interior Point Algorithmic Details" on page 41, the vector $infeas_d$ is defined. If $\sum_{i=1}^{n} infeas_{di} \leq t$, the solution is considered dual feasible. $infeas_d$ is replaced by a zero vector, making computations faster. This option is the dual equivalent to the TOLTOTPINF= option. Increasing the value of the TOLTOTDINF= option too much can lead to instability, but a modest increase can give the algorithm added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

TOLTOTPINF=*t*

RTOLTOTPINF=*t*

specifies the allowed total amount of primal infeasibility. This option is the primal equivalent to the TOLTOTDINF= option. In the section "Interior Point: Upper Bounds" on page 45, the vector $infeas_b$ is defined. In the section "Interior Point Algorithmic Details" on page 41, the vector $infeas_c$ is defined. If $\sum_{i=1}^{n} infeas_{bi} \leq t$ and $\sum_{i=1}^{m} infeas_{ci} \leq t$, the solution is considered primal feasible. $infeas_b$ and $infeas_c$ are replaced by zero vectors, making computations faster. Increasing the value of the TOLTOTPINF= option too much can lead to instability, but a modest increase can give the algorithm

added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

CHOLTINYTOL=*c*

RCHOLTINYTOL=*c*

specifies the cut-off tolerance for Cholesky factorization of the $A\Theta A^{-1}$. If a diagonal value drops below *c*, the row is essentially treated as dependent and is ignored in the factorization. Valid values for *c* are between 1.0E−30 and 1.0E−6. The default value is 1.0E−8.

DENSETHR=*d*

RDENSETHR=*d*

specifies the density threshold for Cholesky factorization. When the symbolic factorization encounters a column of L (where L is the remaining unfactorized submatrix) that has DENSETHR= proportion of nonzeros and the remaining part of L is at least 12×12, the remainder of L is treated as dense. In practice, the lower right part of the Cholesky triangular factor L is quite dense and it can be computationally more efficient to treat it as 100% dense. The default value for *d* is 0.7. A specification of $d \leq 0.0$ causes all dense processing; $d \geq 1.0$ causes all sparse processing.

PDSTEPMULT=*p*

RPDSTEPMULT=*p*

specifies the step-length multiplier. The maximum feasible step-length chosen by the interior point algorithm is multiplied by the value of the PDSTEPMULT= option. This number must be less than 1 to avoid moving beyond the barrier. An actual step-length greater than 1 indicates numerical difficulties. Valid values for *p* are between 0.01 and 0.999999. The default value is 0.99995.

In the section "Interior Point Algorithmic Details" on page 41, the solution of the next iteration is obtained by moving along a direction from the current iteration's solution:

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where α is the maximum feasible step-length chosen by the interior point algorithm. If $\alpha \leq 1$, then α is reduced slightly by multiplying it by *p*. α is a value as large as possible but ≤ 1.0 and not so large that an x_i^{k+1} or s_i^{k+1} of some variable i is "too close" to zero.

PRSLTYPE=*p*

IPRSLTYPE=*p*

Preprocessing the linear programming problem often succeeds in allowing some variables and constraints to be temporarily eliminated from the resulting LP that must be solved. This reduces the solution time and possibly also the chance that the optimizer will run into numerical difficulties. The task of preprocessing is inexpensive to do.

You control how much preprocessing to do by specifying PRSLTYPE=*p*, where *p* can be –1, 0, 1, 2, or 3:

–1 Do not perform preprocessing. For most problems, specifying PRSLTYPE= –1 is *not* recommended.

0 Given upper and lower bounds on each variable, the greatest and least contribution to the row activity of each variable is computed. If these are within the limits set by the upper and lower bounds on the row activity, then the row is redundant and can be discarded. Otherwise, whenever possible, the bounds on any of the variables are tightened. For example, if all coefficients in a constraint are positive and all variables have zero lower bounds, then the row's smallest contribution is zero. If the rhs value of this constraint is zero, then if the constraint type is = or $\leq$, all the variables in that constraint are fixed to zero. These variables and the constraint are removed. If the constraint type is $\geq$, the constraint is redundant. If the rhs is negative and the constraint is $\leq$, the problem is infeasible. If just one variable in a row is not fixed, the row to used to impose an implicit upper or lower bound on the variable and then this row is eliminated. The preprocessor also tries to tighten the bounds on constraint right-hand sides.

1 When there are exactly two unfixed variables with coefficients in an equality constraint, one variable is solved in terms of the other. The problem will have one less variable. The new matrix will have at least two fewer coefficients and one less constraint. In other constraints where both variables appear, two coefficients are combined into one. PRSLTYPE=0 reductions are also done.

2 It may be possible to determine that an equality constraint is not constraining a variable. That is, if all variables are nonnegative, then $x - \sum_i y_i = 0$ does not constrain x, since it must be nonnegative if all the y_i's are nonnegative. In this case, x is eliminated by subtracting this equation from all others containing x. This is useful when the only other entry for x is in the objective function. This reduction is performed if there is at most one other nonobjective coefficient. PRSLTYPE=0 reductions are also done.

3 All possible reductions are performed. PRSLTYPE=3 is the default.

Preprocessing is iterative. As variables are fixed and eliminated, and constraints are found to be redundant and they too are eliminated, and as variable bounds and constraint right-hand sides are tightened, the LP to be optimized is modified to reflect these changes. Another iteration of preprocessing of the modified LP may reveal more variables and constraints that are eliminated, or tightened.

PRINTLEVEL2=*p*

is used when you want to see PROC INTPOINT's progress to the optimum. PROC INTPOINT will produce a table on the SAS log. A row of the table is generated during each iteration and may consist of values of

- the affine step complementarity
- the complementarity of the solution for the next iteration
- the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45)
- the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41)
- the total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41)

As optimization progresses, the values in all columns should converge to zero. If you specify PRINTLEVEL2=2, all columns will appear in the table. If PRINTLEVEL2=1 is specified, only the affine step complementarity and the complementarity of the solution for the next iteration will appear. Some time is saved by not calculating the infeasibility values.

PRINTLEVEL2=2 is specified in all PROC INTPOINT runs in the section "Examples: INTPOINT Procedure" on page 126.

RTTOL=*r*

specifies the zero tolerance used during the ratio test of the interior point algorithm. The ratio test determines α, the maximum feasible step length.

Valid values for *r* are greater than 1.0E−14. The default value is 1.0E−10.

In the section "Interior Point Algorithmic Details" on page 41, the solution of the next iteration is obtained by moving along a direction from the current iteration's solution:

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where α is the maximum feasible step-length chosen by the interior point algorithm. If $\alpha \leq 1$, then α is reduced slightly by multiplying it by the value of the PDSTEPMULT= option. α is a value as large as possible but ≤ 1.0 and not so large that an x_i^{k+1} or s_i^{k+1} of some variable i is negative. When determining α, only negative elements of Δx and Δs are important.

RTTOL=*r* indicates a number close to zero so that another number n is considered truly negative if $n \leq -r$. Even though $n < 0$, if $n > -r$, n may be too close to zero and may have the wrong sign due to rounding error.

Interior Point Algorithm Options: Stopping Criteria

MAXITERB=*m*

IMAXITERB=*m*

specifies the maximum number of iterations that the interior point algorithm can perform. The default value for *m* is 100. One of the most remarkable aspects of the interior point algorithm is that for most problems, it usually needs to do a small number of iterations, *no matter the size of the problem*.

PDGAPTOL=*p*

RPDGAPTOL=*p*

specifies the primal-dual gap or duality gap tolerance. Duality gap is defined in the section "Interior Point Algorithmic Details" on page 41. If the relative gap ($duality\ gap/(c^T x)$) between the primal and dual objectives is smaller than the value of the PDGAPTOL= option and both the primal and dual problems are feasible, then PROC INTPOINT stops optimization with a solution that is deemed optimal. Valid values for *p* are between 1.0E−12 and 1.0E−1. The default is 1.0E−7.

STOP_C=*s*

is used to determine whether optimization should stop. At the beginning of each iteration, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

STOP_DG=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

STOP_IB=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

STOP_IC=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is <= *s*, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

STOP_ID=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is <= *s*, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

AND_STOP_C=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, *and* the other conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

AND_STOP_DG=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, *and* the other conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

AND_STOP_IB=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is <= *s*, *and* the other conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

AND_STOP_IC=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is <= *s*, *and* the other conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

AND_STOP_ID=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is <= *s*, *and* the other conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 123.

KEEPGOING_C=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

KEEPGOING_DG=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

KEEPGOING_IB=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

KEEPGOING_IC=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is > *s*, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

KEEPGOING_ID=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is > *s*, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

AND_KEEPGOING_C=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, *and* the other conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

AND_KEEPGOING_DG=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, *and* the other conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

AND_KEEPGOING_IB=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the other conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

AND_KEEPGOING_IC=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the other conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

AND_KEEPGOING_ID=*s*
is used to determine whether optimization should stop. When a stopping condition is met, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the other conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 123.

CAPACITY Statement

CAPACITY *variable* **;**

CAPAC *variable* **;**

UPPERBD *variable* **;**

The CAPACITY statement identifies the SAS variable in the ARCDATA= data set that contains the maximum feasible flow or capacity of the network arcs. If an observation contains nonarc variable information, the

CAPACITY list variable is the upper value bound for the nonarc variable named in the NAME list variable in that observation.

When solving an LP, the CAPACITY statement identifies the SAS variable in the ARCDATA= data set that contains the maximum feasible value of the LP variables.

The CAPACITY list variable must have numeric values. It is not necessary to have a CAPACITY statement if the name of the SAS variable is _CAPAC_, _UPPER_, _UPPERBD, or _HI_.

COEF Statement

COEF *variables* **;**

The COEF list is used with the sparse input format of the CONDATA= data set. The COEF list can contain more than one SAS variable, each of which must have numeric values. If the COEF statement is not specified, the CONDATA= data set is searched and SAS variables with names beginning with _COE are used. The number of SAS variables in the COEF list must be no greater than the number of SAS variables in the ROW list.

The values of the COEF list variables in an observation can be interpreted differently than these variables' values in other observations. The values can be coefficients in the side constraints, costs and objective function coefficients, bound data, constraint type data, or rhs data. If the COLUMN list variable has a value that is a name of an arc or a nonarc variable, the *i*th COEF list variable is associated with the constraint or special row name named in the *i*th ROW list variable. Otherwise, the COEF list variables indicate type values, rhs values, or missing values.

When solving an LP, the values of the COEF list variables in an observation can be interpreted differently than these variables' values in other observations. The values can be coefficients in the constraints, objective function coefficients, bound data, constraint type data, or rhs data. If the COLUMN list variable has a value that is a name of an LP variable, the *i*th COEF list variable is associated with the constraint or special row name named in the *i*th ROW list variable. Otherwise, the COEF list variables indicate type values, rhs values, or missing values.

COLUMN Statement

COLUMN *variable* **;**

The COLUMN list is used with the sparse input format of the CONDATA= data set.

This list consists of one SAS variable in the CONDATA= data set that has as values the names of arc variables, nonarc variables, or missing values. When solving an LP, this list consists of one SAS variable in the CONDATA= data set that has as values the names of LP variables, or missing values. Some, if not all, of these values also can be values of the NAME list variables of the ARCDATA= data set. The COLUMN list variable can have other special values (Refer to the TYPEOBS= and RHSOBS= options). If the COLUMN list is not specified after the PROC INTPOINT statement, the CONDATA= data set is searched and a SAS variable named _COLUMN_ is used. The COLUMN list variable must have character values.

COST Statement

COST *variable* **;**

OBJFN *variable* **;**

The COST statement identifies the SAS variable in the ARCDATA= data set that contains the per unit flow cost through an arc. If an observation contains nonarc variable information, the value of the COST list variable is the objective function coefficient of the nonarc variable named in the NAME list variable in that observation.

If solving an LP, the COST statement identifies the SAS variable in the ARCDATA= data set that contains the per unit objective function coefficient of an LP variable named in the NAME list variable in that observation.

The COST list variable must have numeric values. It is not necessary to specify a COST statement if the name of the SAS variable is _COST_ or _LENGTH_.

DEMAND Statement

DEMAND *variable* **;**

The DEMAND statement identifies the SAS variable in the ARCDATA= data set that contains the demand at the node named in the corresponding HEADNODE list variable. The DEMAND list variable must have numeric values. It is not necessary to have a DEMAND statement if the name of this SAS variable is _DEMAND_. See the section "Missing S Supply and Missing D Demand Values" on page 114 for cases when the SUPDEM list variable values can have other values. There should be no DEMAND statement if you are solving an LP.

HEADNODE Statement

HEADNODE *variable* **;**

HEAD *variable* **;**

TONODE *variable* **;**

TO *variable* **;**

The HEADNODE statement specifies the SAS variable that must be present in the ARCDATA= data set that contains the names of nodes toward which arcs are directed. It is not necessary to have a HEADNODE statement if the name of the SAS variable is _HEAD_ or _TO_. The HEADNODE variable must have character values.

There should be no HEAD statement if you are solving an LP.

ID Statement

ID *variables* **;**

The ID statement specifies SAS variables containing values for pre- and post-optimal processing and analysis. These variables are not processed by PROC INTPOINT but are read by the procedure and written in the CONOUT= data set. For example, imagine a network used to model a distribution system. The SAS variables listed on the ID statement can contain information on the type of vehicle, the transportation mode, the condition of the road, the time to complete the journey, the name of the driver, or other ancillary information useful for report writing or describing facets of the operation that do not have bearing on the optimization. The ID variables can be character, numeric, or both.

If no ID list is specified, the procedure forms an ID list of all SAS variables not included in any other implicit or explicit list specification. If the ID list is specified, any SAS variables in the ARCDATA= data set not in any list are dropped and do not appear in the CONOUT= data set.

LO Statement

LO *variable* **;**

LOWERBD *variable* **;**

MINFLOW *variable* **;**

The LO statement identifies the SAS variable in the ARCDATA= data set that contains the minimum feasible flow or lower flow bound for arcs in the network. If an observation contains nonarc variable information, the LO list variable has the value of the lower bound for the nonarc variable named in the NAME list variable. If solving an LP, the LO statement identifies the SAS variable in the ARCDATA= data set that contains the lower value bound for LP variables. The LO list variables must have numeric values. It is not necessary to have a LO statement if the name of this SAS variable is _LOWER_, _LO_, _LOWERBD, or _MINFLOW.

NAME Statement

NAME *variable* **;**

ARCNAME *variable* **;**

VARNAME *variable* **;**

Each arc and nonarc variable in an NPSC, or each variable in an LP, that has data in the CONDATA= data set must have a unique name. This variable is identified in the ARCDATA= data set. The NAME list variable must have character values (see the NAMECTRL= option in the PROC INTPOINT statement for more information). It is not necessary to have a NAME statement if the name of this SAS variable is _NAME_.

NODE Statement

NODE *variable* **;**

The NODE list variable, which must be present in the NODEDATA= data set, has names of nodes as values. These values must also be values of the TAILNODE list variable, the HEADNODE list variable, or both. If this list is not explicitly specified, the NODEDATA= data set is searched for a SAS variable with the name _NODE_. The NODE list variable must have character values.

QUIT Statement

QUIT ;

The QUIT statement indicates that PROC INTPOINT is to stop immediately. The solution is not saved in the CONOUT= data set. The QUIT statement has no options.

RHS Statement

RHS *variable* **;**

The RHS variable list is used when the dense format of the CONDATA= data set is used. The values of the SAS variable specified in the RHS list are constraint right-hand-side values. If the RHS list is not specified, the CONDATA= data set is searched and a SAS variable with the name _RHS_ is used. The RHS list variable must have numeric values. If there is no RHS list and no SAS variable named _RHS_, all constraints are assumed to have zero right-hand-side values.

ROW Statement

ROW *variables* **;**

The ROW list is used when either the sparse or the dense format of the CONDATA= data set is being used. SAS variables in the ROW list have values that are constraint or special row names. The SAS variables in the ROW list must have character values.

If the dense data format is used, there must be only one SAS variable in this list. In this case, if a ROW list is not specified, the CONDATA= data set is searched and the SAS variable with the name _ROW_ or _CON_ is used. If that search fails to find a suitable SAS variable, data for each constraint must reside in only one observation.

If the sparse data format is used and the ROW statement is not specified, the CONDATA= data set is searched and SAS variables with names beginning with _ROW or _CON are used. The number of SAS variables in the ROW list must not be less than the number of SAS variables in the COEF list. The *i*th ROW list variable is paired with the *i*th COEF list variable. If the number of ROW list variables is greater than the number of COEF list variables, the last ROW list variables have no COEF partner. These ROW list variables that have no corresponding COEF list variable are used in observations that have a TYPE list variable value. All ROW list variable values are tagged as having the type indicated. If there is no TYPE list variable, all ROW list variable values are constraint names.

RUN Statement

RUN ;

The RUN statement causes optimization to be started. The RUN statement has no options. If PROC INTPOINT is called and is not terminated because of an error or a QUIT statement, and you have not used a RUN statement, a RUN statement is assumed implicitly as the last statement of PROC INTPOINT. Therefore, PROC INTPOINT reads that data, performs optimization, and saves the optimal solution in the CONOUT= data set.

SUPDEM Statement

SUPDEM *variable* **;**

The SAS variable in this list, which must be present in the NODEDATA= data set, contains supply and demand information for the nodes in the NODE list. A positive SUPDEM list variable value s ($s > 0$) denotes that the node named in the NODE list variable can supply s units of flow. A negative SUPDEM list variable value $-d$ ($d > 0$) means that this node demands d units of flow. If a SAS variable is not explicitly specified, a SAS variable with the name _SUPDEM_ or _SD_ in the NODEDATA= data set is used as the SUPDEM variable. If a node is a transshipment node (neither a supply nor a demand node), an observation associated with this node need not be present in the NODEDATA= data set. If present, the SUPDEM list variable value must be zero or a missing value. See the section "Missing S Supply and Missing D Demand Values" on page 114 for cases when the SUPDEM list variable values can have other values.

SUPPLY Statement

SUPPLY *variable* **;**

The SUPPLY statement identifies the SAS variable in the ARCDATA= data set that contains the supply at the node named in that observation's TAILNODE list variable. If a tail node does not supply flow, use zero or a missing value for the observation's SUPPLY list variable value. If a tail node has supply capability, a missing value indicates that the supply quantity is given in another observation. It is not necessary to have a SUPPLY statement if the name of this SAS variable is _SUPPLY_. See the section "Missing S Supply and Missing D Demand Values" on page 114 for cases when the SUPDEM list variable values can have other values. There should be no SUPPLY statement if you are solving an LP.

TAILNODE Statement

TAILNODE *variable* **;**

TAIL *variable* **;**

FROMNODE *variable* **;**

FROM *variable* **;**

The TAILNODE statement specifies the SAS variable that must (when solving an NPSC problem) be present in the ARCDATA= data set that has as values the names of tail nodes of arcs. The TAILNODE variable must have character values. It is not necessary to have a TAILNODE statement if the name of the SAS variable is _TAIL_ or _FROM_. If the TAILNODE list variable value is missing, it is assumed that the observation of the ARCDATA= data set contains information concerning a nonarc variable. There should be no TAILNODE statement if you are solving an LP.

TYPE Statement

TYPE *variable* **;**

CONTYPE *variable* **;**

The TYPE list, which is optional, names the SAS variable that has as values keywords that indicate either the constraint type for each constraint or the type of special rows in the CONDATA= data set. The values of the TYPE list variable also indicate, in each observation of the CONDATA= data set, how values of the VAR or COEF list variables are to be interpreted and how the type of each constraint or special row name is determined. If the TYPE list is not specified, the CONDATA= data set is searched and a SAS variable with the name _TYPE_ is used. Valid keywords for the TYPE variable are given below. If there is no TYPE statement and no other method is used to furnish type information (see the DEFCONTYPE= option), all constraints are assumed to be of the type "less than or equal to" and no special rows are used. The TYPE list variable must have character values and can be used when the data in the CONDATA= data set is in either the sparse or the dense format. If the TYPE list variable value has a * as its first character, the observation is ignored because it is a comment observation.

TYPE List Variable Values

The following are valid TYPE list variable values. The letters in boldface denote the characters that PROC INTPOINT uses to determine what type the value suggests. You need to have at least these characters. In the following list, the minimal TYPE list variable values have additional characters to aid you in remembering these values.

<	less than or equal to ($\leq$)
=	equal to (=)
>	greater than or equal to ($\geq$)
CAPAC	capacity
COST	cost
EQ	equal to
FREE	free row (used only for linear programs solved by interior point)
GE	greater than or equal to
LE	less than or equal to
LOWERBD	lower flow or value bound
LOW*blank*	lower flow or value bound
MAXIMIZE	maximize (opposite of cost)
MINIMIZE	minimize (same as cost)
OBJECTIVE	objective function (same as cost)
RHS	rhs of constraint
TYPE	type of constraint

UPPCOST	reserved for future use
UNREST	unrestricted variable (used only for linear programs solved by interior point)
UPPER	upper value bound or capacity; second letter must not be N

The valid TYPE list variable values in function order are

- **LE** less than or equal to ($\leq$)
- **EQ** equal to ($=$)
- **GE** greater than or equal to ($\geq$)
- **CO**ST
  MINIMIZE
  MAXIMIZE
  OBJECTIVE
  cost or objective function coefficient
- **CA**PAC
  UPPER
  capacity or upper value bound
- **LOW**ERBD
  LOW*blank*
  lower flow or value bound
- **R**HS rhs of constraint
- **T**YPE type of constraint

A TYPE list variable value that has the first character $*$ causes the observation to be treated as a comment. If the first character is a negative sign, then $\leq$ is the type. If the first character is a zero, then $=$ is the type. If the first character is a positive number, then $\geq$ is the type.

VAR Statement

VAR *variables* **;**

The VAR variable list is used when the dense data format is used for the CONDATA= data set. The names of these SAS variables are also names of the arc and nonarc variables that have data in the CONDATA= data set. If solving an LP, the names of these SAS variables are also names of the LP variables. If no explicit VAR list is specified, all numeric SAS variables in the CONDATA= data set that are not in other SAS variable lists are put onto the VAR list. The VAR list variables must have numeric values. The values of the VAR list variables in some observations can be interpreted differently than in other observations. The values can be coefficients in the side constraints, costs and objective function coefficients, or bound data. When solving an LP, the values of the SAS variables in the VAR list can be constraint coefficients, objective function coefficients, or

bound data. How these numeric values are interpreted depends on the value of each observation's TYPE or ROW list variable value. If there are no TYPE list variables, the VAR list variable values are all assumed to be side constraint coefficients.

Details: INTPOINT Procedure

Input Data Sets

PROC INTPOINT is designed so that there are as few rules as possible that you must obey when inputting a problem's data. Raw data are acceptable. This should cut the amount of processing required to groom the data before it is input to PROC INTPOINT. Data formats are so flexible that, due to space restrictions, all possible forms for a problem's data are not shown here. Try any reasonable form for your problem's data; it should be acceptable. PROC INTPOINT will outline its objections.

You can supply the same piece of data several ways. You do not have to restrict yourself to using any particular one. If you use several ways, PROC INTPOINT checks that the data are consistent each time that the data are encountered. After all input data sets have been read, data are merged so that the problem is described completely. The observations can be in any order.

ARCDATA= Data Set

See the section "NPSC Problems" on page 55 and the section "Introductory NPSC Example" on page 57 for a description of this input data set.

NOTE: Information for an arc or nonarc variable can be specified in more than one observation. For example, consider an arc directed from node A toward node B that has a cost of 50, capacity of 100, and lower flow bound of 10 flow units. Some possible observations in the ARCDATA= data set are as follows:

tail	_head_	_cost_	_capac_	_lo_
A	B	50	.	.
A	B	.	100	.
A	B	.	.	10
A	B	50	100	.
A	B	.	100	10
A	B	50	.	10
A	B	50	100	10

Similarly, for a nonarc variable that has an upper bound of 100, a lower bound of 10, and an objective function coefficient of 50, the _TAIL_ and _HEAD_ values are missing.

When solving an LP that has an LP variable named my_var with an upper bound of 100, a lower bound of 10, and an objective function coefficient of 50, some possible observations in the ARCDATA= data set are

name	_cost_	_capac_	_lo_
my_var	50	.	.
my_var	.	100	.
my_var	.	.	10

```
my_var    50       100      .
my_var     .       100     10
my_var    50        .      10
my_var    50       100     10
```

CONDATA= Data Set

Regardless of whether the data in the CONDATA= data set is in the sparse or dense format, you will receive a warning if PROC INTPOINT finds a constraint row that has no coefficients. You will also be warned if any nonarc or LP variable has no constraint coefficients.

Dense Input Format

If the dense format is used, most SAS variables in the CONDATA= data set belong to the VAR list. The names of the SAS variables belonging to this list have names of arc and nonarc variables or, if solving an LP, names of the LP variables. These names can be values of the SAS variables in the ARCDATA= data set that belong to the NAME list, or names of nonarc variables, or names in the form *tail_head*, or any combination of these three forms. Names in the form *tail_head* are default arc names, and if you use them, you must specify node names in the ARCDATA= data set (values of the TAILNODE and HEADNODE list variables).

The CONDATA= data set can have three other SAS variables belonging, respectively, to the ROW, the TYPE, and the RHS lists. The CONDATA= data set of the oil industry example in the section "Introductory NPSC Example" on page 57 uses the dense data format.

Consider the SAS code that creates a dense format CONDATA= data set that has data for three constraints. This data set was used in the section "Introductory NPSC Example" on page 57.

```
data cond1;
   input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
      _type_ $ _rhs_;
   datalines;
-2  .  1 .  . . >= -15
 . -2  . .  1 . GE -15
 .  . -3 4  . . EQ   0
 .  .  . . -3 4  =   0
;
```

You can use nonconstraint type values to furnish data on costs, capacities, lower flow bounds (and, if there are nonarc or LP variables, objective function coefficients and upper and lower bounds). You need not have such (or as much) data in the ARCDATA= data set. The first three observations in the following data set are examples of observations that provide cost, capacity, and lower bound data.

```
data cond1b;
   input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
         _type_ $ _rhs_;
   datalines;
63 81 200   . 220   . cost   .
95 80 175 140 100 100 capac  .
20 10  50   .  35   . lo     .
-2  .   1   .   .   . >=   -15
 . -2   .   .   1   . GE   -15
 .  .  -3   4   .   . EQ     0
 .  .   .   .  -3   4  =     0
;
```

If a ROW list variable is used, the data for a constraint can be spread over more than one observation. To illustrate, the data for the first constraint (which is called con1) and the cost and capacity data (in special rows called costrow and caprow, respectively) are spread over more than one observation in the following data set.

```
data cond1c;
   input _row_ $
         m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
         _type_ $ _rhs_;
   datalines;
costrow 63  .   .   .   .   . .      .
costrow  . 81 200   .   .   . cost   .
.        .  .   .   . 220   . cost   .
caprow   .  .   .   .   .   . capac  .
caprow  95  . 175   . 100 100 .      .
caprow   . 80 175 140   .   . .      .
lorow   20 10  50   .  35   . lo     .
con1    -2  .   1   .   .   . .      .
con1     .  .   .   .   .   . >=   -15
con2     . -2   .   .   1   . GE   -15
con3     .  .  -3   4   .   . EQ     0
con4     .  .   .   .  -3   4  =     0
;
```

Using both ROW and TYPE lists, you can use special row names. Examples of these are costrow and caprow in the last data set. It should be restated that in any of the input data sets of PROC INTPOINT, the order of the observations does not matter. However, the CONDATA= data set can be read more quickly if PROC INTPOINT knows what type of constraint or special row a ROW list variable value is. For example, when the first observation is read, PROC INTPOINT does not know whether costrow is a constraint or special row and how to interpret the value 63 for the arc with the name m_e_ref1. When PROC INTPOINT reads the second observation, it learns that costrow has cost type and that the values 81 and 200 are costs. When the entire CONDATA= data set has been read, PROC INTPOINT knows the type of all special rows and constraints. Data that PROC INTPOINT had to set aside (such as the first observation 63 value and the costrow ROW list variable value, which at the time had unknown type, but is subsequently known to be a cost special row) is reprocessed. During this second pass, if a ROW list variable value has unassigned constraint or special row type, it is treated as a constraint with DEFCONTYPE= (or DEFCONTYPE= default) type. Associated VAR list variable values are coefficients of that constraint.

Sparse Input Format

The side constraints usually become sparse as the problem size increases. When the sparse data format of the CONDATA= data set is used, only nonzero constraint coefficients must be specified. Remember to specify the SPARSECONDATA option in the PROC INTPOINT statement. With the sparse method of specifying constraint information, the names of arc and nonarc variables or, if solving an LP, the names of LP variables do not have to be valid SAS variable names.

A sparse format CONDATA= data set for the oil industry example in the section "Introductory NPSC Example" on page 57 is displayed below.

```
title 'Setting Up Condata = Cond2 for PROC INTPOINT';
data cond2;
   input _column_ $ _row1 $ _coef1 _row2 $ _coef2 ;
   datalines;
```

```
m_e_ref1  con1  -2     .    .
m_e_ref2  con2  -2     .    .
thruput1  con1   1  con3  -3
r1_gas       .   .  con3   4
thruput2  con2   1  con4  -3
r2_gas       .   .  con4   4
_type_    con1   1  con2   1
_type_    con3   0  con4   0
_rhs_     con1 -15  con2 -15
;
```

Recall that the COLUMN list variable values _type_ and _rhs_ are the default values of the TYPEOBS= and RHSOBS= options. Also, the default rhs value of constraints (con3 and con4) is zero. The third to last observation has the value _type_ for the COLUMN list variable. The _ROW1 variable value is con1, and the _COEF1_ variable has the value 1. This indicates that the constraint con1 is *greater than* or equal to type (because the value 1 is *greater than* zero). Similarly, the data in the second to last observation's _ROW2 and _COEF2 variables indicate that con2 is an *equality* constraint (0 *equals* zero).

An alternative, using a TYPE list variable, is

```
title 'Setting Up Condata = Cond3 for PROC INTPOINT';
data cond3;
   input _column_ $ _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
m_e_ref1  con1  -2     .    . >=
m_e_ref2  con2  -2     .    . .
thruput1  con1   1  con3  -3 .
r1_gas       .   .  con3   4 .
thruput2  con2   1  con4  -3 .
r2_gas       .   .  con4   4 .
.         con3   .  con4   . eq
.         con1 -15  con2 -15 ge
;
```

If the COLUMN list variable is missing in a particular observation (the last 2 observations in the data set cond3, for instance), the constraints named in the ROW list variables all have the constraint type indicated by the value in the TYPE list variable. It is for this type of observation that you are allowed more ROW list variables than COEF list variables. If corresponding COEF list variables are not missing (for example, the last observation in the data set cond3), these values are the rhs values of those constraints. Therefore, you can specify both constraint type and rhs in the same observation.

As in the previous CONDATA= data set, if the COLUMN list variable is an arc or nonarc variable, the COEF list variable values are coefficient values for that arc or nonarc variable in the constraints indicated in the corresponding ROW list variables. If in this same observation the TYPE list variable contains a constraint type, all constraints named in the ROW list variables in that observation have this constraint type (for example, the first observation in the data set cond3). Therefore, you can specify both constraint type and coefficient information in the same observation.

Also note that DEFCONTYPE=EQ could have been specified, saving you from having to include in the data that con3 and con4 are of this type.

In the oil industry example, arc costs, capacities, and lower flow bounds are presented in the ARCDATA= data set. Alternatively, you could have used the following input data sets. The arcd2 data set has only two SAS variables. For each arc, there is an observation in which the arc's tail and head node are specified.

```
title3 'Setting Up Arcdata = Arcd2 for PROC INTPOINT';
data arcd2;
   input  _from_&$11. _to_&$15. ;
   datalines;
middle east  refinery 1
middle east  refinery 2
u.s.a.       refinery 1
u.s.a.       refinery 2
refinery 1   r1
refinery 2   r2
r1           ref1 gas
r1           ref1 diesel
r2           ref2 gas
r2           ref2 diesel
ref1 gas     servstn1 gas
ref1 gas     servstn2 gas
ref1 diesel  servstn1 diesel
ref1 diesel  servstn2 diesel
ref2 gas     servstn1 gas
ref2 gas     servstn2 gas
ref2 diesel  servstn1 diesel
ref2 diesel  servstn2 diesel
;

title 'Setting Up Condata = Cond4 for PROC INTPOINT';
data cond4;
   input _column_&$27. _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
.                              con1 -15   con2 -15    ge
.                            costrow   .       .   .  cost
.                                  .   . caprow   . capac
middle east_refinery 1          con1  -2       .   .     .
middle east_refinery 2          con2  -2       .   .     .
refinery 1_r1                   con1   1   con3  -3     .
r1_ref1 gas                        .   .   con3   4     =
refinery 2_r2                   con2   1   con4  -3     .
r2_ref2 gas                        .   .   con4   4    eq
middle east_refinery 1       costrow  63 caprow  95     .
middle east_refinery 2       costrow  81 caprow  80     .
u.s.a._refinery 1            costrow  55       .   .     .
u.s.a._refinery 2            costrow  49       .   .     .
refinery 1_r1                costrow 200 caprow 175     .
refinery 2_r2                costrow 220 caprow 100     .
r1_ref1 gas                        .   . caprow 140     .
r1_ref1 diesel                     .   . caprow  75     .
r2_ref2 gas                        .   . caprow 100     .
r2_ref2 diesel                     .   . caprow  75     .
ref1 gas_servstn1 gas        costrow  15 caprow  70     .
ref1 gas_servstn2 gas        costrow  22 caprow  60     .
ref1 diesel_servstn1 diesel  costrow  18       .   .     .
```

```
ref1 diesel_servstn2 diesel  costrow  17      .   .     .
ref2 gas_servstn1 gas        costrow  17 caprow  35     .
ref2 gas_servstn2 gas        costrow  31      .   .     .
ref2 diesel_servstn1 diesel  costrow  36      .   .     .
ref2 diesel_servstn2 diesel  costrow  23      .   .     .
middle east_refinery 1             .  20      .   .    lo
middle east_refinery 2             .  10      .   .    lo
refinery 1_r1                      .  50      .   .    lo
refinery 2_r2                      .  35      .   .    lo
ref2 gas_servstn1 gas              .   5      .   .    lo
;
```

The first observation in the cond4 data set defines con1 and con2 as *greater than or equal to* (≥) constraints that both (by coincidence) have rhs values of -15. The second observation defines the special row costrow as a cost row. When costrow is a ROW list variable value, the associated COEF list variable value is interpreted as a cost or objective function coefficient. PROC INTPOINT has to do less work if constraint names and special rows are defined in observations near the top of a data set, but this is not a strict requirement. The fourth to ninth observations contain constraint coefficient data. Observations seven and nine have TYPE list variable values that indicate that constraints con3 and con4 are equality constraints. The last five observations contain lower flow bound data. Observations that have an arc or nonarc variable name in the COLUMN list variable, a nonconstraint type TYPE list variable value, and a value in (one of) the COEF list variables are valid.

The following data set is equivalent to the cond4 data set.

```
title 'Setting Up Condata = Cond5 for PROC INTPOINT';
data cond5;
   input _column_&$27. _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
middle east_refinery 1          con1  -2 costrow  63     .
middle east_refinery 2          con2  -2   lorow  10     .
refinery 1_r1                      .   .    con3  -3     =
r1_ref1 gas                   caprow 140    con3   4     .
refinery 2_r2                   con2   1    con4  -3     .
r2_ref2 gas                        .   .    con4   4    eq
.                               CON1 -15    CON2 -15    GE
ref2 diesel_servstn1 diesel        .  36 costrow   .  cost
.                                  .   .  caprow   . capac
.                             lorow   .       .   .    lo
middle east_refinery 1        caprow  95   lorow  20     .
middle east_refinery 2        caprow  80 costrow  81     .
u.s.a._refinery 1                  .   .       .  55  cost
u.s.a._refinery 2            costrow  49       .   .     .
refinery 1_r1                   con1   1  caprow 175     .
refinery 1_r1                  lorow  50 costrow 200     .
refinery 2_r2                costrow 220  caprow 100     .
refinery 2_r2                      .  35       .   .    lo
r1_ref1 diesel               caprow2  75       .   . capac
r2_ref2 gas                        .   .  caprow 100     .
r2_ref2 diesel               caprow2  75       .   .     .
ref1 gas_servstn1 gas        costrow  15  caprow  70     .
ref1 gas_servstn2 gas        caprow2  60 costrow  22     .
ref1 diesel_servstn1 diesel        .   . costrow  18     .
ref1 diesel_servstn2 diesel  costrow  17       .   .     .
```

```
ref2 gas_servstn1 gas         costrow  17   lorow   5     .
ref2 gas_servstn1 gas               .   .  caprow2  35     .
ref2 gas_servstn2 gas               .  31        .   .  cost
ref2 diesel_servstn2 diesel         .   .  costrow  23     .
;
```

Converting from an NPSC to an LP Problem

If you have data for a linear programming program that has an embedded network, the steps required to change that data into a form that is acceptable by PROC INTPOINT are

1. Identify the nodal flow conservation constraints. The coefficient matrix of these constraints (a submatrix of the LP's constraint coefficient matrix) has only two nonzero elements in each column, -1 and 1.
2. Assign a node to each nodal flow conservation constraint.
3. The rhs values of conservation constraints are the corresponding node's supplies and demands. Use this information to create the NODEDATA= data set.
4. Assign an arc to each column of the flow conservation constraint coefficient matrix. The arc is directed from the node associated with the row that has the *1* element in it and directed toward to the node associated with the row that has the -1 element in it. Set up the ARCDATA= data set that has two SAS variables. This data set could resemble ARCDATA=arcd2. These will eventually be the TAILNODE and HEADNODE list variables when PROC INTPOINT is used. Each observation consists of the tail and head node of each arc.
5. Remove from the data of the linear program all data concerning the nodal flow conservation constraints.
6. Put the remaining data into a CONDATA= data set. This data set will probably resemble CONDATA=cond4 or CONDATA=cond5.

The Sparse Format Summary

The following list illustrates possible CONDATA= data set observation sparse formats. a1, b1, b2, b3 and c1 have as a _COLUMN_ variable value either the name of an arc (possibly in the form *tail_head*) or the name of a nonarc variable (if you are solving an NPSC), or the name of the LP variable (if you are solving an LP). These are collectively referred to as variable in the tables that follow.

- If there is no TYPE list variable in the CONDATA= data set, the problem must be constrained and there is no nonconstraint data in the CONDATA= data set:

	COLUMN	_ROWx_	_COEFx_	_ROWy_ (no _COEFy_) (may not be in CONDATA)
a1	variable	constraint	lhs coef	
a2	_TYPE_ or TYPEOBS=	constraint	-1 0 1	
a3	_RHS_ or RHSOBS= or missing	constraint	rhs value	constraint or missing
a4	_TYPE_ or TYPEOBS=	constraint	missing	
a5	_RHS_ or RHSOBS= or missing	constraint	missing	

  Observations of the form a4 and a5 serve no useful purpose but are still allowed to make problem generation easier.

- If there are no ROW list variables in the data set, the problem has no constraints and the information is nonconstraint data. There must be a TYPE list variable and only one COEF list variable in this case. The COLUMN list variable has as values the names of arcs or nonarc variables and must not have missing values or special row names as values:

	COLUMN	_TYPE_	_COEFx_
b1	variable	UPPERBD	capacity
b2	variable	LOWERBD	lower flow
b3	variable	COST	cost

- Using a TYPE list variable for constraint data implies the following:

	COLUMN	_TYPE_	_ROWx_	_COEFx_	_ROWy_ (no _COEFy_) (may not be in CONDATA)
c1	variable	missing	c o	lhs coef -1 0 1	
c2	_TYPE_ or TYPEOBS=	missing			
c3	_RHS_ or missing or RHSOBS=	missing	n s t	rhs value	constraint or missing
c4	variable	con type	r	lhs coef	
c5	_RHS_ or missing or RHSOBS=	con type	a i n	rhs value	
c6	missing	TYPE	t	-1 0 1	
c7	missing	RHS		rhs value	

If the observation is in form c4 or c5, and the _COEFx_ values are missing, the constraint is assigned the type data specified in the _TYPE_ variable.

- Using a TYPE list variable for arc and nonarc variable data implies the following:

	COLUMN	_TYPE_	_ROWx_	_COEFx_	_ROWy_ (no _COEFy_) (may not be in CONDATA)
d1	variable	UPPERBD	missing or special row name	capacity	missing or special row name
d2	variable	LOWERBD		lowerflow	
d3	variable	COST		cost	
d4	missing		special row name		
d5	variable	missing		value that is interpreted according to _ROWx_	missing

The observations of the form d1 to d5 can have ROW list variable values. Observation d4 must have ROW list variable values. The ROW value is put into the ROW name tree so that when dealing with observation d4 or d5, the COEF list variable value is interpreted according to the type of ROW list variable value. For example, the following three observations define the _ROWx_ variable values up_row, lo_row, and co_row as being an upper value bound row, lower value bound row, and cost row, respectively:

```
_COLUMN_          _TYPE_          _ROWx_          _COEFx_

   .              UPPERBD         up_row             .
variable_a        LOWERBD         lo_row         lower flow
variable_b        COST            co_row         cost
```

PROC INTPOINT is now able to correctly interpret the following observation:

```
_COLUMN_ _TYPE_ _ROW1_ _COEF1_ _ROW2_ _COEF2_ _ROW3_ _COEF3_

 var_c      .    up_row  upval  lo_row  loval  co_row   cost
```

If the TYPE list variable value is a constraint type and the value of the COLUMN list variable equals the value of the TYPEOBS= option or the default value _TYPE_, the TYPE list variable value is ignored.

NODEDATA= Data Set

See the section "NPSC Problems" on page 55 and the section "Introductory NPSC Example" on page 57 for a description of this input data set.

Output Data Sets

For NPSC problems, the procedure determines the flow that should pass through each arc as well as the value that should be assigned to each nonarc variable. The goal is that the minimum flow bounds, capacities, lower and upper value bounds, and side constraints are not violated. This goal is reached when total cost incurred by such a flow pattern and value assignment is feasible and optimal. The solution found must also conserve flow at each node.

For LP problems, the procedure determines the value that should be assigned to each variable. The goal is that the lower and upper value bounds and the constraints are not violated. This goal is reached when the total cost incurred by such a value assignment is feasible and optimal.

The CONOUT= data set can be produced and contains a solution obtained after performing optimization.

CONOUT= Data Set

The variables in the CONOUT= data set depend on whether or not the problem has a network component. If the problem has a network component, the variables and their possible values in an observation are as follows:

FROM	a tail node of an arc. This is a missing value if an observation has information about a nonarc variable.
_TO_	a head node of an arc. This is a missing value if an observation has information about a nonarc variable.
COST	the cost of an arc or the objective function coefficient of a nonarc variable
CAPAC	the capacity of an arc or upper value bound of a nonarc variable

_LO_	the lower flow bound of an arc or lower value bound of a nonarc variable
NAME	a name of an arc or nonarc variable
SUPPLY	the supply of the tail node of the arc in the observation. This is a missing value if an observation has information about a nonarc variable.
DEMAND	the demand of the head node of the arc in the observation. This is a missing value if an observation has information about a nonarc variable.
FLOW	the flow through the arc or value of the nonarc variable
FCOST	flow cost, the product of _COST_ and _FLOW_
RCOST	the reduced cost of the arc or nonarc variable
ANUMB	the number of the arc (positive) or nonarc variable (nonpositive); used for warm starting PROC NETFLOW
TNUMB	the number of the tail node in the network basis spanning tree; used for warm starting PROC NETFLOW
STATUS	the status of the arc or nonarc variable

If the problem does not have a network component, the variables and their possible values in an observation are as follows:

OBJFN	the objective function coefficient of a variable
_UPPERBD	the upper value bound of a variable
_LOWERBD	the lower value bound of a variable
NAME	the name of a variable
VALUE	the value of the variable
FCOST	objective function value for that variable; the product of _OBJFN_ and _VALUE_

The variables present in the ARCDATA= data set are present in a CONOUT= data set. For example, if there is a variable called tail in the ARCDATA= data set and you specified the SAS variable list

```
from tail;
```

then tail is a variable in the CONOUT= data sets instead of _FROM_. Any ID list variables also appear in the CONOUT= data sets.

MPSOUT= Data Set

The MPSOUT= data set contains problem data converted from a PROC INTPOINT format into an MPS-format SAS data set. The six fields, FIELD1 to FIELD6, in the MPSOUT= data set correspond to the six columns in MPS standard. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

Converting Any PROC INTPOINT Format to an MPS-Format SAS Data Set

The MPSOUT= option enables you to convert an input data set for the INTPOINT procedure into an MPS-format SAS data set. The converted data set is readable by the OPTLP procedure.

The conversion can handle linear programs and network flow formulations. If you specify a network flow formulation, it will be converted into an equivalent linear program. When multiple objective row names are present, rows with the name encountered first are combined into the objective row. The remaining rows are marked as free rows.

For information about how the contents of the MPS-format SAS data set are interpreted, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*). For examples that demonstrate the use of the MPSOUT= option and migration to the OPTMODEL procedure, see the section "Examples: INTPOINT Procedure" on page 126.

Case Sensitivity

Whenever the INTPOINT procedure has to compare character strings, whether they are node names, arc names, nonarc names, LP variable names, or constraint names, if the two strings have different lengths, or on a character by character basis the character is different *or has different cases*, PROC INTPOINT judges the character strings to be different.

Not only is this rule enforced when one or both character strings are obtained as values of SAS variables in PROC INTPOINT's input data sets, it also should be obeyed if one or both character strings were originally SAS variable names, or were obtained as the values of options or statements parsed to PROC INTPOINT. For example, if the network has only one node that has supply capability, or if you are solving a MAXFLOW or SHORTPATH problem, you can indicate that node using the SOURCE= option. If you specify

```
proc intpoint source=NotableNode
```

then PROC INTPOINT looks for a value of the TAILNODE list variable that is NotableNode.

Version 6 of the SAS System converts text that makes up statements into uppercase. The name of the node searched for would be NOTABLENODE, even if this was your SAS code:

```
proc intpoint source=NotableNode
```

If you want PROC INTPOINT to behave as it did in Version 6, specify

```
options validvarname=v6;
```

If the SPARSECONDATA option is not specified, and you are running SAS software Version 6, or you are running SAS software Version 7 onward and have specified

```
options validvarname=v6;
```

all values of the SAS variables that belong to the NAME list are uppercased. This is because the SAS System has uppercased all SAS variable names, particularly those in the VAR list of the CONDATA= data set.

Entities that contain blanks must be enclosed in quotes.

Loop Arcs

Loop arcs (which are arcs directed toward nodes from which they originate) are prohibited. Rather, introduce a dummy intermediate node in loop arcs. For example, replace arc (A,A) with (A,B) and (B,A); B is the name of a new node, and it must be distinct for each loop arc.

Multiple Arcs

Multiple arcs with the same tail and head nodes are prohibited. PROC INTPOINT checks to ensure there are no such arcs before proceeding with the optimization. Introduce a new dummy intermediate node in multiple arcs. This node must be distinct for each multiple arc. For example, if some network has three arcs directed from node A toward node B, then replace one of these three with arcs (A,C) and (C,B) and replace another one with (A,D) and (D,B). C and D are new nodes added to the network.

Flow and Value Bounds

The capacity and lower flow bound of an arc can be equal. Negative arc capacities and lower flow bounds are permitted. If both arc capacities and lower flow bounds are negative, the lower flow bound must be at least as negative as the capacity. An arc (A,B) that has a negative flow of $-f$ units can be interpreted as an arc that conveys f units of flow from node B to node A.

The upper and lower value bound of a nonarc variable can be equal. Negative upper and lower bounds are permitted. If both are negative, the lower bound must be at least as negative as the upper bound.

When solving an LP, the upper and lower value bounds of an LP variable can be equal. Negative upper and lower bounds are permitted. If both are negative, the lower bound must be at least as negative as the upper bound.

In short, for any problem to be feasible, a lower bound must be $\leq$ the associated upper bound.

Tightening Bounds and Side Constraints

If any piece of data is furnished to PROC INTPOINT more than once, PROC INTPOINT checks for consistency so that no conflict exists concerning the data values. For example, if the cost of some arc is seen to be one value and as more data are read, the cost of the same arc is seen to be another value, PROC INTPOINT issues an error message on the SAS log and stops. There are two exceptions to this:

- The bounds of arcs and nonarc variables, or the bounds of LP variables, are made as tight as possible. If several different values are given for the lower flow bound of an arc, the greatest value is used. If several different values are given for the lower bound of a nonarc or LP variable, the greatest value is used. If several different values are given for the capacity of an arc, the smallest value is used. If several different values are given for the upper bound of a nonarc or LP variable, the smallest value is used.

- Several values can be given for inequality constraint right-hand sides. For a particular constraint, the lowest rhs value is used for the rhs if the constraint is of *less than or equal to* type. For a particular constraint, the greatest rhs value is used for the rhs if the constraint is of *greater than or equal to* type.

Reasons for Infeasibility

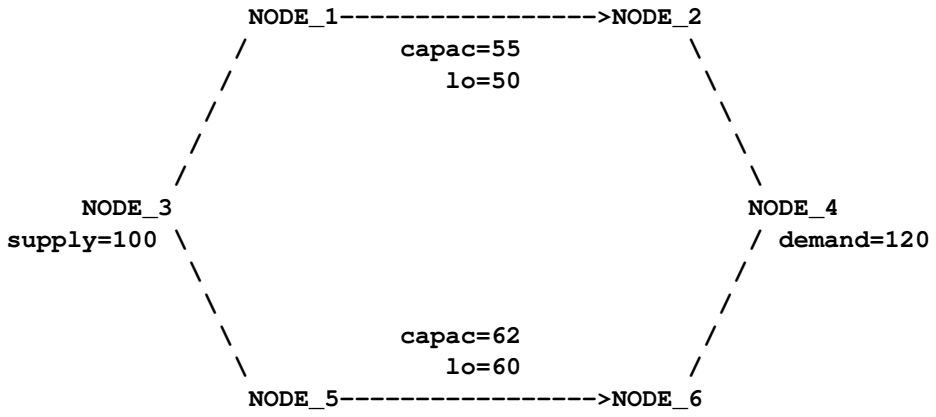
Before optimization commences, PROC INTPOINT tests to ensure that the problem is not infeasible by ensuring that, with respect to supplies, demands, and arc flow bounds, flow conservation can be obeyed at each node:

- Let *IN* be the sum of lower flow bounds of arcs directed toward a node plus the node's supply. Let *OUT* be the sum of capacities of arcs directed from that node plus the node's demand. If *IN* exceeds *OUT*, not enough flow can leave the node.
- Let *OUT* be the sum of lower flow bounds of arcs directed from a node plus the node's demand. Let *IN* be the total capacity of arcs directed toward the node plus the node's supply. If *OUT* exceeds *IN*, not enough flow can arrive at the node.

Reasons why a network problem can be infeasible are similar to those previously mentioned but apply to a set of nodes rather than for an individual node.

Consider the network illustrated in Figure 4.10.

Figure 4.10 An Infeasible Network



The demand of NODE_4 is 120. That can never be satisfied because the maximal flow through arcs (NODE_1, NODE_2) and (NODE_5, NODE_6) is 117. More specifically, the implicit supply of NODE_2 and NODE_6 is only 117, which is insufficient to satisfy the demand of other nodes (real or implicit) in the network.

Furthermore, the lower flow bounds of arcs (NODE_1, NODE_2) and (NODE_5, NODE_6) are greater than the flow that can reach the tail nodes of these arcs, that, by coincidence, is the total supply of the network. The implicit demand of nodes NODE_1 and NODE_5 is 110, which is greater than the amount of flow that can reach these nodes.

Missing S Supply and Missing D Demand Values

In some models, you may want a node to be either a supply or demand node but you want the node to supply or demand the optimal number of flow units. To indicate that a node is such a supply node, use a missing S value in the SUPPLY list variable in the ARCDATA= data set or the SUPDEM list variable in the NODEDATA= data set. To indicate that a node is such a demand node, use a missing D value in the DEMAND list variable in the ARCDATA= data set or the SUPDEM list variable in the NODEDATA= data set.

Suppose the oil example in the section "Introductory NPSC Example" on page 57 is changed so that crude oil can be obtained from either the Middle East or U.S.A. in any amounts. You should specify that the node middle east is a supply node, but you do not want to stipulate that it supplies 100 units, as before. The node u.s.a. should also remain a supply node, but you do not want to stipulate that it supplies 80 units. You must specify that these nodes have missing S supply capabilities:

```
title  'Oil Industry Example';
title3 'Crude Oil can come from anywhere';
data miss_s;
   missing S;
   input   _node_&$15. _sd_;
   datalines;
middle east          S
u.s.a.               S
servstn1 gas       -95
servstn1 diesel    -30
servstn2 gas       -40
servstn2 diesel    -15
;
```

The following PROC INTPOINT run uses the same ARCDATA= and CONDATA= data sets used in the section "Introductory NPSC Example" on page 57:

```
proc intpoint
   bytes=100000
   nodedata=miss_s       /* the supply (missing S) and */
                         /* demand data                */
   arcdata=arcd1         /* the arc descriptions       */
   condata=cond1         /* the side constraints       */
   conout=solution;      /* the solution data set      */
   run;


proc print;
   var _from_ _to_ _cost_ _capac_ _lo_ _flow_ _fcost_;
   sum _fcost_;
   run;
```

The following messages appear on the SAS log:

```
NOTE: Number of nodes= 14 .
NOTE: All supply nodes have unspecified (.S) supply capability. Number of these
      nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 0 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of side constraint coefficients= 8 .
NOTE: The following messages relate to the equivalent Linear Programming
      problem solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 17 .
NOTE: Number of >= constraints= 2 .
NOTE: Number of constraint coefficients= 48 .
NOTE: Number of variables= 20 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 5.
NOTE: After preprocessing, number of >= constraints= 2.
NOTE: The preprocessor eliminated 12 constraints from the problem.
NOTE: The preprocessor eliminated 33 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 6.
NOTE: The preprocessor eliminated 14 variables from the problem.
NOTE: 6 columns, 0 rows and 6 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 19 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 7 factor nodes make up 2 supernodes
NOTE: There are 4 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
NOTE: Bound feasibility attained by iteration 1.
NOTE: Dual feasibility attained by iteration 1.
NOTE: Constraint feasibility attained by iteration 1.
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 6
      iterations.
NOTE: Optimum reached.
NOTE: Objective= 50075.
NOTE: The data set WORK.SOLUTION has 18 observations and 10 variables.
NOTE: There were 18 observations read from the data set WORK.ARCD1.
NOTE: There were 6 observations read from the data set WORK.MISS_S.
NOTE: There were 4 observations read from the data set WORK.COND1.
```

The CONOUT= data set is shown in Figure 4.11.

Figure 4.11 Missing S SUPDEM Values in NODEDATA

Obs	_from_	_to_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_
1	refinery 1	r1	200	175	50	145.000	29000.00
2	refinery 2	r2	220	100	35	35.000	7700.00
3	r1	ref1 diesel	0	75	0	36.250	0.00
4	r1	ref1 gas	0	140	0	108.750	0.00
5	r2	ref2 diesel	0	75	0	8.750	0.00
6	r2	ref2 gas	0	100	0	26.250	0.00
7	middle east	refinery 1	63	95	20	20.000	1260.00
8	u.s.a.	refinery 1	55	99999999	0	125.000	6875.00
9	middle east	refinery 2	81	80	10	10.000	810.00
10	u.s.a.	refinery 2	49	99999999	0	25.000	1225.00
11	ref1 diesel	servstn1 diesel	18	99999999	0	30.000	540.00
12	ref2 diesel	servstn1 diesel	36	99999999	0	0.000	0.00
13	ref1 gas	servstn1 gas	15	70	0	68.750	1031.25
14	ref2 gas	servstn1 gas	17	35	5	26.250	446.25
15	ref1 diesel	servstn2 diesel	17	99999999	0	6.250	106.25
16	ref2 diesel	servstn2 diesel	23	99999999	0	8.750	201.25
17	ref1 gas	servstn2 gas	22	60	0	40.000	880.00
18	ref2 gas	servstn2 gas	31	99999999	0	0.000	0.00
							50075.00

The optimal supplies of nodes middle east and u.s.a. are 30 and 150 units, respectively. For this example, the same optimal solution is obtained if these nodes had supplies less than these values (each supplies 1 unit, for example) and the THRUNET option was specified in the PROC INTPOINT statement. With the THRUNET option active, when total supply exceeds total demand, the specified nonmissing demand values are the lowest number of flow units that must be absorbed by the corresponding node. This is demonstrated in the following PROC INTPOINT run. The missing S is most useful when nodes are to supply optimal numbers of flow units and it turns out that for some nodes, the optimal supply is 0.

```
data miss_s_x;
   missing S;
   input   _node_&$15. _sd_;
   datalines;
middle east         1
u.s.a.              1
servstn1 gas      -95
servstn1 diesel   -30
servstn2 gas      -40
servstn2 diesel   -15
;

proc intpoint
   bytes=100000
   thrunet
   nodedata=miss_s_x     /* No supply (missing S)      */
   arcdata=arcd1         /* the arc descriptions       */
   condata=cond1         /* the side constraints       */
   conout=solution;      /* the solution data set      */
   run;
```

```
proc print;
   var _from_ _to_ _cost_ _capac_ _lo_ _flow_ _fcost_;
   sum _fcost_;
   run;
```

The following messages appear on the SAS log. Note that the Total supply= 2, not 0 as in the last run:

```
NOTE: Number of nodes= 14 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 2 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of side constraint coefficients= 8 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 17 .
NOTE: Number of >= constraints= 2 .
NOTE: Number of constraint coefficients= 48 .
NOTE: Number of variables= 20 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 5.
NOTE: After preprocessing, number of >= constraints= 2.
NOTE: The preprocessor eliminated 12 constraints from the problem.
NOTE: The preprocessor eliminated 33 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 6.
NOTE: The preprocessor eliminated 14 variables from the problem.
NOTE: 6 columns, 0 rows and 6 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 19 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 7 factor nodes make up 2 supernodes
NOTE: There are 4 nonzero sub-rows or sub-columns outside the supernodal triangular
      regions along the factors leading diagonal.
NOTE: Bound feasibility attained by iteration 1.
NOTE: Dual feasibility attained by iteration 1.
NOTE: Constraint feasibility attained by iteration 1.
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 6
      iterations.
NOTE: Optimum reached.
NOTE: Objective= 50075.
NOTE: The data set WORK.SOLUTION has 18 observations and 10 variables.
NOTE: There were 18 observations read from the data set WORK.ARCD1.
NOTE: There were 6 observations read from the data set WORK.MISS_S_X.
NOTE: There were 4 observations read from the data set WORK.COND1.
```

If total supply exceeds total demand, any missing S values are ignored. If total demand exceeds total supply, any missing D values are ignored.

Balancing Total Supply and Total Demand

When Total Supply Exceeds Total Demand

When total supply of a network problem exceeds total demand, PROC INTPOINT adds an extra node (called the *excess node*) to the problem and sets the demand at that node equal to the difference between total supply and total demand. There are three ways that this excess node can be joined to the network. All three ways entail PROC INTPOINT generating a set of arcs (henceforth referred to as the *generated* arcs) that are directed toward the excess node. The total amount of flow in generated arcs equals the demand of the excess node. The generated arcs originate from one of three sets of nodes.

When you specify the THRUNET option, the set of nodes that generated arcs originate from are all demand nodes, even those demand nodes with unspecified demand capability. You indicate that a node has unspecified demand capability by using a missing D value instead of an actual value for demand data (discussed in the section "Missing S Supply and Missing D Demand Values" on page 114). The value specified as the demand of a demand node is in effect a lower bound of the number of flow units that node can actually demand. For missing D demand nodes, this lower bound is zero.

If you do not specify the THRUNET option, the way in which the excess node is joined to the network depends on whether there are demand nodes with unspecified demand capability (nodes with missing D demand) or not.

If there are missing D demand nodes, these nodes are the set of nodes that generated arcs originate from. The value specified as the demand of a demand node, if not missing D, is the number of flow units that node can actually demand. For a missing D demand node, the actual demand of that node may be zero or greater.

If there are no missing D demand nodes, the set of nodes that generated arcs originate from are the set of supply nodes. The value specified as the supply of a supply node is in effect an upper bound of the number of flow units that node can actually supply. For missing S supply nodes (discussed in the section "Missing S Supply and Missing D Demand Values" on page 114), this upper bound is zero, so missing S nodes when total supply exceeds total demand are transshipment nodes, that is, nodes that neither supply nor demand flow.

When Total Supply Is Less Than Total Demand

When total supply of a network problem is less than total demand, PROC INTPOINT adds an extra node (called the *excess node*) to the problem and sets the supply at that node equal to the difference between total demand and total supply. There are three ways that this excess node can be joined to the network. All three ways entail PROC INTPOINT generating a set of arcs (henceforth referred to as the *generated* arcs) that originate from the excess node. The total amount of flow in generated arcs equals the supply of the excess node. The generated arcs are directed toward one of three sets of nodes.

When you specify the THRUNET option, the set of nodes that generated arcs are directed toward are all supply nodes, even those supply nodes with unspecified supply capability. You indicate that a node has unspecified supply capability by using a missing S value instead of an actual value for supply data (discussed in the section "Missing S Supply and Missing D Demand Values" on page 114). The value specified as the supply of a supply node is in effect a lower bound of the number of flow units that the node can actually supply. For missing S supply nodes, this lower bound is zero.

If you do not specify the THRUNET option, the way in which the excess node is joined to the network depends on whether there are supply nodes with unspecified supply capability (nodes with missing S supply) or not.

If there are missing S supply nodes, these nodes are the set of nodes that generated arcs are directed toward. The value specified as the supply of a supply node, if not missing S, is the number of flow units that the node can actually supply. For a missing S supply node, the actual supply of that node may be zero or greater.

If there are no missing S supply nodes, the set of nodes that generated arcs are directed toward are the set of demand nodes. The value specified as the demand of a demand node is in effect an upper bound of the number of flow units that node can actually demand. For missing D demand nodes (discussed in the section "Missing S Supply and Missing D Demand Values" on page 114), this upper bound is zero, so missing D nodes when total supply is less than total demand are transshipment nodes, that is, nodes that neither supply nor demand flow.

How to Make the Data Read of PROC INTPOINT More Efficient

This section contains information that is useful when you want to solve large constrained network problems. However, much of this information is also useful if you have a large linear programming problem. All of the options described in this section that are not directly applicable to networks (options such as ARCS_ONLY_ARCDATA, ARC_SINGLE_OBS, NNODES=, and NARCS=) can be specified to improve the speed at which LP data are read.

Large Constrained Network Problems

Many of the models presented to PROC INTPOINT are enormous. They can be considered large by linear programming standards; problems with thousands, even millions, of variables and constraints. When dealing with side constrained network programming problems, models can have not only a linear programming component of that magnitude, but also a larger, possibly *much* larger, network component.

The majority of network problem's decision variables are arcs. Like an LP decision variable, an arc has an objective function coefficient, upper and lower value bounds, and a name. Arcs can have coefficients in constraints. Therefore, an arc is quite similar to an LP variable and places the same memory demands on optimization software as an LP variable. But a typical network model has many more arcs and nonarc variables than the typical LP model has variables. And arcs have tail and head nodes. Storing and processing node names require huge amounts of memory. To make matters worse, node names occupy memory at times when a large amount of other data should also reside in memory.

While memory requirements are lower for a model with embedded network component compared with the equivalent LP *once optimization starts*, the same is usually not true *during the data read*. Even though nodal flow conservation constraints in the LP should not be specified in the constrained network formulation, the memory requirements to read the latter are greater because each arc (unlike an LP variable) originates at one node and is directed toward another.

Paging

PROC INTPOINT has facilities to read data when the available memory is insufficient to store all the data at once. PROC INTPOINT does this by allocating memory for different purposes; for example, to store an array or receive data read from an input SAS data set. After that memory has filled, the information is written to disk and PROC INTPOINT can resume filling that memory with new information. Often, information must be retrieved from disk so that data previously read can be examined or checked for consistency. Sometimes,

to prevent any data from being lost, or to retain any changes made to the information in memory, the contents of the memory must be sent to disk before other information can take its place. This process of swapping information to and from disk is called paging. Paging can be very time-consuming, so it is crucial to minimize the amount of paging performed.

There are several steps you can take to make PROC INTPOINT read the data of network and linear programming models more efficiently, particularly when memory is scarce and the amount of paging must be reduced. PROC INTPOINT will then be able to tackle large problems in what can be considered reasonable amounts of time.

The Order of Observations

PROC INTPOINT is quite flexible in the ways data can be supplied to it. Data can be given by any reasonable means. PROC INTPOINT has convenient defaults that can save you work when generating the data. There can be several ways to supply the same piece of data, and some pieces of data can be given more than once. PROC INTPOINT reads everything, then merges it all together. However, this flexibility and convenience come at a price; PROC INTPOINT may not assume the data has a characteristic that, if possessed by the data, could save time and memory during the data read. Several options can indicate that the data has some exploitable characteristic.

For example, an arc cost can be specified once or several times in the ARCDATA= data set or the CONDATA= data set, or both. Every time it is given in the ARCDATA= data set, a check is made to ensure that the new value is the same as any corresponding value read in a previous observation of the ARCDATA= data set. Every time it is given in the CONDATA= data set, a check is made to ensure that the new value is the same as the value read in a previous observation of the CONDATA= data set, or previously in the ARCDATA= data set. PROC INTPOINT would save time if it knew that arc cost data would be encountered only once while reading the ARCDATA= data set, so performing the time-consuming check for consistency would not be necessary. Also, if you indicate that the CONDATA= data set contains data for constraints only, PROC INTPOINT will not expect any arc information, so memory will not be allocated to receive such data while reading the CONDATA= data set. This memory is used for other purposes and this might lead to a reduction in paging. If applicable, use the ARC_SINGLE_OBS or the CON_SINGLE_OBS option, or both, and the NON_REPLIC=COEFS specification to improve how the ARCDATA= data set and the CONDATA= data set are read.

PROC INTPOINT allows the observations in input data sets to be in any order. However, major time savings can result if you are prepared to order observations in particular ways. Time spent by the SORT procedure to sort the input data sets, particularly the CONDATA= data set, may be more than made up for when PROC INTPOINT reads them, because PROC INTPOINT has in memory information possibly used when the previous observation was read. PROC INTPOINT can assume a piece of data is either similar to that of the last observation read or is new. In the first case, valuable information such as an arc or a nonarc variable number or a constraint number is retained from the previous observation. In the last case, checking the data with what has been read previously is not necessary.

Even if you do not sort the CONDATA= data set, grouping observations that contain data for the same arc or nonarc variable or the same row pays off. PROC INTPOINT establishes whether an observation being read is similar to the observation just read.

In practice, many input data sets for PROC INTPOINT have this characteristic, because it is natural for data for each constraint to be grouped together (when using the dense format of the CONDATA= data set) or data for each column to be grouped together (when using the sparse format of the CONDATA= data set). If data

for each arc or nonarc is spread over more than one observation of the ARCDATA= data set, it is natural to group these observations together.

Use the GROUPED= option to indicate whether observations of the ARCDATA= data set, the CONDATA= data set, or both, are grouped in a way that can be exploited during data read.

You can save time if the type data for each row appears near the top of the CONDATA= data set, especially if it has the sparse format. Otherwise, when reading an observation, if PROC INTPOINT does not know if a row is a constraint or special row, the data are set aside. Once the data set has been completely read, PROC INTPOINT must reprocess the data it set aside. By then, it knows the type of each constraint or row or, if its type was not provided, it is assumed to have a default type.

Better Memory Utilization

In order for PROC INTPOINT to make better utilization of available memory, you can specify options that indicate the approximate size of the model. PROC INTPOINT then knows what to expect. For example, if you indicate that the problem has no nonarc variables, PROC INTPOINT will not allocate memory to store nonarc data. That memory is better utilized for other purposes. Memory is often allocated to receive or store data of some type. If you indicate that the model does not have much data of a particular type, the memory that would otherwise have been allocated to receive or store that data can be used to receive or store data of another type.

The problem size options are as follows:

- NNODES= approximate number of nodes
- NARCS= approximate number of arcs
- NNAS= approximate number of nonarc variables or LP variables
- NCONS= approximate number of NPSC side constraints or LP constraints
- NCOEFS= approximate number of NPSC side constraint coefficients or LP constraint coefficients

These options will sometimes be referred to as Nxxxx= options.

You do not need to specify all these options for the model, but the more you do, the better. If you do not specify some or all of these options, PROC INTPOINT guesses the size of the problem by using what it already knows about the model. Sometimes PROC INTPOINT guesses the size of the model by looking at the number of observations in the ARCDATA= and the CONDATA= data sets. However, PROC INTPOINT uses rough rules of thumb, that typical models are proportioned in certain ways (for example, if there are constraints, then arcs, nonarc variables, or LP variables usually have about five constraint coefficients). If your model has an unusual shape or structure, you are encouraged to use these options.

If you do use the options and you do not know the exact values to specify, *overestimate* the values. For example, if you specify NARCS=10000 but the model has 10100 arcs, when dealing with the last 100 arcs, PROC INTPOINT might have to page out data for 10000 arcs each time one of the last arcs must be dealt with. Memory could have been allocated for all 10100 arcs without affecting (much) the rest of the data read, so NARCS=10000 could be more of a hindrance than a help.

The point of these Nxxxx= options is to indicate the model size when PROC INTPOINT does not know it. When PROC INTPOINT knows the “real” value, that value is used instead of Nxxxx= .

ARCS_ONLY_ARCDATA indicates that data for only arcs are in the ARCDATA= data set. Memory would not be wasted to receive data for nonarc variables.

Use the memory usage options:

- The BYTES= option specifies the size of PROC INTPOINT main working memory in number of bytes.
- The MEMREP option indicates that memory usage report is to be displayed on the SAS log.

Specifying an appropriate value for the BYTES= parameter is particularly important. Specify as large a number as possible, but not so large a number that will cause PROC INTPOINT (that is, the SAS System running underneath PROC INTPOINT) to run out of memory.

PROC INTPOINT reports its memory requirements on the SAS log if you specify the MEMREP option.

Use Defaults to Reduce the Amount of Data

Use the parameters that specify default values as much as possible. For example, if there are many arcs with the same cost value *c*, use DEFCOST=*c* for arcs that have that cost. Use missing values in the COST variable in the ARCDATA= data set instead of *c*. PROC INTPOINT ignores missing values, but must read, store, and process nonmissing values, even if they are equal to a default option or could have been equal to a default parameter had it been specified. Sometimes, using default parameters makes the need for some SAS variables in the ARCDATA= and the CONDATA= data sets no longer necessary, or reduces the quantity of data that must be read. The default options are

- DEFCOST= default cost of arcs, objective function of nonarc variables or LP variables
- DEFMINFLOW= default lower flow bound of arcs, lower bound of nonarc variables or LP variables
- DEFCAPACITY= default capacity of arcs, upper bound of nonarc variables or LP variables
- DEFCONTYPE= LE or DEFCONTYPE= <=
  DEFCONTYPE= EQ or DEFCONTYPE= =
  DEFCONTYPE= GE or DEFCONTYPE= >=

DEFCONTYPE=LE is the default.

The default options themselves have defaults. For example, you do not need to specify DEFCOST=0 in the PROC INTPOINT statement. You should still have missing values in the COST variable in the ARCDATA= data set for arcs that have zero costs.

If the network has only one supply node, one demand node, or both, use

- SOURCE= name of single node that has supply capability
- SUPPLY= the amount of supply at SOURCE
- SINK= name of single node that demands flow
- DEMAND= the amount of flow SINK demands

Do not specify that a constraint has zero right-hand-side values. That is the default. The only time it might be practical to specify a zero rhs is in observations of the CONDATA= data set read early so that PROC INTPOINT can infer that a row is a constraint. This could prevent coefficient data from being put aside because PROC INTPOINT did not know the row was a constraint.

Names of Things

To cut data read time and memory requirements, reduce the number of bytes in the longest node name, the longest arc name, the longest nonarc variable name, the longest LP variable name, and the longest constraint name to 8 bytes or less. The longer a name, the more bytes must be stored and compared with other names.

If an arc has no constraint coefficients, do not give it a name in the NAME list variable in the ARCDATA= data set. Names for such arcs serve no purpose.

PROC INTPOINT can have a default name for each arc. If an arc is directed from node *tailname* toward node *headname*, the default name for that arc is *tailname_headname*. If you do not want PROC INTPOINT to use these default arc names, specify NAMECTRL=1. Otherwise, PROC INTPOINT must use memory for storing node names and these node names must be searched often.

If you want to use the default *tailname_headname name*, that is, NAMECTRL=2 or NAMECTRL=3, do not use underscores in node names. If the CONDATA has a dense format and has a variable in the VAR list A_B_C_D, or if the value A_B_C_D is encountered as a value of the COLUMN list variable when reading the CONDATA= data set that has the sparse format, PROC INTPOINT first looks for a node named A. If it finds it, it looks for a node called B_C_D. It then looks for a node with the name A_B and possibly a node with name C_D. A search is then conducted for a node named A_B_C and possibly a node named D is done. Underscores could have caused PROC INTPOINT to look unnecessarily for nonexistent nodes. Searching for node names can be expensive, and the amount of memory to store node names is often large. It might be better to assign the arc name A_B_C_D directly to an arc by having that value as a NAME list variable value for that arc in the ARCDATA= data set and specify NAMECTRL=1.

Other Ways to Speed Up Data Reads

Arcs and nonarc variables, or LP variables, can have associated with them values or quantities that have no bearing on the optimization. This information is given in the ARCDATA= data set in the ID list variables. For example, in a distribution problem, information such as truck number and driver's name can be associated with each arc. This is useful when the optimal solution saved in the CONOUT= data set is analyzed. However, PROC INTPOINT needs to reserve memory to process this information when data are being read. For large problems when memory is scarce, it might be better to remove ancillary data from the ARCDATA. After PROC INTPOINT runs, use SAS software to merge this information into the CONOUT= data set that contains the optimal solution.

Stopping Criteria

There are several reasons why PROC INTPOINT stops interior point optimization. Optimization stops when

- the number of iteration equals MAXITERB=*m*

- the relative gap ($duality\ gap/(c^T x)$) between the primal and dual objectives is smaller than the value of the PDGAPTOL= option, and both the primal and dual problems are feasible. Duality gap is defined in the section "Interior Point Algorithmic Details" on page 41.

PROC INTPOINT may stop optimization when it detects that the rate at which the complementarity or duality gap is being reduced is too slow; that is, that there are consecutive iterations when the complementarity or duality gap has stopped getting smaller and the infeasibilities, if nonzero, have also stalled. Sometimes this indicates that the problem is infeasible.

The reasons to stop optimization outlined in the previous paragraph will be termed the *usual* stopping conditions in the following explanation.

However, when solving some problems, especially if the problems are large, the usual stopping criteria are inappropriate. PROC INTPOINT might stop optimizing prematurely. If it were allowed to perform additional optimization, a better solution would be found. On other occasions, PROC INTPOINT might do too much work. A sufficiently good solution might be reached several iterations before PROC INTPOINT eventually stops.

You can see PROC INTPOINT's progress to the optimum by specifying PRINTLEVEL2=2. PROC INTPOINT will produce a table on the SAS log. A row of the table is generated during each iteration and consists of values of the affine step complementarity, the complementarity of the solution for the next iteration, the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 45), the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 41), and the total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 41). As optimization progresses, the values in all columns should converge to zero.

To tailor stopping criteria to your problem, you can use two sets of parameters: the STOP_x and the KEEPGOING_x parameters. The STOP_x parameters (STOP_C, STOP_DG, STOP_IB, STOP_IC, and STOP_ID) are used to test for some condition at the beginning of each iteration and if met, to stop optimizing immediately. The KEEPGOING_x parameters (KEEPGOING_C, KEEPGOING_DG, KEEPGOING_IB, KEEPGOING_IC, and KEEPGOING_ID) are used when PROC INTPOINT would ordinarily stop optimizing but does not if some conditions are not met.

For the sake of conciseness, a set of options might be referred to as the part of the option name they have in common followed by the suffix x. For example, STOP_C, STOP_DG, STOP_IB, STOP_IC, and STOP_ID will collectively be referred to as STOP_x.

At the beginning of each iteration, PROC INTPOINT will test whether complementarity is <= STOP_C (provided you have specified a STOP_C parameter) and if it is, PROC INTPOINT will stop optimizing. If the duality gap is <= STOP_DG (provided you have specified a STOP_DG parameter), PROC INTPOINT will stop optimizing immediately. This is also true for the other STOP_x parameters that are related to infeasibilities, STOP_IB, STOP_IC, and STOP_ID.

For example, if you want PROC INTPOINT to stop optimizing for the usual stopping conditions, plus the additional condition, complementarity ≤ 100 *or* duality gap ≤ 0.001, then use

```
proc intpoint stop_c=100 stop_dg=0.001
```

If you want PROC INTPOINT to stop optimizing for the usual stopping conditions, plus the additional condition, complementarity ≤ 1000 *and* duality gap ≤ 0.001 *and* constraint infeasibility ≤ 0.0001, then use

```
proc intpoint
   and_stop_c=1000 and_stop_dg=0.01 and_stop_ic=0.0001
```

Unlike the STOP_x parameters that cause PROC INTPOINT to stop optimizing when any one of them is satisfied, the corresponding AND_STOP_x parameters (AND_STOP_C, AND_STOP_DG, AND_STOP_IB, AND_STOP_IC, and AND_STOP_ID) cause PROC INTPOINT to stop only if all (more precisely, all that are specified) options are satisfied. For example, if PROC INTPOINT should stop optimizing when

- complementarity ≤ 100 *or* duality gap ≤ 0.001 or
- complementarity ≤ 1000 *and* duality gap ≤ 0.001 *and* constraint infeasibility ≤ 0.000

then use

```
proc intpoint
   stop_c=100 stop_dg=0.001
   and_stop_c=1000 and_stop_dg=0.01 and_stop_ic=0.0001
```

Just as the STOP_x parameters have AND_STOP_x partners, the KEEPGOING_x parameters have AND_KEEPGOING_x partners. The role of the KEEPGOING_x and AND_KEEPGOING_x parameters is to prevent optimization from stopping too early, even though a usual stopping criteria is met.

When PROC INTPOINT detects that it should stop optimizing for a usual stopping condition, it will perform the following tests:

- It will test whether complementarity is $>$ KEEPGOING_C (provided you have specified a KEEPGOING_C parameter), and if it is, PROC INTPOINT will perform more optimization.
- Otherwise, PROC INTPOINT will then test whether the primal-dual gap is $>$ KEEPGOING_DG (provided you have specified a KEEPGOING_DG parameter), and if it is, PROC INTPOINT will perform more optimization.
- Otherwise, PROC INTPOINT will then test whether the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi} >$ KEEPGOING_IB (provided you have specified a KEEPGOING_IB parameter), and if it is, PROC INTPOINT will perform more optimization.
- Otherwise, PROC INTPOINT will then test whether the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci} >$ KEEPGOING_IC (provided you have specified a KEEPGOING_IC parameter), and if it is, PROC INTPOINT will perform more optimization.
- Otherwise, PROC INTPOINT will then test whether the total dual infeasibility $\sum_{i=1}^{n} infeas_{di} >$ KEEPGOING_ID (provided you have specified a KEEPGOING_ID parameter), and if it is, PROC INTPOINT will perform more optimization.
- Otherwise it will test whether complementarity is $>$ AND_KEEPGOING_C (provided you have specified an AND_KEEPGOING_C parameter), *and* the primal-dual gap is $>$ AND_KEEPGOING_DG (provided you have specified an AND_KEEPGOING_DG parameter), *and* the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi} >$ AND_KEEPGOING_IB (provided you have specified an AND_KEEPGOING_IB parameter), *and* the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci} >$ AND_KEEPGOING_IC (provided you have specified an AND_KEEPGOING_IC parameter), *and* the total dual infeasibility $\sum_{i=1}^{n} infeas_{di} >$ AND_KEEPGOING_ID (provided you have specified an AND_KEEPGOING_ID parameter), and if it is, PROC INTPOINT will perform more optimization.

If all these tests to decide whether more optimization should be performed are false, optimization is stopped.

The following PROC INTPOINT example is used to illustrate how several stopping criteria options can be used together:

```
proc intpoint
   stop_c=1000
   and_stop_c=2000 and_stop_dg=0.01
   and_stop_ib=1 and_stop_ic=1 and_stop_id=1
   keepgoing_c=1500
   and_keepgoing_c=2500 and_keepgoing_dg=0.05
   and_keepgoing_ib=1 and_keepgoing_ic=1 and_keepgoing_id=1
```

At the beginning of each iteration, PROC INTPOINT will stop optimizing if

- complementarity $\leq$ 1000 or
- complementarity $\leq$ 2000 and duality gap $\leq$ 0.01 and the total bound, constraint, and dual infeasibilities are each $\leq$ 1

When PROC INTPOINT determines it should stop optimizing because a usual stopping condition is met, it will stop optimizing only if

- complementarity $\leq$ 1500 or
- complementarity $\leq$ 2500 and duality gap $\leq$ 0.05 and the total bound, constraint, and dual infeasibilities are each $\leq$ 1

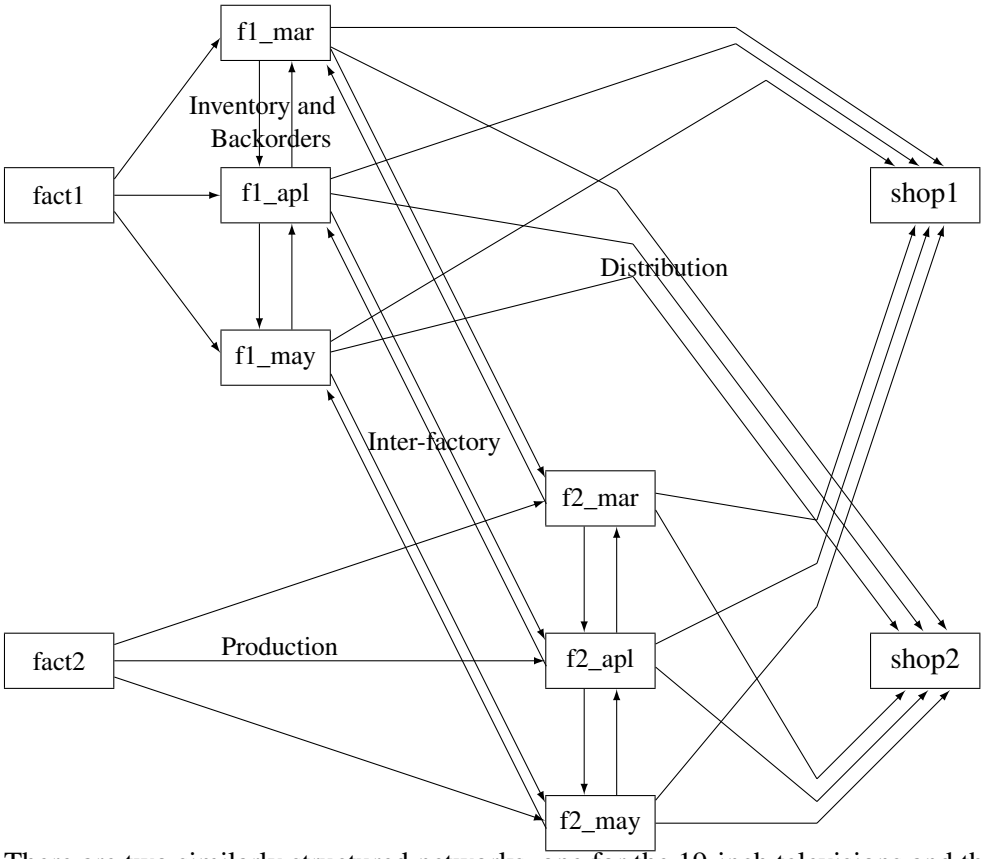
Examples: INTPOINT Procedure

The following examples illustrate some of the capabilities of PROC INTPOINT. These examples, together with the other SAS/OR examples, can be found in the SAS sample library.

In order to illustrate variations in the use of the INTPOINT procedure, Example 4.1 through Example 4.5 use data from a company that produces two sizes of televisions. The company makes televisions with a diagonal screen measurement of either 19 inches or 25 inches. These televisions are made between March and May at both of the company's two factories. Each factory has a limit on the total number of televisions of each screen dimension that can be made during those months.

The televisions are distributed to one of two shops, stored at the factory where they were made, and sold later or shipped to the other factory. Some sets can be used to fill backorders from the previous months. Each shop demands a number of each type of TV for the months of March through May. The following network in Figure 4.12 illustrates the model. Arc costs can be interpreted as production costs, storage costs, backorder penalty costs, inter-factory transportation costs, and sales profits. The arcs can have capacities and lower flow bounds.

Figure 4.12 TV Problem



There are two similarly structured networks, one for the 19-inch televisions and the other for the 25-inch screen TVs. The minimum cost production, inventory, and distribution plan for both TV types can be determined in the same run of PROC INTPOINT. To ensure that node names are unambiguous, the names of nodes in the 19-inch network have suffix _1, and the node names in the 25-inch network have suffix _2.

Example 4.1: Production, Inventory, Distribution Problem

The following code shows how to save a specific problem's data in data sets and solve the model with PROC INTPOINT.

```
title 'Production Planning/Inventory/Distribution';
title2 'Minimum Cost Flow problem';
title3;

data node0;
   input _node_ $ _supdem_ ;
   datalines;
fact1_1   1000
fact2_1    850
fact1_2   1000
fact2_2   1500
shop1_1   -900
shop2_1   -900
```

```
shop1_2   -900
shop2_2  -1450
;

data arc0;
   input _tail_ $ _head_ $ _cost_ _capac_ _lo_ diagonal factory
         key_id $10. mth_made $ _name_&$17. ;
   datalines;
fact1_1  f1_mar_1  127.9  500 50 19 1 production March prod f1 19 mar
fact1_1  f1_apr_1   78.6  600 50 19 1 production April prod f1 19 apl
fact1_1  f1_may_1   95.1  400 50 19 1 production May   .
f1_mar_1 f1_apr_1   15     50  . 19 1 storage    March .
f1_apr_1 f1_may_1   12     50  . 19 1 storage    April .
f1_apr_1 f1_mar_1   28     20  . 19 1 backorder  April back f1 19 apl
f1_may_1 f1_apr_1   28     20  . 19 1 backorder  May   back f1 19 may
f1_mar_1 f2_mar_1   11      .  . 19 . f1_to_2    March .
f1_apr_1 f2_apr_1   11      .  . 19 . f1_to_2    April .
f1_may_1 f2_may_1   16      .  . 19 . f1_to_2    May   .
f1_mar_1 shop1_1  -327.65 250  . 19 1 sales      March .
f1_apr_1 shop1_1  -300    250  . 19 1 sales      April .
f1_may_1 shop1_1  -285    250  . 19 1 sales      May   .
f1_mar_1 shop2_1  -362.74 250  . 19 1 sales      March .
f1_apr_1 shop2_1  -300    250  . 19 1 sales      April .
f1_may_1 shop2_1  -245    250  . 19 1 sales      May   .
fact2_1  f2_mar_1   88.0  450 35 19 2 production March prod f2 19 mar
fact2_1  f2_apr_1   62.4  480 35 19 2 production April prod f2 19 apl
fact2_1  f2_may_1  133.8  250 35 19 2 production May   .
f2_mar_1 f2_apr_1   18     30  . 19 2 storage    March .
f2_apr_1 f2_may_1   20     30  . 19 2 storage    April .
f2_apr_1 f2_mar_1   17     15  . 19 2 backorder  April back f2 19 apl
f2_may_1 f2_apr_1   25     15  . 19 2 backorder  May   back f2 19 may
f2_mar_1 f1_mar_1   10     40  . 19 . f2_to_1    March .
f2_apr_1 f1_apr_1   11     40  . 19 . f2_to_1    April .
f2_may_1 f1_may_1   13     40  . 19 . f2_to_1    May   .
f2_mar_1 shop1_1  -297.4  250  . 19 2 sales      March .
f2_apr_1 shop1_1  -290    250  . 19 2 sales      April .
f2_may_1 shop1_1  -292    250  . 19 2 sales      May   .
f2_mar_1 shop2_1  -272.7  250  . 19 2 sales      March .
f2_apr_1 shop2_1  -312    250  . 19 2 sales      April .
f2_may_1 shop2_1  -299    250  . 19 2 sales      May   .
fact1_2  f1_mar_2  217.9  400 40 25 1 production March prod f1 25 mar
fact1_2  f1_apr_2  174.5  550 50 25 1 production April prod f1 25 apl
fact1_2  f1_may_2  133.3  350 40 25 1 production May   .
f1_mar_2 f1_apr_2   20     40  . 25 1 storage    March .
f1_apr_2 f1_may_2   18     40  . 25 1 storage    April .
f1_apr_2 f1_mar_2   32     30  . 25 1 backorder  April back f1 25 apl
f1_may_2 f1_apr_2   41     15  . 25 1 backorder  May   back f1 25 may
f1_mar_2 f2_mar_2   23      .  . 25 . f1_to_2    March .
f1_apr_2 f2_apr_2   23      .  . 25 . f1_to_2    April .
f1_may_2 f2_may_2   26      .  . 25 . f1_to_2    May   .
f1_mar_2 shop1_2  -559.76   .  . 25 1 sales      March .
f1_apr_2 shop1_2  -524.28   .  . 25 1 sales      April .
f1_may_2 shop1_2  -475.02   .  . 25 1 sales      May   .
f1_mar_2 shop2_2  -623.89   .  . 25 1 sales      March .
```

```
f1_apr_2 shop2_2  -549.68   .   . 25 1 sales      April .
f1_may_2 shop2_2  -460.00   .   . 25 1 sales      May   .
fact2_2  f2_mar_2  182.0  650 35 25 2 production March prod f2 25 mar
fact2_2  f2_apr_2  196.7  680 35 25 2 production April prod f2 25 apl
fact2_2  f2_may_2  201.4  550 35 25 2 production May   .
f2_mar_2 f2_apr_2   28     50  . 25 2 storage    March .
f2_apr_2 f2_may_2   38     50  . 25 2 storage    April .
f2_apr_2 f2_mar_2   31     15  . 25 2 backorder  April back f2 25 apl
f2_may_2 f2_apr_2   54     15  . 25 2 backorder  May   back f2 25 may
f2_mar_2 f1_mar_2   20     25  . 25 . f2_to_1    March .
f2_apr_2 f1_apr_2   21     25  . 25 . f2_to_1    April .
f2_may_2 f1_may_2   43     25  . 25 . f2_to_1    May   .
f2_mar_2 shop1_2  -567.83 500  . 25 2 sales      March .
f2_apr_2 shop1_2  -542.19 500  . 25 2 sales      April .
f2_may_2 shop1_2  -461.56 500  . 25 2 sales      May   .
f2_mar_2 shop2_2  -542.83 500  . 25 2 sales      March .
f2_apr_2 shop2_2  -559.19 500  . 25 2 sales      April .
f2_may_2 shop2_2  -489.06 500  . 25 2 sales      May   .
;

proc intpoint
   bytes=1000000
   printlevel2=2
   nodedata=node0
   arcdata=arc0
   conout=arc1;
   run;

proc print data=arc1 width=min;
   var _tail_ _head_ _cost_ _capac_ _lo_ _flow_ _fcost_
       diagonal factory key_id mth_made;
   sum _fcost_;
   run;
```

The following notes appear on the SAS log:

```
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 21 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 136 .
NOTE: Number of variables= 68 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 20.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 1 constraints from the problem.
NOTE: The preprocessor eliminated 9 constraint coefficients from the problem.
NOTE: 0 columns, 0 rows and 0 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 48 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 20 factor nodes make up 8 supernodes
NOTE: There are 27 nonzero sub-rows or sub-columns outside the supernodal triangular
      regions along the factors leading diagonal.
 Iter  Complem_aff  Complem-ity  Duality_gap  Tot_infeasb  Tot_infeasc  Tot_infeasd
    0    -1.000000    192857968     0.895105        66024        25664            0
    1     37620673     24479828     0.919312  4575.155540  1778.391068            0
    2      4392127      1833947     0.594993            0            0            0
    3       654204       426961     0.249790            0            0            0
    4       161214       108340     0.075186            0            0            0
    5        50985        43146     0.030894            0            0            0
    6        37774        34993     0.025167            0            0            0
    7        17695  9774.172272     0.007114            0            0            0
    8  2421.777663  1427.435257     0.001042            0            0            0
    9   522.394743   240.454270     0.000176            0            0            0
   10    57.447587     7.581156  0.000005540            0            0            0
   11     0.831035     0.007569 5.5317109E-9            0            0            0
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 11
      iterations.
NOTE: Optimum reached.
NOTE: Objective= -1281110.338.
NOTE: The data set WORK.ARC1 has 64 observations and 14 variables.
NOTE: There were 64 observations read from the data set WORK.ARC0.
NOTE: There were 8 observations read from the data set WORK.NODE0.
```

The solution is given in the CONOUT=arc1 data sets. In the CONOUT= data set, shown in Output 4.1.1, the variables diagonal, factory, key_id, and mth_made form an implicit ID list. The diagonal variable has one of two values, 19 or 25. factory also has one of two values, 1 or 2, to denote the factory where either production or storage occurs, from where TVs are either sold to shops or used to satisfy backorders. production, storage, sales, and backorder are values of the key_id variable.

Other values of this variable, f1_to_2 and f2_to_1, are used when flow through arcs represents the transportation of TVs between factories. The mth_made variable has values March, April, and May, the months when TVs that are modeled as flow through an arc were made (assuming that no televisions are stored for more than one month and none manufactured in May are used to fill March backorders).

These ID variables can be used after the PROC INTPOINT run to produce reports and perform analysis on particular parts of the company's operation. For example, reports can be generated for production numbers for each factory; optimal sales figures for each shop; and how many TVs should be stored, used to fill backorders, sent to the other factory, or any combination of these, for TVs with a particular screen, those produced in a particular month, or both.

Output 4.1.1 CONOUT=ARC1

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_	diagonal	factory	key_id	mth_made
1	fact1_1	f1_apr_1	78.60	600	50	600.000	47160.00	19	1	production	April
2	f1_mar_1	f1_apr_1	15.00	50	0	0.000	0.00	19	1	storage	March
3	f1_may_1	f1_apr_1	28.00	20	0	0.000	0.00	19	1	backorder	May
4	f2_apr_1	f1_apr_1	11.00	40	0	0.000	0.00	19	.	f2_to_1	April
5	fact1_2	f1_apr_2	174.50	550	50	550.000	95975.00	25	1	production	April
6	f1_mar_2	f1_apr_2	20.00	40	0	0.000	0.00	25	1	storage	March
7	f1_may_2	f1_apr_2	41.00	15	0	15.000	615.00	25	1	backorder	May
8	f2_apr_2	f1_apr_2	21.00	25	0	0.000	0.00	25	.	f2_to_1	April
9	fact1_1	f1_mar_1	127.90	500	50	344.999	44125.43	19	1	production	March
10	f1_apr_1	f1_mar_1	28.00	20	0	20.000	560.00	19	1	backorder	April
11	f2_mar_1	f1_mar_1	10.00	40	0	40.000	400.00	19	.	f2_to_1	March
12	fact1_2	f1_mar_2	217.90	400	40	400.000	87160.00	25	1	production	March
13	f1_apr_2	f1_mar_2	32.00	30	0	30.000	960.00	25	1	backorder	April
14	f2_mar_2	f1_mar_2	20.00	25	0	25.000	500.00	25	.	f2_to_1	March
15	fact1_1	f1_may_1	95.10	400	50	50.001	4755.06	19	1	production	May
16	f1_apr_1	f1_may_1	12.00	50	0	50.000	600.00	19	1	storage	April
17	f2_may_1	f1_may_1	13.00	40	0	0.000	0.00	19	.	f2_to_1	May
18	fact1_2	f1_may_2	133.30	350	40	40.000	5332.04	25	1	production	May
19	f1_apr_2	f1_may_2	18.00	40	0	0.000	0.00	25	1	storage	April
20	f2_may_2	f1_may_2	43.00	25	0	0.000	0.00	25	.	f2_to_1	May
21	f1_apr_1	f2_apr_1	11.00	99999999	0	30.000	330.00	19	.	f1_to_2	April
22	fact2_1	f2_apr_1	62.40	480	35	480.000	29952.00	19	2	production	April
23	f2_mar_1	f2_apr_1	18.00	30	0	0.000	0.00	19	2	storage	March
24	f2_may_1	f2_apr_1	25.00	15	0	0.000	0.00	19	2	backorder	May
25	f1_apr_2	f2_apr_2	23.00	99999999	0	0.000	0.00	25	.	f1_to_2	April
26	fact2_2	f2_apr_2	196.70	680	35	680.000	133755.99	25	2	production	April
27	f2_mar_2	f2_apr_2	28.00	50	0	0.000	0.00	25	2	storage	March
28	f2_may_2	f2_apr_2	54.00	15	0	15.000	810.00	25	2	backorder	May
29	f1_mar_1	f2_mar_1	11.00	99999999	0	0.000	0.00	19	.	f1_to_2	March
30	fact2_1	f2_mar_1	88.00	450	35	290.000	25520.00	19	2	production	March
31	f2_apr_1	f2_mar_1	17.00	15	0	0.000	0.00	19	2	backorder	April
32	f1_mar_2	f2_mar_2	23.00	99999999	0	0.000	0.00	25	.	f1_to_2	March
33	fact2_2	f2_mar_2	182.00	650	35	645.000	117389.96	25	2	production	March
34	f2_apr_2	f2_mar_2	31.00	15	0	0.000	0.00	25	2	backorder	April
35	f1_may_1	f2_may_1	16.00	99999999	0	100.000	1600.01	19	.	f1_to_2	May
36	fact2_1	f2_may_1	133.80	250	35	35.000	4683.00	19	2	production	May
37	f2_apr_1	f2_may_1	20.00	30	0	15.000	299.99	19	2	storage	April
38	f1_may_2	f2_may_2	26.00	99999999	0	0.000	0.00	25	.	f1_to_2	May
39	fact2_2	f2_may_2	201.40	550	35	35.000	7049.00	25	2	production	May
40	f2_apr_2	f2_may_2	38.00	50	0	0.000	0.00	25	2	storage	April
41	f1_mar_1	shop1_1	-327.65	250	0	154.999	-50785.56	19	1	sales	March
42	f1_apr_1	shop1_1	-300.00	250	0	250.000	-75000.00	19	1	sales	April
43	f1_may_1	shop1_1	-285.00	250	0	0.000	0.00	19	1	sales	May
44	f2_mar_1	shop1_1	-297.40	250	0	250.000	-74349.99	19	2	sales	March
45	f2_apr_1	shop1_1	-290.00	250	0	245.001	-71050.17	19	2	sales	April
46	f2_may_1	shop1_1	-292.00	250	0	0.000	0.00	19	2	sales	May
47	f1_mar_2	shop1_2	-559.76	99999999	0	0.000	0.00	25	1	sales	March
48	f1_apr_2	shop1_2	-524.28	99999999	0	0.000	-0.01	25	1	sales	April

Output 4.1.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_	diagonal	factory	key_id	mth_made
49	f1_may_2	shop1_2	-475.02	99999999	0	25.000	-11875.64	25	1	sales	May
50	f2_mar_2	shop1_2	-567.83	500	0	500.000	-283915.00	25	2	sales	March
51	f2_apr_2	shop1_2	-542.19	500	0	375.000	-203321.08	25	2	sales	April
52	f2_may_2	shop1_2	-461.56	500	0	0.000	0.00	25	2	sales	May
53	f1_mar_1	shop2_1	-362.74	250	0	250.000	-90685.00	19	1	sales	March
54	f1_apr_1	shop2_1	-300.00	250	0	250.000	-75000.00	19	1	sales	April
55	f1_may_1	shop2_1	-245.00	250	0	0.000	0.00	19	1	sales	May
56	f2_mar_1	shop2_1	-272.70	250	0	0.000	0.00	19	2	sales	March
57	f2_apr_1	shop2_1	-312.00	250	0	250.000	-78000.00	19	2	sales	April
58	f2_may_1	shop2_1	-299.00	250	0	150.000	-44850.00	19	2	sales	May
59	f1_mar_2	shop2_2	-623.89	99999999	0	455.000	-283869.94	25	1	sales	March
60	f1_apr_2	shop2_2	-549.68	99999999	0	535.000	-294078.78	25	1	sales	April
61	f1_may_2	shop2_2	-460.00	99999999	0	0.000	0.00	25	1	sales	May
62	f2_mar_2	shop2_2	-542.83	500	0	120.000	-65139.47	25	2	sales	March
63	f2_apr_2	shop2_2	-559.19	500	0	320.000	-178940.96	25	2	sales	April
64	f2_may_2	shop2_2	-489.06	500	0	20.000	-9781.20	25	2	sales	May
							-1281110.34				

Example 4.2: Altering Arc Data

This example examines the effect of changing some of the arc costs. The backorder penalty costs are increased by 20 percent. The sales profit of 25-inch TVs sent to the shops in May is increased by 30 units. The backorder penalty costs of 25-inch TVs manufactured in May for April consumption is decreased by 30 units. The production cost of 19-inch and 25-inch TVs made in May are decreased by 5 units and 20 units, respectively. How does the optimal solution of the network after these arc cost alterations compare with the optimum of the original network?

These SAS statements produce the new NODEDATA= and ARCDATA= data sets:

```
title2 'Minimum Cost Flow problem- Altered Arc Data';
data arc2;
   set arc1;
   oldcost=_cost_;
   oldfc=_fcost_;
   oldflow=_flow_;
   if key_id='backorder'
      then _cost_=_cost_*1.2;
      else if _tail_='f2_may_2' then _cost_=_cost_-30;
   if key_id='production' & mth_made='May' then
      if diagonal=19 then _cost_=_cost_-5;
                     else _cost_=_cost_-20;
   run;

proc intpoint
   bytes=100000
   printlevel2=2
   nodedata=node0
   arcdata=arc2
   conout=arc3;
   run;

proc print data=arc3;
var _tail_ _head_ _capac_ _lo_ _supply_ _demand_ _name_
 _cost_ _flow_ _fcost_ oldcost oldflow oldfc
 diagonal factory key_id mth_made;
 /* to get this variable order */
     sum oldfc _fcost_;
run;
```

The following notes appear on the SAS log:

```
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 21 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 136 .
NOTE: Number of variables= 68 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 20.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 1 constraints from the problem.
NOTE: The preprocessor eliminated 9 constraint coefficients from the problem.
NOTE: 0 columns, 0 rows and 0 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 48 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 20 factor nodes make up 8 supernodes
NOTE: There are 27 nonzero sub-rows or sub-columns outside the supernodal triangular
      regions along the factors leading diagonal.
 Iter  Complem_aff  Complem-ity  Duality_gap  Tot_infeasb  Tot_infeasc  Tot_infeasd
    0    -1.000000    193775969     0.894415        66024        25664            0
    1     37797544     24594220     0.918149  4566.893212  1775.179450            0
    2      4408681      1844606     0.590964            0            0            0
    3       347168       312126     0.194113            0            0            0
    4       145523        86002     0.060330            0            0            0
    5        43008        38240     0.027353            0            0            0
    6        31097        21145     0.015282            0            0            0
    7  9308.807034  4158.399675     0.003029            0            0            0
    8  1710.832075   752.174595     0.000549            0            0            0
    9   254.197112    47.755299  0.000034846            0            0            0
   10     5.252560     0.010692 7.8017564E-9            0            0            0
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 10
      iterations.
NOTE: Optimum reached.
NOTE: Objective= -1285086.442.
NOTE: The data set WORK.ARC3 has 64 observations and 17 variables.
NOTE: There were 64 observations read from the data set WORK.ARC2.
NOTE: There were 8 observations read from the data set WORK.NODE0.
```

The solution is displayed in Output 4.2.1.

Output 4.2.1 CONOUT=ARC3

Minimum Cost Flow Problem- Altered Arc Data

tail	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_
fact1_1	f1_apr_1	600	50	1000	.	prod f1 19 apl	78.60	540.000
f1_mar_1	f1_apr_1	50	0	.	.		15.00	0.000
f1_may_1	f1_apr_1	20	0	.	.	back f1 19 may	33.60	0.000
f2_apr_1	f1_apr_1	40	0	.	.		11.00	0.000
fact1_2	f1_apr_2	550	50	1000	.	prod f1 25 apl	174.50	250.000
f1_mar_2	f1_apr_2	40	0	.	.		20.00	0.000
f1_may_2	f1_apr_2	15	0	.	.	back f1 25 may	49.20	15.000
f2_apr_2	f1_apr_2	25	0	.	.		21.00	0.000
fact1_1	f1_mar_1	500	50	1000	.	prod f1 19 mar	127.90	340.000
f1_apr_1	f1_mar_1	20	0	.	.	back f1 19 apl	33.60	20.000
f2_mar_1	f1_mar_1	40	0	.	.		10.00	40.000
fact1_2	f1_mar_2	400	40	1000	.	prod f1 25 mar	217.90	400.000
f1_apr_2	f1_mar_2	30	0	.	.	back f1 25 apl	38.40	30.000
f2_mar_2	f1_mar_2	25	0	.	.		20.00	25.000
fact1_1	f1_may_1	400	50	1000	.		90.10	115.000
f1_apr_1	f1_may_1	50	0	.	.		12.00	0.000
f2_may_1	f1_may_1	40	0	.	.		13.00	0.000
fact1_2	f1_may_2	350	40	1000	.		113.30	350.000
f1_apr_2	f1_may_2	40	0	.	.		18.00	0.000
f2_may_2	f1_may_2	25	0	.	.		13.00	0.000
f1_apr_1	f2_apr_1	99999999	0	.	.		11.00	20.000
fact2_1	f2_apr_1	480	35	850	.	prod f2 19 apl	62.40	480.000
f2_mar_1	f2_apr_1	30	0	.	.		18.00	0.000
f2_may_1	f2_apr_1	15	0	.	.	back f2 19 may	30.00	0.000
f1_apr_2	f2_apr_2	99999999	0	.	.		23.00	0.000
fact2_2	f2_apr_2	680	35	1500	.	prod f2 25 apl	196.70	680.000
f2_mar_2	f2_apr_2	50	0	.	.		28.00	0.000
f2_may_2	f2_apr_2	15	0	.	.	back f2 25 may	64.80	0.000
f1_mar_1	f2_mar_1	99999999	0	.	.		11.00	0.000
fact2_1	f2_mar_1	450	35	850	.	prod f2 19 mar	88.00	290.000
f2_apr_1	f2_mar_1	15	0	.	.	back f2 19 apl	20.40	0.000
f1_mar_2	f2_mar_2	99999999	0	.	.		23.00	0.000
fact2_2	f2_mar_2	650	35	1500	.	prod f2 25 mar	182.00	635.000
f2_apr_2	f2_mar_2	15	0	.	.	back f2 25 apl	37.20	0.000
f1_may_1	f2_may_1	99999999	0	.	.		16.00	115.000
fact2_1	f2_may_1	250	35	850	.		128.80	35.000
f2_apr_1	f2_may_1	30	0	.	.		20.00	0.000
f1_may_2	f2_may_2	99999999	0	.	.		26.00	335.000
fact2_2	f2_may_2	550	35	1500	.		181.40	35.000
f2_apr_2	f2_may_2	50	0	.	.		38.00	0.000
f1_mar_1	shop1_1	250	0	.	900		-327.65	150.000
f1_apr_1	shop1_1	250	0	.	900		-300.00	250.000
f1_may_1	shop1_1	250	0	.	900		-285.00	0.000
f2_mar_1	shop1_1	250	0	.	900		-297.40	250.000
f2_apr_1	shop1_1	250	0	.	900		-290.00	250.000
f2_may_1	shop1_1	250	0	.	900		-292.00	0.000
f1_mar_2	shop1_2	99999999	0	.	900		-559.76	0.000

Output 4.2.1 *continued*

Minimum Cost Flow Problem- Altered Arc Data

tail	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_
f1_apr_2	shop1_2	99999999	0	.	900		-524.28	0.000
f1_may_2	shop1_2	99999999	0	.	900		-475.02	0.000
f2_mar_2	shop1_2	500	0	.	900		-567.83	500.000
f2_apr_2	shop1_2	500	0	.	900		-542.19	400.000
f2_may_2	shop1_2	500	0	.	900		-491.56	0.000
f1_mar_1	shop2_1	250	0	.	900		-362.74	250.000
f1_apr_1	shop2_1	250	0	.	900		-300.00	250.000
f1_may_1	shop2_1	250	0	.	900		-245.00	0.000
f2_mar_1	shop2_1	250	0	.	900		-272.70	0.000
f2_apr_1	shop2_1	250	0	.	900		-312.00	250.000
f2_may_1	shop2_1	250	0	.	900		-299.00	150.000
f1_mar_2	shop2_2	99999999	0	.	1450		-623.89	455.000
f1_apr_2	shop2_2	99999999	0	.	1450		-549.68	235.000
f1_may_2	shop2_2	99999999	0	.	1450		-460.00	0.000
f2_mar_2	shop2_2	500	0	.	1450		-542.83	110.000
f2_apr_2	shop2_2	500	0	.	1450		-559.19	280.000
f2_may_2	shop2_2	500	0	.	1450		-519.06	370.000

Minimum Cost Flow Problem- Altered Arc Data

Obs	_FCOST_	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made
1	42444.01	78.60	600.000	47160.00	19	1	production	April
2	0.00	15.00	0.000	0.00	19	1	storage	March
3	0.00	28.00	0.000	0.00	19	1	backorder	May
4	0.00	11.00	0.000	0.00	19	.	f2_to_1	April
5	43625.00	174.50	550.000	95975.00	25	1	production	April
6	0.00	20.00	0.000	0.00	25	1	storage	March
7	738.00	41.00	15.000	615.00	25	1	backorder	May
8	0.00	21.00	0.000	0.00	25	.	f2_to_1	April
9	43486.02	127.90	344.999	44125.43	19	1	production	March
10	672.00	28.00	20.000	560.00	19	1	backorder	April
11	400.00	10.00	40.000	400.00	19	.	f2_to_1	March
12	87160.00	217.90	400.000	87160.00	25	1	production	March
13	1152.00	32.00	30.000	960.00	25	1	backorder	April
14	500.00	20.00	25.000	500.00	25	.	f2_to_1	March
15	10361.47	95.10	50.001	4755.06	19	1	production	May
16	0.00	12.00	50.000	600.00	19	1	storage	April
17	0.00	13.00	0.000	0.00	19	.	f2_to_1	May
18	39655.00	133.30	40.000	5332.04	25	1	production	May
19	0.00	18.00	0.000	0.00	25	1	storage	April
20	0.00	43.00	0.000	0.00	25	.	f2_to_1	May
21	220.00	11.00	30.000	330.00	19	.	f1_to_2	April
22	29952.00	62.40	480.000	29952.00	19	2	production	April
23	0.00	18.00	0.000	0.00	19	2	storage	March
24	0.00	25.00	0.000	0.00	19	2	backorder	May
25	0.00	23.00	0.000	0.00	25	.	f1_to_2	April
26	133755.99	196.70	680.000	133755.99	25	2	production	April
27	0.00	28.00	0.000	0.00	25	2	storage	March
28	0.00	54.00	15.000	810.00	25	2	backorder	May
29	0.00	11.00	0.000	0.00	19	.	f1_to_2	March
30	25520.00	88.00	290.000	25520.00	19	2	production	March
31	0.00	17.00	0.000	0.00	19	2	backorder	April
32	0.00	23.00	0.000	0.00	25	.	f1_to_2	March
33	115570.01	182.00	645.000	117389.96	25	2	production	March
34	0.00	31.00	0.000	0.00	25	2	backorder	April
35	1840.00	16.00	100.000	1600.01	19	.	f1_to_2	May
36	4508.00	133.80	35.000	4683.00	19	2	production	May
37	0.00	20.00	15.000	299.99	19	2	storage	April
38	8710.00	26.00	0.000	0.00	25	.	f1_to_2	May
39	6349.00	201.40	35.000	7049.00	25	2	production	May
40	0.00	38.00	0.000	0.00	25	2	storage	April
41	-49147.54	-327.65	154.999	-50785.56	19	1	sales	March
42	-75000.00	-300.00	250.000	-75000.00	19	1	sales	April
43	-0.01	-285.00	0.000	0.00	19	1	sales	May
44	-74350.00	-297.40	250.000	-74349.99	19	2	sales	March
45	-72499.96	-290.00	245.001	-71050.17	19	2	sales	April
46	0.00	-292.00	0.000	0.00	19	2	sales	May
47	0.00	-559.76	0.000	0.00	25	1	sales	March

Minimum Cost Flow Problem- Altered Arc Data

Obs	_FCOST_	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made
48	-0.01	-524.28	0.000	-0.01	25	1	sales	April
49	-0.06	-475.02	25.000	-11875.64	25	1	sales	May
50	-283915.00	-567.83	500.000	-283915.00	25	2	sales	March
51	-216875.92	-542.19	375.000	-203321.08	25	2	sales	April
52	0.00	-461.56	0.000	0.00	25	2	sales	May
53	-90685.00	-362.74	250.000	-90685.00	19	1	sales	March
54	-75000.00	-300.00	250.000	-75000.00	19	1	sales	April
55	0.00	-245.00	0.000	0.00	19	1	sales	May
56	-0.01	-272.70	0.000	0.00	19	2	sales	March
57	-78000.00	-312.00	250.000	-78000.00	19	2	sales	April
58	-44849.99	-299.00	150.000	-44850.00	19	2	sales	May
59	-283869.94	-623.89	455.000	-283869.94	25	1	sales	March
60	-129174.80	-549.68	535.000	-294078.78	25	1	sales	April
61	0.00	-460.00	0.000	0.00	25	1	sales	May
62	-59711.32	-542.83	120.000	-65139.47	25	2	sales	March
63	-156573.27	-559.19	320.000	-178940.96	25	2	sales	April
64	-192052.13	-489.06	20.000	-9781.20	25	2	sales	May
	-1285086.44			**-1281110.34**				

Example 4.3: Adding Side Constraints

The manufacturer of Gizmo chips, which are parts needed to make televisions, can supply only 2,600 chips to factory 1 and 3,750 chips to factory 2 in time for production in each of the months of March and April. However, Gizmo chips will not be in short supply in May. Three chips are required to make each 19-inch TV while the 25-inch TVs require four chips each. To limit the production of televisions produced at factory 1 in March so that the TVs have the correct number of chips, a side constraint called FACT1 MAR GIZMO is used. The form of this constraint is

```
3 * prod f1 19 mar   +   4 * prod f1 25 mar   <=   2600
```

prod f1 19 mar is the name of the arc directed from the node fact1_1 toward node f1_mar_1 and, in the previous constraint, designates the flow assigned to this arc. The ARCDATA= and CONOUT= data sets have arc names in a variable called _name_.

The other side constraints (shown below) are called FACT2 MAR GIZMO, FACT1 APL GIZMO, and FACT2 APL GIZMO.

```
3 * prod f2 19 mar   +   4 * prod f2 25 mar   <=   3750
3 * prod f1 19 apl   +   4 * prod f1 25 apl   <=   2600
3 * prod f2 19 apl   +   4 * prod f2 25 apl   <=   3750
```

To maintain customer goodwill, the total number of backorders is not to exceed 50 sets. The side constraint TOTAL BACKORDER that models this restriction is

```
back f1 19 apl  +  back f1 25 apl  +
back f2 19 apl  +  back f2 25 apl  +
back f1 19 may  +  back f1 25 may  +
back f2 19 may  +  back f2 25 may   <=   50
```

The sparse CONDATA= data set format is used. All side constraints are of less than or equal type. Because this is the default type value for the DEFCONTYPE= option, type information is not necessary in the following CONDATA=con3. Also, DEFCONTYPE= <= does not have to be specified in the PROC INTPOINT statement that follows. Notice that the _column_ variable value CHIP/BO LIMIT indicates that an observation of the con3 data set contains rhs information. Therefore, specify RHSOBS='CHIP/BO LIMIT'

```
title2 'Adding Side Constraints';
data con3;
   input _column_ &$14. _row_ &$15. _coef_ ;
   datalines;
prod f1 19 mar  FACT1 MAR GIZMO  3
prod f1 25 mar  FACT1 MAR GIZMO  4
CHIP/BO LIMIT   FACT1 MAR GIZMO  2600
prod f2 19 mar  FACT2 MAR GIZMO  3
prod f2 25 mar  FACT2 MAR GIZMO  4
CHIP/BO LIMIT   FACT2 MAR GIZMO  3750
prod f1 19 apl  FACT1 APL GIZMO  3
prod f1 25 apl  FACT1 APL GIZMO  4
CHIP/BO LIMIT   FACT1 APL GIZMO  2600
prod f2 19 apl  FACT2 APL GIZMO  3
prod f2 25 apl  FACT2 APL GIZMO  4
CHIP/BO LIMIT   FACT2 APL GIZMO  3750
back f1 19 apl  TOTAL BACKORDER  1
back f1 25 apl  TOTAL BACKORDER  1
back f2 19 apl  TOTAL BACKORDER  1
back f2 25 apl  TOTAL BACKORDER  1
back f1 19 may  TOTAL BACKORDER  1
back f1 25 may  TOTAL BACKORDER  1
back f2 19 may  TOTAL BACKORDER  1
back f2 25 may  TOTAL BACKORDER  1
CHIP/BO LIMIT   TOTAL BACKORDER  50
;
```

The four pairs of data sets that follow can be used as ARCDATA= and NODEDATA= data sets in the following PROC INTPOINT run. The set used depends on which cost information the arcs are to have.

```
ARCDATA=arc0    NODEDATA=node0
ARCDATA=arc1    NODEDATA=node0
ARCDATA=arc2    NODEDATA=node0
ARCDATA=arc3    NODEDATA=node0
```

arc0, node0, and arc1 were created in Example 4.1. The first two data sets are the original input data sets.

In the previous example, arc2 was created by modifying arc1 to reflect different arc costs. arc2 and node0 can also be used as the ARCDATA= and NODEDATA= data sets in a PROC INTPOINT run.

If you are going to continue optimization using the changed arc costs, it is probably best to use arc3 and node0 as the ARCDATA= and NODEDATA= data sets.

PROC INTPOINT is used to find the changed cost network solution that obeys the chip limit and backorder side constraints. An explicit ID list has also been specified so that the variables oldcost, oldfc, and oldflow do not appear in the subsequent output data sets:

```
proc intpoint
   bytes=1000000
   printlevel2=2
   nodedata=node0 arcdata=arc3
   condata=con3 sparsecondata rhsobs='CHIP/BO LIMIT'
   conout=arc4;
   id diagonal factory key_id mth_made;
   run;

proc print data=arc4;
   var _tail_ _head_ _cost_ _capac_ _lo_ _flow_ _fcost_;
       /* to get this variable order */
   sum _fcost_;
   run;
```

The following messages appear on the SAS log:

```
NOTE: The following variables in ARCDATA do not belong to any SAS variable list.
      These will be ignored.
      _FLOW_
      _FCOST_
      oldcost
      oldfc
      oldflow
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: Number of <= side constraints= 5 .
NOTE: Number of == side constraints= 0 .
NOTE: Number of >= side constraints= 0 .
NOTE: Number of side constraint coefficients= 16 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 5 .
NOTE: Number of == constraints= 21 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 152 .
NOTE: Number of variables= 68 .
NOTE: After preprocessing, number of <= constraints= 5.
NOTE: After preprocessing, number of == constraints= 20.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 1 constraints from the problem.
NOTE: The preprocessor eliminated 9 constraint coefficients from the problem.
NOTE: 5 columns, 0 rows and 5 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 74 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 25 factor nodes make up 17 supernodes
NOTE: There are 88 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
```

Iter	Complem_aff	Complem-ity	Duality_gap	Tot_infeasb	Tot_infeasc	Tot_infeasd
0	-1.000000	199456613	0.894741	65408	35351	10906
1	38664128	25735020	0.919726	4738.839318	2561.195456	248.292591
2	5142982	1874540	0.595158	0	0	6.669426
3	366112	338310	0.207256	0	0	1.207816
4	172159	90907	0.063722	0	0	0.238703
5	48403	38889	0.027839	0	0	0.115586
6	28882	17979	0.013029	0	0	0.019825
7	7800.003324	3605.779203	0.002631	0	0	0.004077
8	1564.193112	422.251530	0.000309	0	0	0.000225
9	94.768595	16.589795	0.000012126	0	0	0
10	0.294833	0.001048	5.96523E-10	0	0	0

```
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 10
      iterations.
```

```
NOTE: Optimum reached.
NOTE: Objective= -1282708.622.
NOTE: The data set WORK.ARC4 has 64 observations and 14 variables.
NOTE: There were 64 observations read from the data set WORK.ARC3.
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 21 observations read from the data set WORK.CON3.
```

Output 4.3.1 CONOUT=ARC4

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_
1	fact1_1	f1_apr_1	78.60	600	50	533.333	41920.00
2	f1_mar_1	f1_apr_1	15.00	50	0	0.000	0.00
3	f1_may_1	f1_apr_1	33.60	20	0	0.000	0.00
4	f2_apr_1	f1_apr_1	11.00	40	0	0.000	0.00
5	fact1_2	f1_apr_2	174.50	550	50	250.000	43625.00
6	f1_mar_2	f1_apr_2	20.00	40	0	0.000	0.00
7	f1_may_2	f1_apr_2	49.20	15	0	0.000	0.00
8	f2_apr_2	f1_apr_2	21.00	25	0	0.000	0.00
9	fact1_1	f1_mar_1	127.90	500	50	333.333	42633.33
10	f1_apr_1	f1_mar_1	33.60	20	0	20.000	672.00
11	f2_mar_1	f1_mar_1	10.00	40	0	40.000	400.00
12	fact1_2	f1_mar_2	217.90	400	40	400.000	87160.00
13	f1_apr_2	f1_mar_2	38.40	30	0	30.000	1152.00
14	f2_mar_2	f1_mar_2	20.00	25	0	25.000	500.00
15	fact1_1	f1_may_1	90.10	400	50	128.333	11562.83
16	f1_apr_1	f1_may_1	12.00	50	0	0.000	0.00
17	f2_may_1	f1_may_1	13.00	40	0	0.000	0.00
18	fact1_2	f1_may_2	113.30	350	40	350.000	39655.00
19	f1_apr_2	f1_may_2	18.00	40	0	0.000	0.00
20	f2_may_2	f1_may_2	13.00	25	0	0.000	0.00
21	f1_apr_1	f2_apr_1	11.00	99999999	0	13.333	146.67
22	fact2_1	f2_apr_1	62.40	480	35	480.000	29952.00
23	f2_mar_1	f2_apr_1	18.00	30	0	0.000	0.00
24	f2_may_1	f2_apr_1	30.00	15	0	0.000	0.00
25	f1_apr_2	f2_apr_2	23.00	99999999	0	0.000	0.00
26	fact2_2	f2_apr_2	196.70	680	35	577.500	113594.25
27	f2_mar_2	f2_apr_2	28.00	50	0	0.000	0.00
28	f2_may_2	f2_apr_2	64.80	15	0	0.000	0.00
29	f1_mar_1	f2_mar_1	11.00	99999999	0	0.000	0.00
30	fact2_1	f2_mar_1	88.00	450	35	290.000	25520.00
31	f2_apr_1	f2_mar_1	20.40	15	0	0.000	0.00
32	f1_mar_2	f2_mar_2	23.00	99999999	0	0.000	0.00
33	fact2_2	f2_mar_2	182.00	650	35	650.000	118300.00
34	f2_apr_2	f2_mar_2	37.20	15	0	0.000	0.00
35	f1_may_1	f2_may_1	16.00	99999999	0	115.000	1840.00
36	fact2_1	f2_may_1	128.80	250	35	35.000	4508.00
37	f2_apr_1	f2_may_1	20.00	30	0	0.000	0.00
38	f1_may_2	f2_may_2	26.00	99999999	0	350.000	9100.00
39	fact2_2	f2_may_2	181.40	550	35	122.500	22221.50
40	f2_apr_2	f2_may_2	38.00	50	0	0.000	0.00
41	f1_mar_1	shop1_1	-327.65	250	0	143.333	-46963.16
42	f1_apr_1	shop1_1	-300.00	250	0	250.000	-75000.00
43	f1_may_1	shop1_1	-285.00	250	0	13.333	-3800.00
44	f2_mar_1	shop1_1	-297.40	250	0	250.000	-74350.00
45	f2_apr_1	shop1_1	-290.00	250	0	243.333	-70566.67
46	f2_may_1	shop1_1	-292.00	250	0	0.000	0.00
47	f1_mar_2	shop1_2	-559.76	99999999	0	0.000	0.00
48	f1_apr_2	shop1_2	-524.28	99999999	0	0.000	0.00

Output 4.3.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_
49	f1_may_2	shop1_2	-475.02	99999999	0	0.000	0.00
50	f2_mar_2	shop1_2	-567.83	500	0	500.000	-283915.00
51	f2_apr_2	shop1_2	-542.19	500	0	400.000	-216876.00
52	f2_may_2	shop1_2	-491.56	500	0	0.000	0.00
53	f1_mar_1	shop2_1	-362.74	250	0	250.000	-90685.00
54	f1_apr_1	shop2_1	-300.00	250	0	250.000	-75000.00
55	f1_may_1	shop2_1	-245.00	250	0	0.000	0.00
56	f2_mar_1	shop2_1	-272.70	250	0	0.000	0.00
57	f2_apr_1	shop2_1	-312.00	250	0	250.000	-78000.00
58	f2_may_1	shop2_1	-299.00	250	0	150.000	-44850.00
59	f1_mar_2	shop2_2	-623.89	99999999	0	455.000	-283869.95
60	f1_apr_2	shop2_2	-549.68	99999999	0	220.000	-120929.60
61	f1_may_2	shop2_2	-460.00	99999999	0	0.000	0.00
62	f2_mar_2	shop2_2	-542.83	500	0	125.000	-67853.75
63	f2_apr_2	shop2_2	-559.19	500	0	177.500	-99256.23
64	f2_may_2	shop2_2	-519.06	500	0	472.500	-245255.85
							-1282708.62

Example 4.4: Using Constraints and More Alteration to Arc Data

Suppose the 25-inch screen TVs produced at factory 1 in May can be sold at either shop with an increased profit of 40 dollars each. What is the new optimal solution?

```
title2 'Using Constraints and Altering arc data';
data new_arc4;
   set arc4;
   oldcost=_cost_;
   oldflow=_flow_;
   oldfc=_fcost_;
   if _tail_='f1_may_2' & (_head_='shop1_2' | _head_='shop2_2')
      then _cost_=_cost_-40;
   run;

proc intpoint
bytes=1000000
printlevel2=2
arcdata=new_arc4 nodedata=node0
condata=con3 sparsecondata rhsobs='CHIP/BO LIMIT'
conout=arc5;
run;

title2 'Using Constraints and Altering Arc Data';
proc print data=arc5;
   var _tail_ _head_ _cost_ _capac_ _lo_
       _supply_ _demand_ _name_ _flow_ _fcost_ oldflow oldfc;
          /* to get this variable order */
   sum oldfc _fcost_;
run;
```

The following messages appear on the SAS log:

```
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: Number of <= side constraints= 5 .
NOTE: Number of == side constraints= 0 .
NOTE: Number of >= side constraints= 0 .
NOTE: Number of side constraint coefficients= 16 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 5 .
NOTE: Number of == constraints= 21 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 152 .
NOTE: Number of variables= 68 .
NOTE: After preprocessing, number of <= constraints= 5.
NOTE: After preprocessing, number of == constraints= 20.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 1 constraints from the problem.
NOTE: The preprocessor eliminated 9 constraint coefficients from the problem.
NOTE: 5 columns, 0 rows and 5 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 74 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 25 factor nodes make up 17 supernodes
NOTE: There are 88 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
 Iter  Complem_aff  Complem-ity  Duality_gap  Tot_infeasb  Tot_infeasc  Tot_infeasd
    0    -1.000000    201073760     0.894528        65408        35351        10995
    1     39022799     25967436     0.919693  4741.966761  2562.885742   256.192394
    2      5186078      1844990     0.589523            0            0     6.174556
    3       371920       320310     0.197224            0            0     1.074616
    4       151369        87643     0.060906            0            0     0.267952
    5        35115        25158     0.018017            0            0     0.072961
    6        14667  6194.354873     0.004475            0            0     0.005048
    7  2723.955063  2472.352937     0.001789            0            0     0.001714
    8  1028.390365   280.346187     0.000203            0            0     0.000235
    9    39.957867     5.611483  0.000004063            0            0            0
   10     0.014117     0.000291 9.492733E-11            0            0            0
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 10
      iterations.
NOTE: Optimum reached.
NOTE: Objective= -1295661.8.
NOTE: The data set WORK.ARC5 has 64 observations and 17 variables.
NOTE: There were 64 observations read from the data set WORK.NEW_ARC4.
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 21 observations read from the data set WORK.CON3.
```

Output 4.4.1 CONOUT=ARC5

Using Constraints and Altering Arc Data

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_SUPPLY_	_DEMAND_
1	fact1_1	f1_apr_1	78.60	600	50	1000	.
2	f1_mar_1	f1_apr_1	15.00	50	0	.	.
3	f1_may_1	f1_apr_1	33.60	20	0	.	.
4	f2_apr_1	f1_apr_1	11.00	40	0	.	.
5	fact1_2	f1_apr_2	174.50	550	50	1000	.
6	f1_mar_2	f1_apr_2	20.00	40	0	.	.
7	f1_may_2	f1_apr_2	49.20	15	0	.	.
8	f2_apr_2	f1_apr_2	21.00	25	0	.	.
9	fact1_1	f1_mar_1	127.90	500	50	1000	.
10	f1_apr_1	f1_mar_1	33.60	20	0	.	.
11	f2_mar_1	f1_mar_1	10.00	40	0	.	.
12	fact1_2	f1_mar_2	217.90	400	40	1000	.
13	f1_apr_2	f1_mar_2	38.40	30	0	.	.
14	f2_mar_2	f1_mar_2	20.00	25	0	.	.
15	fact1_1	f1_may_1	90.10	400	50	1000	.
16	f1_apr_1	f1_may_1	12.00	50	0	.	.
17	f2_may_1	f1_may_1	13.00	40	0	.	.
18	fact1_2	f1_may_2	113.30	350	40	1000	.
19	f1_apr_2	f1_may_2	18.00	40	0	.	.
20	f2_may_2	f1_may_2	13.00	25	0	.	.
21	f1_apr_1	f2_apr_1	11.00	99999999	0	.	.
22	fact2_1	f2_apr_1	62.40	480	35	850	.
23	f2_mar_1	f2_apr_1	18.00	30	0	.	.
24	f2_may_1	f2_apr_1	30.00	15	0	.	.
25	f1_apr_2	f2_apr_2	23.00	99999999	0	.	.
26	fact2_2	f2_apr_2	196.70	680	35	1500	.
27	f2_mar_2	f2_apr_2	28.00	50	0	.	.
28	f2_may_2	f2_apr_2	64.80	15	0	.	.
29	f1_mar_1	f2_mar_1	11.00	99999999	0	.	.
30	fact2_1	f2_mar_1	88.00	450	35	850	.
31	f2_apr_1	f2_mar_1	20.40	15	0	.	.
32	f1_mar_2	f2_mar_2	23.00	99999999	0	.	.
33	fact2_2	f2_mar_2	182.00	650	35	1500	.
34	f2_apr_2	f2_mar_2	37.20	15	0	.	.
35	f1_may_1	f2_may_1	16.00	99999999	0	.	.
36	fact2_1	f2_may_1	128.80	250	35	850	.
37	f2_apr_1	f2_may_1	20.00	30	0	.	.
38	f1_may_2	f2_may_2	26.00	99999999	0	.	.
39	fact2_2	f2_may_2	181.40	550	35	1500	.
40	f2_apr_2	f2_may_2	38.00	50	0	.	.
41	f1_mar_1	shop1_1	-327.65	250	0	.	900
42	f1_apr_1	shop1_1	-300.00	250	0	.	900
43	f1_may_1	shop1_1	-285.00	250	0	.	900
44	f2_mar_1	shop1_1	-297.40	250	0	.	900
45	f2_apr_1	shop1_1	-290.00	250	0	.	900
46	f2_may_1	shop1_1	-292.00	250	0	.	900
47	f1_mar_2	shop1_2	-559.76	99999999	0	.	900

Output 4.4.1 *continued*

Using Constraints and Altering Arc Data

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_SUPPLY_	_DEMAND_
48	f1_apr_2	shop1_2	-524.28	99999999	0	.	900
49	f1_may_2	shop1_2	-515.02	99999999	0	.	900
50	f2_mar_2	shop1_2	-567.83	500	0	.	900
51	f2_apr_2	shop1_2	-542.19	500	0	.	900
52	f2_may_2	shop1_2	-491.56	500	0	.	900
53	f1_mar_1	shop2_1	-362.74	250	0	.	900
54	f1_apr_1	shop2_1	-300.00	250	0	.	900
55	f1_may_1	shop2_1	-245.00	250	0	.	900
56	f2_mar_1	shop2_1	-272.70	250	0	.	900
57	f2_apr_1	shop2_1	-312.00	250	0	.	900
58	f2_may_1	shop2_1	-299.00	250	0	.	900
59	f1_mar_2	shop2_2	-623.89	99999999	0	.	1450
60	f1_apr_2	shop2_2	-549.68	99999999	0	.	1450
61	f1_may_2	shop2_2	-500.00	99999999	0	.	1450
62	f2_mar_2	shop2_2	-542.83	500	0	.	1450
63	f2_apr_2	shop2_2	-559.19	500	0	.	1450
64	f2_may_2	shop2_2	-519.06	500	0	.	1450

Using Constraints and Altering Arc Data

Obs	_name_	_FLOW_	_FCOST_	oldflow	oldfc
1	prod f1 19 apl	533.333	41920.00	533.333	41920.00
2		0.000	0.00	0.000	0.00
3	back f1 19 may	0.000	0.00	0.000	0.00
4		0.000	0.00	0.000	0.00
5	prod f1 25 apl	250.000	43625.00	250.000	43625.00
6		0.000	0.00	0.000	0.00
7	back f1 25 may	0.000	0.00	0.000	0.00
8		0.000	0.00	0.000	0.00
9	prod f1 19 mar	333.333	42633.33	333.333	42633.33
10	back f1 19 apl	20.000	672.00	20.000	672.00
11		40.000	400.00	40.000	400.00
12	prod f1 25 mar	400.000	87160.00	400.000	87160.00
13	back f1 25 apl	30.000	1152.00	30.000	1152.00
14		25.000	500.00	25.000	500.00
15		128.333	11562.83	128.333	11562.83
16		0.000	0.00	0.000	0.00
17		0.000	0.00	0.000	0.00
18		350.000	39655.00	350.000	39655.00
19		0.000	0.00	0.000	0.00
20		0.000	0.00	0.000	0.00
21		13.333	146.67	13.333	146.67
22	prod f2 19 apl	480.000	29952.00	480.000	29952.00
23		0.000	0.00	0.000	0.00
24	back f2 19 may	0.000	0.00	0.000	0.00
25		0.000	0.00	0.000	0.00
26	prod f2 25 apl	550.000	108185.00	577.500	113594.25
27		0.000	0.00	0.000	0.00
28	back f2 25 may	0.000	0.00	0.000	0.00
29		0.000	0.00	0.000	0.00
30	prod f2 19 mar	290.000	25520.00	290.000	25520.00
31	back f2 19 apl	0.000	0.00	0.000	0.00
32		0.000	0.00	0.000	0.00
33	prod f2 25 mar	650.000	118300.00	650.000	118300.00
34	back f2 25 apl	0.000	0.00	0.000	0.00
35		115.000	1840.00	115.000	1840.00
36		35.000	4508.00	35.000	4508.00
37		0.000	0.00	0.000	0.00
38		0.000	0.00	350.000	9100.00
39		150.000	27210.00	122.500	22221.50
40		0.000	0.00	0.000	0.00
41		143.333	-46963.17	143.333	-46963.16
42		250.000	-75000.00	250.000	-75000.00
43		13.333	-3800.00	13.333	-3800.00
44		250.000	-74350.00	250.000	-74350.00
45		243.333	-70566.67	243.333	-70566.67
46		0.000	0.00	0.000	0.00
47		0.000	0.00	0.000	0.00

Using Constraints and Altering Arc Data

Obs	_name_	_FLOW_	_FCOST_	oldflow	oldfc
48		0.000	0.00	0.000	0.00
49		350.000	-180257.00	0.000	0.00
50		500.000	-283915.00	500.000	-283915.00
51		50.000	-27109.50	400.000	-216876.00
52		0.000	0.00	0.000	0.00
53		250.000	-90685.00	250.000	-90685.00
54		250.000	-75000.00	250.000	-75000.00
55		0.000	0.00	0.000	0.00
56		0.000	0.00	0.000	0.00
57		250.000	-78000.00	250.000	-78000.00
58		150.000	-44850.00	150.000	-44850.00
59		455.000	-283869.95	455.000	-283869.95
60		220.000	-120929.60	220.000	-120929.60
61		0.000	0.00	0.000	0.00
62		125.000	-67853.75	125.000	-67853.75
63		500.000	-279595.00	177.500	-99256.23
64		150.000	-77859.00	472.500	-245255.85
			-1295661.80		**-1282708.62**

Example 4.5: Nonarc Variables in the Side Constraints

You can verify that the FACT2 MAR GIZMO constraint has a left-hand-side activity of 3,470, which is not equal to the _RHS_ of this constraint. Not all of the 3,750 chips that can be supplied to factory 2 for March production are used. It is suggested that all the possible chips be obtained in March and those not used be saved for April production. Because chips must be kept in an air-controlled environment, it costs one dollar to store each chip purchased in March until April. The maximum number of chips that can be stored in this environment at each factory is 150. In addition, a search of the parts inventory at factory 1 turned up 15 chips available for their March production.

Nonarc variables are used in the side constraints that handle the limitations of supply of Gizmo chips. A nonarc variable called f1 unused chips has as a value the number of chips that are not used at factory 1 in March. Another nonarc variable, f2 unused chips, has as a value the number of chips that are not used at factory 2 in March. f1 chips from mar has as a value the number of chips left over from March used for production at factory 1 in April. Similarly, f2 chips from mar has as a value the number of chips left over from March used for April production at factory 2 in April. The last two nonarc variables have objective function coefficients of 1 and upper bounds of 150. The Gizmo side constraints are

```
3*prod f1 19 mar + 4*prod f1 25 mar + f1 unused chips = 2615
3*prod f2 19 apl + 4*prod f2 25 apl + f2 unused chips = 3750
3*prod f1 19 apl + 4*prod f1 25 apl - f1 chips from mar = 2600
3*prod f2 19 apl + 4*prod f2 25 apl - f2 chips from mar = 3750
f1 unused chips + f2 unused chips -
f1 chips from mar - f2 chips from mar >= 0
```

The last side constraint states that the number of chips not used in March is not less than the number of chips left over from March and used in April. Here, this constraint is called CHIP LEFTOVER.

The following SAS code creates a new data set containing constraint data. It seems that most of the constraints are now equalities, so you specify DEFCONTYPE=EQ in the PROC INTPOINT statement from now on and provide constraint type data for constraints that are not "equal to" type, using the default TYPEOBS value _TYPE_ as the _COLUMN_ variable value to indicate observations that contain constraint type data. Also, from now on, the default RHSOBS value is used.

```
title2 'Nonarc Variables in the Side Constraints';
data con6;
   input _column_ &$17. _row_ &$15. _coef_ ;
   datalines;
prod f1 19 mar     FACT1 MAR GIZMO  3
prod f1 25 mar     FACT1 MAR GIZMO  4
f1 unused chips    FACT1 MAR GIZMO  1
_RHS_              FACT1 MAR GIZMO  2615
prod f2 19 mar     FACT2 MAR GIZMO  3
prod f2 25 mar     FACT2 MAR GIZMO  4
f2 unused chips    FACT2 MAR GIZMO  1
_RHS_              FACT2 MAR GIZMO  3750
prod f1 19 apl     FACT1 APL GIZMO  3
prod f1 25 apl     FACT1 APL GIZMO  4
f1 chips from mar  FACT1 APL GIZMO  -1
_RHS_              FACT1 APL GIZMO  2600
prod f2 19 apl     FACT2 APL GIZMO  3
prod f2 25 apl     FACT2 APL GIZMO  4
f2 chips from mar  FACT2 APL GIZMO  -1
_RHS_              FACT2 APL GIZMO  3750
f1 unused chips    CHIP LEFTOVER    1
f2 unused chips    CHIP LEFTOVER    1
f1 chips from mar  CHIP LEFTOVER    -1
f2 chips from mar  CHIP LEFTOVER    -1
_TYPE_             CHIP LEFTOVER    1
back f1 19 apl     TOTAL BACKORDER  1
back f1 25 apl     TOTAL BACKORDER  1
back f2 19 apl     TOTAL BACKORDER  1
back f2 25 apl     TOTAL BACKORDER  1
back f1 19 may     TOTAL BACKORDER  1
back f1 25 may     TOTAL BACKORDER  1
back f2 19 may     TOTAL BACKORDER  1
back f2 25 may     TOTAL BACKORDER  1
_TYPE_             TOTAL BACKORDER  -1
_RHS_              TOTAL BACKORDER  50
;
```

The nonarc variables f1 chips from mar and f2 chips from mar have objective function coefficients of 1 and upper bounds of 150. There are various ways in which this information can be furnished to PROC INTPOINT. If there were a TYPE list variable in the CONDATA= data set, observations could be in the form

```
_COLUMN_           _TYPE_  _ROW_ _COEF_
f1 chips from mar  objfn      .      1
f1 chips from mar  upperbd    .    150
f2 chips from mar  objfn      .      1
f2 chips from mar  upperbd    .    150
```

It is desirable to assign ID list variable values to all the nonarc variables:

```
data arc6;
   input _tail_ $ _head_ $ _cost_ _capac_ _lo_  diagonal factory
         key_id $10. mth_made $ _name_&$17.;
datalines;
fact1_1  f1_apr_1   78.60 600 50 19 1 production April prod f1 19 apl
f1_mar_1 f1_apr_1   15.00  50  . 19 1 storage    March .
f1_may_1 f1_apr_1   33.60  20  . 19 1 backorder  May   back f1 19 may
f2_apr_1 f1_apr_1   11.00  40  . 19 . f2_to_1    April .
fact1_2  f1_apr_2  174.50 550 50 25 1 production April prod f1 25 apl
f1_mar_2 f1_apr_2   20.00  40  . 25 1 storage    March .
f1_may_2 f1_apr_2   49.20  15  . 25 1 backorder  May   back f1 25 may
f2_apr_2 f1_apr_2   21.00  25  . 25 . f2_to_1    April .
fact1_1  f1_mar_1  127.90 500 50 19 1 production March prod f1 19 mar
f1_apr_1 f1_mar_1   33.60  20  . 19 1 backorder  April back f1 19 apl
f2_mar_1 f1_mar_1   10.00  40  . 19 . f2_to_1    March .
fact1_2  f1_mar_2  217.90 400 40 25 1 production March prod f1 25 mar
f1_apr_2 f1_mar_2   38.40  30  . 25 1 backorder  April back f1 25 apl
f2_mar_2 f1_mar_2   20.00  25  . 25 . f2_to_1    March .
fact1_1  f1_may_1   90.10 400 50 19 1 production May   .
f1_apr_1 f1_may_1   12.00  50  . 19 1 storage    April .
f2_may_1 f1_may_1   13.00  40  . 19 . f2_to_1    May   .
fact1_2  f1_may_2  113.30 350 40 25 1 production May   .
f1_apr_2 f1_may_2   18.00  40  . 25 1 storage    April .
f2_may_2 f1_may_2   13.00  25  . 25 . f2_to_1    May   .
f1_apr_1 f2_apr_1   11.00   .  . 19 . f1_to_2    April .
fact2_1  f2_apr_1   62.40 480 35 19 2 production April prod f2 19 apl
f2_mar_1 f2_apr_1   18.00  30  . 19 2 storage    March .
f2_may_1 f2_apr_1   30.00  15  . 19 2 backorder  May   back f2 19 may
f1_apr_2 f2_apr_2   23.00   .  . 25 . f1_to_2    April .
fact2_2  f2_apr_2  196.70 680 35 25 2 production April prod f2 25 apl
f2_mar_2 f2_apr_2   28.00  50  . 25 2 storage    March .
f2_may_2 f2_apr_2   64.80  15  . 25 2 backorder  May   back f2 25 may
f1_mar_1 f2_mar_1   11.00   .  . 19 . f1_to_2    March .
fact2_1  f2_mar_1   88.00 450 35 19 2 production March prod f2 19 mar
f2_apr_1 f2_mar_1   20.40  15  . 19 2 backorder  April back f2 19 apl
f1_mar_2 f2_mar_2   23.00   .  . 25 . f1_to_2    March .
fact2_2  f2_mar_2  182.00 650 35 25 2 production March prod f2 25 mar
f2_apr_2 f2_mar_2   37.20  15  . 25 2 backorder  April back f2 25 apl
f1_may_1 f2_may_1   16.00   .  . 19 . f1_to_2    May   .
fact2_1  f2_may_1  128.80 250 35 19 2 production May   .
f2_apr_1 f2_may_1   20.00  30  . 19 2 storage    April .
f1_may_2 f2_may_2   26.00   .  . 25 . f1_to_2    May   .
fact2_2  f2_may_2  181.40 550 35 25 2 production May   .
f2_apr_2 f2_may_2   38.00  50  . 25 2 storage    April .
f1_mar_1 shop1_1  -327.65 250  . 19 1 sales      March .
f1_apr_1 shop1_1  -300.00 250  . 19 1 sales      April .
f1_may_1 shop1_1  -285.00 250  . 19 1 sales      May   .
f2_mar_1 shop1_1  -297.40 250  . 19 2 sales      March .
f2_apr_1 shop1_1  -290.00 250  . 19 2 sales      April .
f2_may_1 shop1_1  -292.00 250  . 19 2 sales      May   .
f1_mar_2 shop1_2  -559.76   .  . 25 1 sales      March .
f1_apr_2 shop1_2  -524.28   .  . 25 1 sales      April .
```

```
f1_may_2 shop1_2  -515.02   .   . 25 1 sales      May   .
f2_mar_2 shop1_2  -567.83 500   . 25 2 sales      March .
f2_apr_2 shop1_2  -542.19 500   . 25 2 sales      April .
f2_may_2 shop1_2  -491.56 500   . 25 2 sales      May   .
f1_mar_1 shop2_1  -362.74 250   . 19 1 sales      March .
f1_apr_1 shop2_1  -300.00 250   . 19 1 sales      April .
f1_may_1 shop2_1  -245.00 250   . 19 1 sales      May   .
f2_mar_1 shop2_1  -272.70 250   . 19 2 sales      March .
f2_apr_1 shop2_1  -312.00 250   . 19 2 sales      April .
f2_may_1 shop2_1  -299.00 250   . 19 2 sales      May   .
f1_mar_2 shop2_2  -623.89   .   . 25 1 sales      March .
f1_apr_2 shop2_2  -549.68   .   . 25 1 sales      April .
f1_may_2 shop2_2  -500.00   .   . 25 1 sales      May   .
f2_mar_2 shop2_2  -542.83 500   . 25 2 sales      March .
f2_apr_2 shop2_2  -559.19 500   . 25 2 sales      April .
f2_may_2 shop2_2  -519.06 500   . 25 2 sales      May   .
;

data arc6;
   set arc5;
   drop oldcost oldfc oldflow _flow_ _fcost_ ;
   run;

data arc6_b;
   input _name_ &$17. _cost_ _capac_ factory key_id $ ;
   datalines;
f1 unused chips    .    . 1 chips
f2 unused chips    .    . 2 chips
f1 chips from mar  1 150 1 chips
f2 chips from mar  1 150 2 chips
;


proc append force
   base=arc6 data=arc6_b;
   run;
proc intpoint
   bytes=1000000
   printlevel2=2
   nodedata=node0 arcdata=arc6
   condata=con6 defcontype=eq sparsecondata
   conout=arc7;
   run;
```

The following messages appear on the SAS log:

```
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: Number of nonarc variables= 4 .
NOTE: Number of <= side constraints= 1 .
NOTE: Number of == side constraints= 4 .
NOTE: Number of >= side constraints= 1 .
NOTE: Number of side constraint coefficients= 24 .
NOTE: The following messages relate to the equivalent Linear Programming problem
      solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 1 .
NOTE: Number of == constraints= 25 .
NOTE: Number of >= constraints= 1 .
NOTE: Number of constraint coefficients= 160 .
NOTE: Number of variables= 72 .
NOTE: After preprocessing, number of <= constraints= 1.
NOTE: After preprocessing, number of == constraints= 24.
NOTE: After preprocessing, number of >= constraints= 1.
NOTE: The preprocessor eliminated 1 constraints from the problem.
NOTE: The preprocessor eliminated 9 constraint coefficients from the problem.
NOTE: 2 columns, 0 rows and 2 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 78 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 26 factor nodes make up 18 supernodes
NOTE: There are 101 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
 Iter  Complem_aff  Complem-ity  Duality_gap  Tot_infeasb  Tot_infeasc  Tot_infeasd
    0    -1.000000    210688061     0.904882        69336        35199  4398.024971
    1     54066756     35459986     0.931873  5967.706945  3029.541352   935.225890
    2     10266927      2957978     0.671565            0            0    36.655485
    3       326659       314818     0.177750            0            0     3.893178
    4       137432        83570     0.053111            0            0     0.852994
    5        41386        26985     0.017545            0            0     0.204166
    6        12451  6063.528974     0.003973            0            0     0.041229
    7  2962.309960  1429.369437     0.000939            0            0     0.004395
    8   352.469864   233.620884     0.000153            0            0     0.000297
    9   115.012309    23.329492  0.000015331            0            0            0
   10     1.754859     0.039304 2.5828261E-8            0            0            0
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 10
      iterations.
NOTE: Optimum reached.
NOTE: Objective= -1295542.717.
NOTE: The data set WORK.ARC7 has 68 observations and 14 variables.
NOTE: There were 68 observations read from the data set WORK.ARC6.
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 31 observations read from the data set WORK.CON6.
```

The optimal solution data set, CONOUT=ARC7, is given in Output 4.5.1.

```
proc print data=arc7;
   var _tail_ _head_ _name_ _cost_ _capac_ _lo_
       _flow_ _fcost_;
   sum _fcost_;
   run;
```

The optimal value of the nonarc variable f2 unused chips is 280. This means that although there are 3,750 chips that can be used at factory 2 in March, only 3,470 are used. As the optimal value of f1 unused chips is zero, all chips available for production in March at factory 1 are used. The nonarc variable f2 chips from mar also has zero optimal value. This means that the April production at factory 2 does not need any chips that could have been held in inventory since March. However, the nonarc variable f1 chips from mar has value of 20. Thus, 3,490 chips should be ordered for factory 2 in March. Twenty of these chips should be held in inventory until April, then sent to factory 1.

Output 4.5.1 CONOUT=ARC7

Obs	_tail_	_head_	_name_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_
1	fact1_1	f1_apr_1	prod f1 19 apl	78.60	600	50	540.000	42444.00
2	f1_mar_1	f1_apr_1		15.00	50	0	0.000	0.00
3	f1_may_1	f1_apr_1	back f1 19 may	33.60	20	0	0.000	0.00
4	f2_apr_1	f1_apr_1		11.00	40	0	0.000	0.00
5	fact1_2	f1_apr_2	prod f1 25 apl	174.50	550	50	250.000	43625.01
6	f1_mar_2	f1_apr_2		20.00	40	0	0.000	0.00
7	f1_may_2	f1_apr_2	back f1 25 may	49.20	15	0	0.000	0.00
8	f2_apr_2	f1_apr_2		21.00	25	0	25.000	525.00
9	fact1_1	f1_mar_1	prod f1 19 mar	127.90	500	50	338.333	43272.81
10	f1_apr_1	f1_mar_1	back f1 19 apl	33.60	20	0	20.000	672.00
11	f2_mar_1	f1_mar_1		10.00	40	0	40.000	400.00
12	fact1_2	f1_mar_2	prod f1 25 mar	217.90	400	40	400.000	87159.99
13	f1_apr_2	f1_mar_2	back f1 25 apl	38.40	30	0	30.000	1152.00
14	f2_mar_2	f1_mar_2		20.00	25	0	25.000	500.00
15	fact1_1	f1_may_1		90.10	400	50	116.667	10511.68
16	f1_apr_1	f1_may_1		12.00	50	0	0.000	0.00
17	f2_may_1	f1_may_1		13.00	40	0	0.000	0.00
18	fact1_2	f1_may_2		113.30	350	40	350.000	39655.00
19	f1_apr_2	f1_may_2		18.00	40	0	0.000	0.00
20	f2_may_2	f1_may_2		13.00	25	0	0.000	0.00
21	f1_apr_1	f2_apr_1		11.00	99999999	0	20.000	220.00
22	fact2_1	f2_apr_1	prod f2 19 apl	62.40	480	35	480.000	29952.00
23	f2_mar_1	f2_apr_1		18.00	30	0	0.000	0.00
24	f2_may_1	f2_apr_1	back f2 19 may	30.00	15	0	0.000	0.00
25	f1_apr_2	f2_apr_2		23.00	99999999	0	0.000	0.00
26	fact2_2	f2_apr_2	prod f2 25 apl	196.70	680	35	577.500	113594.25
27	f2_mar_2	f2_apr_2		28.00	50	0	0.000	0.00
28	f2_may_2	f2_apr_2	back f2 25 may	64.80	15	0	0.000	0.00
29	f1_mar_1	f2_mar_1		11.00	99999999	0	0.000	0.00
30	fact2_1	f2_mar_1	prod f2 19 mar	88.00	450	35	290.000	25520.00
31	f2_apr_1	f2_mar_1	back f2 19 apl	20.40	15	0	0.000	0.00
32	f1_mar_2	f2_mar_2		23.00	99999999	0	0.000	0.00
33	fact2_2	f2_mar_2	prod f2 25 mar	182.00	650	35	650.000	118300.00
34	f2_apr_2	f2_mar_2	back f2 25 apl	37.20	15	0	0.000	0.00
35	f1_may_1	f2_may_1		16.00	99999999	0	115.000	1840.00
36	fact2_1	f2_may_1		128.80	250	35	35.000	4508.00
37	f2_apr_1	f2_may_1		20.00	30	0	0.000	0.00
38	f1_may_2	f2_may_2		26.00	99999999	0	0.000	0.00
39	fact2_2	f2_may_2		181.40	550	35	122.500	22221.50
40	f2_apr_2	f2_may_2		38.00	50	0	0.000	0.00
41	f1_mar_1	shop1_1		-327.65	250	0	148.333	-48601.35
42	f1_apr_1	shop1_1		-300.00	250	0	250.000	-75000.00
43	f1_may_1	shop1_1		-285.00	250	0	1.667	-475.01
44	f2_mar_1	shop1_1		-297.40	250	0	250.000	-74350.00
45	f2_apr_1	shop1_1		-290.00	250	0	250.000	-72500.00
46	f2_may_1	shop1_1		-292.00	250	0	0.000	-0.05
47	f1_mar_2	shop1_2		-559.76	99999999	0	0.000	0.00
48	f1_apr_2	shop1_2		-524.28	99999999	0	0.000	0.00

Output 4.5.1 *continued*

Obs	_tail_	_head_	_name_	_cost_	_capac_	_lo_	_FLOW_	_FCOST_
49	f1_may_2	shop1_2		-515.02	99999999	0	347.500	-178969.34
50	f2_mar_2	shop1_2		-567.83	500	0	500.000	-283914.98
51	f2_apr_2	shop1_2		-542.19	500	0	52.500	-28465.09
52	f2_may_2	shop1_2		-491.56	500	0	0.000	0.00
53	f1_mar_1	shop2_1		-362.74	250	0	250.000	-90684.99
54	f1_apr_1	shop2_1		-300.00	250	0	250.000	-75000.00
55	f1_may_1	shop2_1		-245.00	250	0	0.000	-0.00
56	f2_mar_1	shop2_1		-272.70	250	0	0.000	-0.01
57	f2_apr_1	shop2_1		-312.00	250	0	250.000	-78000.00
58	f2_may_1	shop2_1		-299.00	250	0	150.000	-44850.00
59	f1_mar_2	shop2_2		-623.89	99999999	0	455.000	-283869.90
60	f1_apr_2	shop2_2		-549.68	99999999	0	245.000	-134671.54
61	f1_may_2	shop2_2		-500.00	99999999	0	2.500	-1250.00
62	f2_mar_2	shop2_2		-542.83	500	0	125.000	-67853.77
63	f2_apr_2	shop2_2		-559.19	500	0	500.000	-279594.99
64	f2_may_2	shop2_2		-519.06	500	0	122.500	-63584.94
65			f1 chips from mar	1.00	150	0	20.000	20.00
66			f1 unused chips	0.00	99999999	0	0.001	0.00
67			f2 chips from mar	1.00	150	0	0.000	0.00
68			f2 unused chips	0.00	99999999	0	280.000	0.00
								-1295542.72

Example 4.6: Solving an LP Problem with Data in MPS Format

In this example, PROC INTPOINT is ultimately used to solve an LP. But prior to that, there is SAS code that is used to read a MPS format file and initialize an input SAS data set. MPS was an optimization package developed for IBM computers many years ago and the format by which data had to be supplied to that system became the industry standard for other optimization software packages, including those developed recently. The MPS format is described in Murtagh (1981). If you have an LP which has data in MPS format in a file /your-directory/your-filename.dat, then the following SAS code should be run:

```
filename w '/your-directory/your-filename.dat';
data raw;
   infile w lrecl=80 pad;
   input field1 $ 2-3 field2 $ 5-12 field3 $ 15-22
         field4  25-36 field5 $ 40-47 field6 50-61;
   run;
%sasmpsxs;
data lp;
   set;
   if _type_="FREE" then _type_="MIN";
   if lag(_type_)="*HS" then _type_="RHS";
   run;
proc sort data=lp;
   by _col_;
   run;
```

```
proc intpoint
   arcdata=lp
   condata=lp sparsecondata rhsobs=rhs grouped=condata
   conout=solutn  /* SAS data set for the optimal solution */
   bytes=20000000
   nnas=1700 ncoefs=4000 ncons=700
   printlevel2=2 memrep;
   run;
proc lp
   data=lp sparsedata
   endpause time=3600 maxit1=100000 maxit2=100000;
   run;
   show status;
   quit;
```

You will have to specify the appropriate path and file name in which your MPS format data resides.

SASMPSXS is a SAS macro provided within SAS/OR software. The MPS format resembles the sparse format of the CONDATA= data set for PROC INTPOINT. The SAS macro SASMPSXS examines the MPS data and transfers it into a SAS data set while automatically taking into account how the MPS format differs slightly from PROC INTPOINT's sparse format.

The parameters NNAS=1700, NCOEFS=4000, and NCONS=700 indicate the approximate (overestimated) number of variables, coefficients and constraints this model has. You must change these to your problems dimensions. Knowing these, PROC INTPOINT is able to utilize memory better and read the data faster. These parameters are optional.

The PROC SORT preceding PROC INTPOINT is not necessary, but sorting the SAS data set can speed up PROC INTPOINT when it reads the data. After the sort, data for each column is grouped together. GROUPED=condata can be specified.

For small problems, presorting and specifying those additional options is not going to greatly influence PROC INTPOINT's run time. However, when problems are large, presorting and specifying those additional options can be very worthwhile.

If you generate the model yourself, you will be familiar enough with it to know what to specify for the RHSOBS= parameter. If the value of the SAS variable in the COLUMN list is equal to the character string specified as the RHSOBS= option, the data in that observation is interpreted as right-hand-side data as opposed to coefficient data. If you do not know what to specify for the RHSOBS= option, you should first run PROC LP and optionally set MAXIT1=1 and MAXIT2=1. PROC LP will output a Problem Summary that includes the line

```
Rhs Variable    rhs-charstr
```

BYTES=20000000 is the size of working memory PROC INTPOINT is allowed.

The options PRINTLEVEL2=2 and MEMREP indicate that you want to see an iteration log and messages about memory usage. Specifying these options is optional.

Example 4.7: Converting to an MPS-Format SAS Data Set

This example demonstrates the use of the MPSOUT= option to convert a problem data set in PROC INTPOINT input format into an MPS-format SAS data set for use with the OPTLP procedure.

Suppose you want to solve a linear program with the following formulation:

$$
\begin{array}{rrcrcrcr}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & \\
\text{subject to} & & - & 2x_2 & - & 3x_3 & \geq & -5 \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 \\
& x_1 & + & 2x_2 & + & 3x_3 & \geq & 7 \\
& & 0 & \leq & x_1 & \leq & 10 & \\
& & 0 & \leq & x_2 & \leq & 15 & \\
& & 0 & \leq & x_3 & \leq & 20 &
\end{array}
$$

You can save the LP in dense format by using the following DATA step:

```
data exdata;
   input x1 x2 x3 _type_ $ _rhs_;
datalines;
2 -3 -4 min      .
. -2 -3 >=      -5
1  1  2 <=       6
1  2  3 >=       7
10 15 20 upperbd  .
;
```

If you decide to solve the problem by using the OPTLP procedure, you need to convert the data set exdata from dense format to MPS format. You can accomplish this by using the following statements:

```
proc intpoint condata=exdata mpsout=mpsdata bytes=100000;
run;
```

The MPS-format SAS data set mpsdata is shown in Output 4.7.1.

Output 4.7.1 Data Set mpsdata

Obs	field1	field2	field3	field4	field5	field6
1	NAME		modname	.		.
2	ROWS			.		.
3	MIN	objfn		.		.
4	G	_OBS2_		.		.
5	L	_OBS3_		.		.
6	G	_OBS4_		.		.
7	COLUMNS			.		.
8		x1	objfn	2	_OBS3_	1
9		x1	_OBS4_	1		.
10		x2	objfn	-3	_OBS2_	-2
11		x2	_OBS3_	1	_OBS4_	2
12		x3	objfn	-4	_OBS2_	-3
13		x3	_OBS3_	2	_OBS4_	3
14	RHS			.		.
15			_OBS2_	-5	_OBS3_	6
16			_OBS4_	7		.
17	BOUNDS			.		.
18	UP	bdsvect	x1	10		.
19	UP	bdsvect	x2	15		.
20	UP	bdsvect	x3	20		.
21	ENDATA			.		.

The constraint names _OBS2_, _OBS3_, and _OBS4_ are generated by the INTPOINT procedure. If you want to provide your own constraint names, use the ROW list variable in the CONOUT= data set. If you specify the problem data in sparse format instead of dense format, the MPSOUT= option produces the same MPS-format SAS data set shown in the preceding output.

Now that the problem data are in MPS format, you can solve the problem by using the OPTLP procedure. For more information, see Chapter 12, "The OPTLP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

Example 4.8: Migration to OPTMODEL: Production, Inventory, Distribution

The following example shows how to solve Example 4.1 using PROC OPTMODEL. The input data sets are the same as in that example.

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called ARC1:

```
proc optmodel;
   set <str> NODES;
   num _supdem_ {NODES} init 0;
   read data node0 into NODES=[_node_] _supdem_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num diagonal {ARCS};
   num factory {ARCS};
   str key_id {ARCS};
   str mth_made {ARCS};
   str _name_ {ARCS};

   read data arc0 nomiss into ARCS=[_tail_ _head_] _lo_ _capac_ _cost_
      diagonal factory key_id mth_made _name_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   num infinity = constant('BIG');
   num excess = sum {i in NODES} _supdem_[i];
   if (excess > 0) then do;
      /* change equality constraint to le constraint */
      for {i in NODES: _supdem_[i] > 0} balance[i].lb = -infinity;
   end;
   else if (excess < 0) then do;
      /* change equality constraint to ge constraint */
      for {i in NODES: _supdem_[i] < 0} balance[i].ub = infinity;
   end;

   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data arc1 from [_tail_ _head_]
```

```
      _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_=Flow _fcost_
      diagonal factory key_id mth_made;
quit;
```

The statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The _SUPDEM_, _LO_, and _CAPAC_ parameters are given initial values, and the NOMISS option in the READ DATA statement tells OPTMODEL to read only the nonmissing values from the input data set. The balance constraint is initially declared as an equality, but depending on the total supply or demand, the sense of this constraint is changed to "≤" or "≥" by relaxing the constraint's lower or upper bound, respectively. The ARC1 output data set contains the same information as in Example 4.1.

The PROC PRINT statements are the same as in Example 4.1:

```
proc print data=arc1 width=min;
   var _tail_ _head_ _cost_ _capac_ _lo_ _flow_ _fcost_
   diagonal factory key_id mth_made;
   sum _fcost_;
run;
```

The output is displayed in Output 4.8.1.

Output 4.8.1 Output Data Set

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_flow_	_fcost_	diagonal	factory	key_id	mth_made
1	fact1_1	f1_mar_1	127.90	500	50	345	44125.50	19	1	production	March
2	fact1_1	f1_apr_1	78.60	600	50	600	47160.00	19	1	production	April
3	fact1_1	f1_may_1	95.10	400	50	50	4755.00	19	1	production	May
4	f1_mar_1	f1_apr_1	15.00	50	0	0	0.00	19	1	storage	March
5	f1_apr_1	f1_may_1	12.00	50	0	50	600.00	19	1	storage	April
6	f1_apr_1	f1_mar_1	28.00	20	0	20	560.00	19	1	backorder	April
7	f1_may_1	f1_apr_1	28.00	20	0	0	0.00	19	1	backorder	May
8	f1_mar_1	f2_mar_1	11.00	.	0	0	0.00	19	.	f1_to_2	March
9	f1_apr_1	f2_apr_1	11.00	.	0	30	330.00	19	.	f1_to_2	April
10	f1_may_1	f2_may_1	16.00	.	0	100	1600.00	19	.	f1_to_2	May
11	f1_mar_1	shop1_1	-327.65	250	0	155	-50785.75	19	1	sales	March
12	f1_apr_1	shop1_1	-300.00	250	0	250	-75000.00	19	1	sales	April
13	f1_may_1	shop1_1	-285.00	250	0	0	0.00	19	1	sales	May
14	f1_mar_1	shop2_1	-362.74	250	0	250	-90685.00	19	1	sales	March
15	f1_apr_1	shop2_1	-300.00	250	0	250	-75000.00	19	1	sales	April
16	f1_may_1	shop2_1	-245.00	250	0	0	0.00	19	1	sales	May
17	fact2_1	f2_mar_1	88.00	450	35	290	25520.00	19	2	production	March
18	fact2_1	f2_apr_1	62.40	480	35	480	29952.00	19	2	production	April
19	fact2_1	f2_may_1	133.80	250	35	35	4683.00	19	2	production	May
20	f2_mar_1	f2_apr_1	18.00	30	0	0	0.00	19	2	storage	March
21	f2_apr_1	f2_may_1	20.00	30	0	15	300.00	19	2	storage	April
22	f2_apr_1	f2_mar_1	17.00	15	0	0	0.00	19	2	backorder	April
23	f2_may_1	f2_apr_1	25.00	15	0	0	0.00	19	2	backorder	May
24	f2_mar_1	f1_mar_1	10.00	40	0	40	400.00	19	.	f2_to_1	March
25	f2_apr_1	f1_apr_1	11.00	40	0	0	0.00	19	.	f2_to_1	April
26	f2_may_1	f1_may_1	13.00	40	0	0	0.00	19	.	f2_to_1	May
27	f2_mar_1	shop1_1	-297.40	250	0	250	-74350.00	19	2	sales	March
28	f2_apr_1	shop1_1	-290.00	250	0	245	-71050.00	19	2	sales	April
29	f2_may_1	shop1_1	-292.00	250	0	0	0.00	19	2	sales	May
30	f2_mar_1	shop2_1	-272.70	250	0	0	0.00	19	2	sales	March
31	f2_apr_1	shop2_1	-312.00	250	0	250	-78000.00	19	2	sales	April
32	f2_may_1	shop2_1	-299.00	250	0	150	-44850.00	19	2	sales	May
33	fact1_2	f1_mar_2	217.90	400	40	400	87160.00	25	1	production	March
34	fact1_2	f1_apr_2	174.50	550	50	550	95975.00	25	1	production	April
35	fact1_2	f1_may_2	133.30	350	40	40	5332.00	25	1	production	May
36	f1_mar_2	f1_apr_2	20.00	40	0	0	0.00	25	1	storage	March
37	f1_apr_2	f1_may_2	18.00	40	0	0	0.00	25	1	storage	April
38	f1_apr_2	f1_mar_2	32.00	30	0	30	960.00	25	1	backorder	April
39	f1_may_2	f1_apr_2	41.00	15	0	15	615.00	25	1	backorder	May
40	f1_mar_2	f2_mar_2	23.00	.	0	0	0.00	25	.	f1_to_2	March
41	f1_apr_2	f2_apr_2	23.00	.	0	0	0.00	25	.	f1_to_2	April
42	f1_may_2	f2_may_2	26.00	.	0	0	0.00	25	.	f1_to_2	May
43	f1_mar_2	shop1_2	-559.76	.	0	0	0.00	25	1	sales	March
44	f1_apr_2	shop1_2	-524.28	.	0	0	0.00	25	1	sales	April
45	f1_may_2	shop1_2	-475.02	.	0	25	-11875.50	25	1	sales	May
46	f1_mar_2	shop2_2	-623.89	.	0	455	-283869.95	25	1	sales	March
47	f1_apr_2	shop2_2	-549.68	.	0	535	-294078.80	25	1	sales	April
48	f1_may_2	shop2_2	-460.00	.	0	0	0.00	25	1	sales	May

Output 4.8.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_flow_	_fcost_	diagonal	factory	key_id	mth_made
49	fact2_2	f2_mar_2	182.00	650	35	645	117390.00	25	2	production	March
50	fact2_2	f2_apr_2	196.70	680	35	680	133756.00	25	2	production	April
51	fact2_2	f2_may_2	201.40	550	35	35	7049.00	25	2	production	May
52	f2_mar_2	f2_apr_2	28.00	50	0	0	0.00	25	2	storage	March
53	f2_apr_2	f2_may_2	38.00	50	0	0	0.00	25	2	storage	April
54	f2_apr_2	f2_mar_2	31.00	15	0	0	0.00	25	2	backorder	April
55	f2_may_2	f2_apr_2	54.00	15	0	15	810.00	25	2	backorder	May
56	f2_mar_2	f1_mar_2	20.00	25	0	25	500.00	25	.	f2_to_1	March
57	f2_apr_2	f1_apr_2	21.00	25	0	0	0.00	25	.	f2_to_1	April
58	f2_may_2	f1_may_2	43.00	25	0	0	0.00	25	.	f2_to_1	May
59	f2_mar_2	shop1_2	-567.83	500	0	500	-283915.00	25	2	sales	March
60	f2_apr_2	shop1_2	-542.19	500	0	375	-203321.25	25	2	sales	April
61	f2_may_2	shop1_2	-461.56	500	0	0	0.00	25	2	sales	May
62	f2_mar_2	shop2_2	-542.83	500	0	120	-65139.60	25	2	sales	March
63	f2_apr_2	shop2_2	-559.19	500	0	320	-178940.80	25	2	sales	April
64	f2_may_2	shop2_2	-489.06	500	0	20	-9781.20	25	2	sales	May
							-1281110.35				

The optimal objective value is the same as in Example 4.1. The log is displayed in Output 4.8.2.

Output 4.8.2 OPTMODEL Log

```
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 64 observations read from the data set WORK.ARC0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 64 variables (0 free, 0 fixed).
NOTE: The problem has 20 linear constraints (4 LE, 16 EQ, 0 GE, 0 range).
NOTE: The problem has 128 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 64 variables, 20 constraints, and 128
      constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                            Objective
      Phase Iteration         Value         Time
       D 1          1    0.000000E+00          0
       D 2          2   -4.020320E+06          0
       D 2         32   -1.281110E+06          0
NOTE: Optimal.
NOTE: Objective = -1281110.35.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.ARC1 has 64 observations and 14 variables.
```

References

George, J. A., Liu, J. W., and Ng, E. (2001), "Computer Solution of Positive Definite Systems," Unpublished manuscript obtainable from authors.

Lustig, I. J., Marsten, R. E., and Shanno, D. F. (1992), "On Implementing Mehrotra's Predictor-Corrector Interior-Point Method for Linear Programming," *SIAM Journal on Optimization*, 2, 435–449.

Murtagh, B. A. (1981), *Advanced Linear Programming: Computation and Practice*, New York: McGraw-Hill.

Reid, J. K. (1975), *A Sparsity-Exploiting Variant of the Bartels-Golub Decomposition for Linear Programming Bases*, Technical Report Harwell CSS 20, Atomic Energy Research Establishment, Harwell, UK.

Roos, C., Terlaky, T., and Vial, J. (1997), *Theory and Algorithms for Linear Optimization*, Chichester, UK: John Wiley & Sons.

Wright, S. J. (1997), *Primal-Dual Interior-Point Methods*, Philadelphia: SIAM Publications.

Ye, Y. (1996), *Interior Point Algorithms: Theory and Analysis*, New York: John Wiley & Sons.

Chapter 5
The LP Procedure

Contents

Overview: LP Procedure **168**
Getting Started: LP Procedure **170**
An Introductory Example 171
An Integer Programming Example 176
An MPS Format to Sparse Format Conversion Example 177
Syntax: LP Procedure **178**
Functional Summary 179
PROC LP Statement 183
COEF Statement 192
COL Statement 192
ID Statement 193
IPIVOT Statement 193
PIVOT Statement 193
PRINT Statement 193
QUIT Statement 195
RANGE Statement 195
RESET Statement 196
RHS Statement 197
RHSSEN Statement 197
ROW Statement 197
RUN Statement 198
SHOW Statement 198
TYPE Statement 198
VAR Statement 200
Details: LP Procedure **201**
Missing Values 201
Dense Data Input Format 201
Sparse Data Input Format 201
Converting Any PROC LP Format to an MPS-Format SAS Data Set 203
Converting Standard MPS Format to Sparse Format 204
The Reduced Costs, Dual Activities, and Current Tableau 206
Macro Variable _ORLP_ 207
Pricing 208
Scaling 209
Preprocessing 209
Integer Programming 210

Sensitivity Analysis . . . 217
Range Analysis . . . 219
Parametric Programming . . . 220
Interactive Facilities . . . 221
Memory Management . . . 223
Output Data Sets . . . 223
Input Data Sets . . . 226
Displayed Output . . . 226
ODS Table and Variable Names . . . 230
Memory Limit . . . 231
Examples: LP Procedure . . . **232**
Example 5.1: An Oil Blending Problem . . . 233
Example 5.2: A Sparse View of the Oil Blending Problem . . . 238
Example 5.3: Sensitivity Analysis: Changes in Objective Coefficients . . . 241
Example 5.4: Additional Sensitivity Analysis . . . 244
Example 5.5: Price Parametric Programming for the Oil Blending Problem . . . 247
Example 5.6: Special Ordered Sets and the Oil Blending Problem . . . 251
Example 5.7: Goal-Programming a Product Mix Problem . . . 254
Example 5.8: A Simple Integer Program . . . 260
Example 5.9: An Infeasible Problem . . . 263
Example 5.10: Restarting an Integer Program . . . 266
Example 5.11: Alternative Search of the Branch-and-Bound Tree . . . 272
Example 5.12: An Assignment Problem . . . 277
Example 5.13: A Scheduling Problem . . . 282
Example 5.14: A Multicommodity Transshipment Problem with Fixed Charges . . . 289
Example 5.15: Converting to an MPS-Format SAS Data Set . . . 292
Example 5.16: Migration to OPTMODEL: Assignment . . . 294
Example 5.17: Migration to OPTMODEL: Multicommodity Transshipment . . . 296
References . . . **299**

Overview: LP Procedure

The LP procedure solves linear programs, integer programs, and mixed-integer programs. It also performs parametric programming, range analysis, and reports on solution sensitivity to changes in the right-hand-side constants and price coefficients.

The LP procedure provides various control options and solution strategies. It also provides the functionality to produce various kinds of intermediate and final solution information. The procedure's interactive features enable you to take control of the problem solving process. During linear or integer iterations, for example, you can stop the procedure at intermediate stages and examine current results. If necessary, you can change options or strategies and resume the execution of the procedure.

The LP procedure is used to optimize a linear function subject to linear and integer constraints. Specifically, the LP procedure solves the general mixed-integer program of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \{\geq, =, \leq\} b \\ & \ell \leq x \leq u \\ & x_i \text{ is integer}, i \in \mathcal{S} \end{array}$$

where

- A is an $m \times n$ matrix of technological coefficients
- b is an $m \times 1$ matrix of right-hand-side (RHS) constants
- c is an $n \times 1$ matrix of objective function coefficients
- x is an $n \times 1$ matrix of structural variables
- l is an $n \times 1$ matrix of lower bounds on x
- u is an $n \times 1$ matrix of upper bounds on x
- $\mathcal{S}$ is a subset of the set of indices $\{1, \ldots, n\}$

Linear programs (when $\mathcal{S}$ is empty) are denoted by (LP). For these problems, the procedure employs the two-phase revised simplex method, which uses the Bartels-Golub update of the LU decomposed basis matrix to pivot between feasible solutions (Bartels 1971). In phase 1, PROC LP finds a basic feasible solution to (LP), while in phase 2, PROC LP finds an optimal solution, x^{opt}. The procedure implicitly handles unrestricted variables, lower-bounded variables, upper-bounded variables, and ranges on constraints. When no explicit lower bounds are specified, PROC LP assumes that all variables are bounded below by zero.

When a variable is specified as an integer variable, $\mathcal{S}$ has at least one element. The procedure then uses the branch-and-bound technique for optimization.

The relaxed problem (the problem with no integer constraints) is solved initially using the primal algorithm described previously. Constraints are added in defining the subsequent descendant problems in the branch-and-bound tree. These problems are then solved using the dual simplex algorithm. Dual pivots are referred to as phase 3 pivots.

The preprocessing option enables the procedure to identify redundant and infeasible constraints, fix variables, and reduce the feasible region before solving a problem. For linear programs, the option often can reduce the number of constraints and variables, leading to a quicker elapsed solution time and improved reliability. For integer programs, it often reduces the gap between an integer program and its relaxed linear program, which will likely lead to a reduced branch-and-bound tree and a quicker CPU time. In general, it provides users an alternative to solving large, complicated operations research problems.

The LP procedure can also analyze the sensitivity of the solution x^{opt} to changes in both the objective function and the right-hand-side constants. There are three techniques available for this analysis: sensitivity analysis, parametric programming, and range analysis. Sensitivity analysis enables you to examine the size of a perturbation to the right-hand-side or objective vector by an arbitrary change vector for which the basis of the current optimal solution remains optimal.

Parametric programming, on the other hand, enables you to specify the size of the perturbation beforehand and examine how the optimal solution changes as the desired perturbation is realized. With this technique,

the procedure pivots to maintain optimality as the right-hand-side or objective vector is perturbed beyond the range for which the current solution is optimal. Range analysis is used to examine the range of each right-hand-side value or objective coefficient for which the basis of the current optimal solution remains optimal.

The LP procedure can also save both primal and dual solutions, the current tableau, and the branch-and-bound tree in SAS data sets. This enables you to generate solution reports and perform additional analyses with the SAS System. Although PROC LP reports solutions, this feature is particularly useful for reporting solutions in formats tailored to your specific needs. Saving computational results in a data set also enables you to continue executing a problem not solved because of insufficient time or other computational problems.

The LP procedure uses the Output Delivery System (ODS), a SAS subsystem that provides capabilities for displaying and controlling the output from SAS procedures. ODS enables you to modify the headers, column names, data formats, and layouts of the output tables in PROC LP.

There are no restrictions on the problem size in the LP procedure. The number of constraints and variables in a problem that PROC LP can solve depends on the host platform, the available memory, and the available disk space for utility data sets.

You can also solve LP problems by using the OPTLP procedure. The OPTLP procedure requires a linear program to be specified using a SAS data set that adheres to the MPS format, a widely accepted format in the optimization community. You can use the MPSOUT= option in the LP procedure to convert typical PROC LP format data sets into MPS-format SAS data sets.

Getting Started: LP Procedure

PROC LP expects the definition of one or more linear, integer, or mixed-integer programs in an input data set. There are two formats, a dense format and a sparse format, for this data set.

In the dense format, a model is expressed in a similar way as it is formulated. Each SAS variable corresponds to a model's column, and each SAS observation corresponds to a model's row. A SAS variable in the input data set is one of the following:

- a type variable
- an id variable
- a structural variable
- a right-hand-side variable
- a right-hand-side sensitivity analysis variable or
- a range variable

The type variable tells PROC LP how to interpret the observation as a part of the mathematical programming problem. It identifies and classifies objectives, constraints, and the rows that contain information of variables like types, bounds, and so on. PROC LP recognizes the following keywords as values for the type variable: MIN, MAX, EQ, LE, GE, SOSEQ, SOSLE, UNRSTRT, LOWERBD, UPPERBD, FIXED, INTEGER, BINARY, BASIC, PRICESEN, and FREE. The values of the id variable are the names of the rows in the model. The other variables identify and classify the columns with numerical values.

The sparse format to PROC LP is designed to enable you to specify only the nonzero coefficients in the description of linear programs, integer programs, and mixed-integer programs. The SAS data set that describes the sparse model must contain at least four SAS variables:

- a type variable
- a column variable
- a row variable and
- a coefficient variable

Each observation in the data set associates a type with a row or a column, or defines a coefficient or a numerical value in the model, or both. In addition to the keywords in the dense format, PROC LP also recognizes the keywords RHS, RHSSEN, and RANGE as values of the type variable. The values of the row and column variables are the names of the rows and columns in the model. The values of the coefficient variables give the coefficients or other numerical data. The SAS data set can contain multiple pairs of row and coefficient variables. In this way, more information about the model can be specified in each observation in the data set. See the section "Sparse Data Input Format" on page 201 for further discussion.

With both the dense and sparse formats for model specification, the observation order is not important. This feature is particularly useful when using the sparse model input.

An Introductory Example

A simple blending problem illustrates the dense and sparse input formats and the use of PROC LP. A step in refining crude oil into finished oil products involves a distillation process that splits crude into various streams. Suppose there are three types of crude available: Arabian light, Arabian heavy, and Brega. These types of crude are distilled into light naphtha, intermediate naphtha, and heating oil. These in turn are blended into jet fuel using one of two recipes. What amounts of the three crudes maximize the profit from producing jet fuel? A formulation to answer this question is as follows:

$$\text{maximize} \quad -175\,\text{a_light} - 165\,\text{a_heavy} - 205\,\text{brega} + 300\,\text{jet_1} + 300\,\text{jet_2}$$

$$
\begin{aligned}
\text{subject to} \quad .035\,\text{a_light} + .03\,\text{a_heavy} + .045\,\text{brega} &= \text{naphthal} \\
.1\,\text{a_light} + .075\,\text{a_heavy} + .135\,\text{brega} &= \text{naphthai} \\
.39\,\text{a_light} + .3\,\text{a_heavy} + .43\,\text{brega} &= \text{heatingo} \\
.3\,\text{naphthai} + .7\,\text{heatingo} &= \text{jet_1} \\
.2\,\text{naphthal} + .8\,\text{heatingo} &= \text{jet_2} \\
\text{a_light} &\leq 110 \\
\text{a_heavy} &\leq 165 \\
\text{brega} &\leq 80 \\
\text{a_light}, \text{a_heavy}, \text{brega}, \text{naphthai}, & \\
\text{naphthal}, \text{heatingo}, \text{jet_1}, \text{jet_2} &\geq 0
\end{aligned}
$$

The following data set gives the representation of this formulation. Notice that the variable names are the structural variables, the rows are the constraints, and the coefficients are given as the values for the structural variables.

```
data;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2
         _type_ $ _rhs_;
   datalines;
profit            -175 -165 -205  0  0  0 300 300 max     .
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq     0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq     0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq     0
recipe_1             0    0    0  0 .3 .7  -1   0  eq     0
recipe_2             0    0    0 .2  0 .8   0  -1  eq     0
available          110  165   80  .  .  .   .   . upperbd .
;
```

The same model can be specified in the sparse format, as follows. This format enables you to omit the zero coefficients.

```
data;
   format _type_ $8. _col_ $8. _row_ $16. ;
   input _type_ $ _col_ $ _row_ $ _coef_ ;
   datalines;
max      .            profit                   .
eq       .            napha_l_conv             .
eq       .            napha_i_conv             .
eq       .            heating_oil_conv         .
eq       .            recipe_1                 .
eq       .            recipe_2                 .
upperbd  .            available                .
.        a_light      profit                -175
.        a_light      napha_l_conv           .035
.        a_light      napha_i_conv           .100
```

```
.        a_light       heating_oil_conv       .390
.        a_light       available               110
.        a_heavy       profit                 -165
.        a_heavy       napha_l_conv           .030
.        a_heavy       napha_i_conv           .075
.        a_heavy       heating_oil_conv       .300
.        a_heavy       available               165
.        brega         profit                 -205
.        brega         napha_l_conv           .045
.        brega         napha_i_conv           .135
.        brega         heating_oil_conv       .430
.        brega         available               80
.        naphthal      napha_l_conv           -1
.        naphthal      recipe_2                .2
.        naphthai      napha_i_conv           -1
.        naphthai      recipe_1                .3
.        heatingo      heating_oil_conv       -1
.        heatingo      recipe_1                .7
.        heatingo      recipe_2                .8
.        jet_1         profit                 300
.        jet_1         recipe_1               -1
.        jet_2         profit                 300
.        jet_2         recipe_2               -1
.        _rhs_         recipe_1                0
;
```

Because the input order of the model into PROC LP is unimportant, this model can be specified in sparse input in arbitrary row order. Example 5.2 in the section "Examples: LP Procedure" on page 232 demonstrates this.

The dense and sparse forms of model input give you flexibility to generate models using the SAS language. The dense form of the model is solved with the statements

```
proc lp;
run;
```

The sparse form is solved with the statements

```
proc lp sparsedata;
run;
```

Example 5.1 and Example 5.2 in the section "Examples: LP Procedure" on page 232 continue with this problem.

Problem Input

As default, PROC LP uses the most recently created SAS data set as the problem input data set. However, if you want to input the problem from a specific SAS data set, use the DATA= option. For example, if the previous dense form data set has the name DENSE, the PROC LP statements can be written as

```
proc lp data=dense;
run;
```

Problem Definition Statements

In the previous dense form data set, the _ID_, _TYPE_, and _RHS_ variables are special variables in PROC LP. They stand for id variable, type variable, and right-hand-side variable. If you replace those variable names with, for example, ROWNAME, TYPE, and RHS, you need the problem definition statements (ID, TYPE and RHS) in PROC LP:

```
proc lp;
   id rowname;
   type type;
   rhs rhs;
run;
```

Other special variables for the dense format are _RHSSEN_ and _RANGE_, which identify the vectors for the right-hand-side sensitivity and range analyses. The corresponding statements are the RHSSEN and RANGE statements. (Notice that a variable name can be identical to a statement name.)

In the same way, if you replace the variables _COL_, _ROW_, _TYPE_, and _COEF_ in the previous sparse form data set by COLUMN, ROW, TYPE, and COEF, you need the problem definition statements (COL, ROW, TYPE, and COEF) in PROC LP.

```
proc lp sparsedata;
   col column;
   row row;
   type type;
   coef coef;
run;
```

In the sparse form data set, the value '_RHS_' under the variable _COL_ is a special column name, which represents the model's right-hand-side column. If you replace it by a value 'R', the PROC LP statements would be

```
proc lp sparsedata;
   rhs r;
run;
```

Other special column names for the sparse format are '_RHSSEN_' and '_RANGE_'. The corresponding statements are the RHSSEN and RANGE statements.

PROC LP is case insensitive to variable names and all character values, including the row and column names in the sparse format. The order of the problem definition statements is not important.

For the dense format, a model's row names appear as character values in a SAS data set. For the sparse format, both the row and the column names of the model appear as character values in the data set. Thus, you can put spaces or other special characters in the names. When referring to these names in the problem definition statement or other LP statements, you must use single or double quotes around them. For example, if you replace '_RHS_' by 'R H S' in the previous sparse form data set, the PROC LP statements would become

```
proc lp sparsedata;
   rhs "r h s";
run;
```

LP Options

The specifications SPARSEDATA and DATA= in the previous examples are PROC LP options. PROC LP options include

- data set options
- display control options
- interactive control options
- preprocessing control options
- branch-and-bound control options
- sensitivity/parametric/ranging control options
- simplex algorithm control options

Interactive Processing

Interactive control options include READPAUSE, ENDPAUSE, and so forth. You can run PROC LP interactively using those options. For example, for the blending problem example in the dense form, you can first pause the procedure before iterations start with the READPAUSE option. The PROC LP statements are

```
proc lp readpause;
run;
```

When the procedure pauses, you run the PRINT statement to display the initial technological matrix and see if the input is correct. Then you run the PIVOT statement to do one simplex pivot and pause. After that you use the SHOW statement to check the current solution status. Then you apply the RESET statement to tell the procedure to stop as soon as it finds a solution. Now you use the RUN statement to continue the execution. When the procedure stops, you run the PRINT statement again to do a price range analysis and QUIT the procedure. Use a SAS %PUT statement to display the contents of PROC LP's macro variable, _ORLP_, which contains iterations and solution information. What follows are the complete statements in batch mode:

```
proc lp readpause;
run;
print matrix(,); /* display all rows and columns. */
pivot;
show status;
reset endpause;
run;
print rangeprice;
quit;
%put &_orlp_;
```

NOTE: You can force PROC LP to pause during iterations by using the CTRL-BREAK key.

An Integer Programming Example

The following is a simple mixed-integer programming problem. Details can be found in Example 5.8 in the section "Examples: LP Procedure" on page 232.

```
data;
   format _row_ $10.;
   input _row_ $ choco gumdr ichoco igumdr _type_ $ _rhs_;
   datalines;
object        .25    .75   -100    -75 max        .
cooking        15     40      0      0 le     27000
color           0  56.25      0      0 le     27000
package     18.75      0      0      0 le     27000
condiments     12     50      0      0 le     27000
chocolate       1      0 -10000      0 le         0
gum             0      1      0 -10000 le         0
only_one        0      0      1      1 eq         1
binary          .      .      1      2 binary     .
;
```

The row with 'binary' type indicates that this problem is a mixed-integer program and all the integer variables are binary. The integer values of the row set an ordering for PROC LP to pick the branching variable when VARSELECT=PRIOR is chosen. Smaller values will have higher priorities. The _ROW_ variable here is an alias of the _ID_ variable.

This problem can be solved with the following statements:

```
proc lp canselect=lifo backtrack=obj varselect=far endpause;
run;
quit;
%put &_orlp_;
```

The options CANSELECT=, BACKTRACK=, and VARSELECT= specify the rules for picking the next active problem and the rule to choose the branching variable. In this example, the values LIFO, OBJ and FAR serve as the default values, so the three options can be omitted from the PROC LP statement. The following is the output from the %PUT statement:

```
STATUS=SUCCESSFUL PHASE=3 OBJECTIVE=285 P_FEAS=YES D_FEAS=YES INT_ITER=3
      INT_FEAS=2 ACTIVE=0 INT_BEST=285 PHASE1_ITER=0 PHASE2_ITER=5
      PHASE3_ITER=5
```

Preprocessing

Using the PREPROCESS= option, you can apply the preprocessing techniques to pre-solve and then solve the preceding mixed-integer program:

```
proc lp preprocess=1 endpause;
run;
quit;
%put &_orlp_;
```

The preprocessing statistics are written to the SAS log file as follows:

```
NOTE: Preprocessing 1 ...
NOTE:     2 upper bounds decreased.
NOTE:     2 coefficients reduced.
NOTE: Preprocessing 2 ...
NOTE:     2 constraints eliminated.
NOTE: Preprocessing done.
```

The new output _ORLP_ is as follows:

```
STATUS=SUCCESSFUL PHASE=3 OBJECTIVE=285 P_FEAS=YES D_FEAS=YES INT_ITER=0
      INT_FEAS=1 ACTIVE=0 INT_BEST=285 PHASE1_ITER=0 PHASE2_ITER=5
      PHASE3_ITER=0
```

In this example, the number of integer iterations (INT_ITER=) is zero, which means that the preprocessing has reduced the gap between the relaxed linear problem and the mixed-integer program to zero.

An MPS Format to Sparse Format Conversion Example

If your model input is in MPS input format, you can convert it to the sparse input format of PROC LP using the SAS macro function SASMPSXS. For example, if your have an MPS file called MODEL.MPS and it is stored in the directory C:\OR on a PC, the following program can help you to convert the file and solve the problem.

```
%sasmpsxs(mpsfile="c:\or\model.mps",lpdata=lp);

data;
   set lp;
   retain i 1;
   if _type_="FREE" and i=1 then
      do;
        _type_="MIN";
        i=0;
      end;
run;

proc lp sparsedata;
run;
```

In the MPS input format, all objective functions, price change rows, and free rows have the type 'N'. The SASMPSXS macro marks them as 'FREE' rows. After the conversion, you must run a DATA step to identify the objective rows and price change rows. In this example, assume that the problem is one of minimization and the first 'FREE' row is an objective row.

Syntax: LP Procedure

Below are statements used in PROC LP, listed in alphabetical order as they appear in the text that follows.

PROC LP *options* **;**
COEF *variables* **;**
COL *variable* **;**
ID *variable(s)* **;**
IPIVOT *;* **;**
PIVOT *;* **;**
PRINT *options* **;**
QUIT *options* **;**
RANGE *variable* **;**
RESET *options* **;**
RHS *variables* **;**
RHSSEN *variables* **;**
ROW *variable(s)* **;**
RUN *;* **;**
SHOW *options* **;**
TYPE *variable* **;**
VAR *variables* **;**

The TYPE, ID (or ROW), VAR, RHS, RHSSEN, and RANGE statements are used for identifying variables in the problem data set when the model is in the dense input format. In the dense input format, a model's variables appear as variables in the problem data set. The TYPE, ID (or ROW), and RHS statements can be omitted if the input data set contains variables _TYPE_, _ID_ (or _ROW_), and _RHS_; otherwise, they must be used. The VAR statement is optional. When it is omitted, PROC LP treats all numeric variables that are not explicitly or implicitly included in RHS, RHSSEN, and RANGE statements as structural variables. The RHSSEN and RANGE statements are optional statements for sensitivity and range analyses. They can be omitted if the input data set contains the _RHSSEN_ and _RANGE_ variables.

The TYPE, COL, ROW (or ID), COEF, RHS, RHSSEN, and RANGE statements are used for identifying variables in the problem data set when the model is in the sparse input format. In the sparse input format, a model's rows and columns appear as observations in the problem data set. The TYPE, COL, ROW (or ID), and COEF statements can be omitted if the input data set contains the _TYPE_ and _COL_ variables, as well as variables beginning with the prefixes _ROW (or _ID) and _COEF. Otherwise, they must be used. The RHS, RHSSEN, and RANGE statements identify the corresponding columns in the model. These statements can be omitted if there are observations that contain the RHS, RHSSEN, and RANGE types or the _RHS_, _RHSSEN_, and _RANGE_ column values.

The SHOW, RESET, PRINT, QUIT, PIVOT, IPIVOT, and RUN statements are especially useful when executing PROC LP interactively. However, they can also be used in batch mode.

Functional Summary

The statements and options available with PROC LP are summarized by purpose in the following table.

Table 5.1 Functional Summary

Description	Statement	Option
Interactive Statements:		
Perform one integer pivot and pause	IPIVOT	
Perform one simplex pivot and pause	PIVOT	
Display information at current iteration	PRINT	
Terminate processing immediately	QUIT	
Reset options specified	RESET	
Start or resume optimization	RUN	
Show settings of options	SHOW	
Variable Lists:		
Variables that contain coefficients (sparse)	COEF	
Variable that contains column names (sparse)	COL	
Alias for the ROW statement	ID	
Variable (column) that contains the range constant for the dense (sparse) format	RANGE	
Variables (columns) that contains RHS constants for the dense (sparse) format	RHS	
Variables (columns) that define RHS change vectors for the dense (sparse) format	RHSSEN	
Variable that contains names of constraints and objective functions (names of rows) for the dense (sparse) format	ROW	
Variable that contains the type of each observation	TYPE	
Structural variables (dense)	VAR	
Data Set Options:		
Active nodes input data set	PROC LP	ACTIVEIN=
Active nodes output data set	PROC LP	ACTIVEOUT=
Input data set	PROC LP	DATA=
Dual output data set	PROC LP	DUALOUT=
Primal input data set	PROC LP	PRIMALIN=
Primal output data set	PROC LP	PRIMALOUT=
Sparse format data input flag	PROC LP	SPARSEDATA
Tableau output data set	PROC LP	TABLEAUOUT=
Convert sparse or dense format input data set into MPS-format output data set	PROC LP	MPSOUT=

Description	Statement	Option
Display Control Options:		
Display iteration log	PROC LP	FLOW
Nonzero tolerance displaying	PROC LP	FUZZ=
Inverse of FLOW option	PROC LP	NOFLOW
Inverse of PARAPRINT option	PROC LP	NOPARAPRINT
Omit some displaying	PROC LP	NOPRINT
Inverse of TABLEAUPRINT	PROC LP	NOTABLEAUPRINT
Parametric programming displaying	PROC LP	PARAPRINT
Inverse of NOPRINT	PROC LP	PRINT
Iteration frequency of display	PROC LP	PRINTFREQ=
Level of display desired	PROC LP	PRINTLEVEL=
Display the final tableau	PROC LP	TABLEAUPRINT
Interactive Control Options:		
Pause before displaying the solution	PROC LP	ENDPAUSE
Pause after first feasible solution	PROC LP	FEASIBLEPAUSE
Pause frequency of integer solutions	PROC LP	IFEASIBLEPAUSE=
Pause frequency of integer iterations	PROC LP	IPAUSE=
Inverse of ENDPAUSE	PROC LP	NOENDPAUSE
Inverse of FEASIBLEPAUSE	PROC LP	NOFEASIBLEPAUSE
Pause frequency of iterations	PROC LP	PAUSE=
Pause if within specified proximity	PROC LP	PROXIMITYPAUSE=
Pause after data are read	PROC LP	READPAUSE
Preprocessing Control Options:		
Do not perform preprocessing	PROC LP	NOPREPROCESS
Preprocessing error tolerance	PROC LP	PEPSILON=
Limit preprocessing iterations	PROC LP	PMAXIT=
Perform preprocessing techniques	PROC LP	PREPROCESS
Branch-and-Bound (BB) Control Options:		
Perform automatic node selection technique	PROC LP	AUTO
Backtrack strategy to be used	PROC LP	BACKTRACK=
Branch on binary variables first	PROC LP	BINFST
Active node selection strategy	PROC LP	CANSELECT=
Comprehensive node selection control parameter	PROC LP	CONTROL=
Backtrack related technique	PROC LP	DELTAIT=
Measure for pruning BB tree	PROC LP	DOBJECTIVE=
Integer tolerance	PROC LP	IEPSILON=
Limit integer iterations	PROC LP	IMAXIT=
Measure for pruning BB tree	PROC LP	IOBJECTIVE=
Order of two branched nodes in adding to BB tree	PROC LP	LIFOTYPE=
Inverse of AUTO	PROC LP	NOAUTO
Inverse of BINFST	PROC LP	NOBINFST
Inverse of POSTPROCESS	PROC LP	NOPOSTPROCESS

Description	Statement	Option
Limit number of branching variables	PROC LP	PENALTYDEPTH=
Measure for pruning BB tree	PROC LP	POBJECTIVE=
Perform variables fixing technique	PROC LP	POSTPROCESS
Percentage used in updating WOBJECTIVE	PROC LP	PWOBJECTIVE=
Compression algorithm for storing active nodes	PROC LP	TREETYPE=
Branching variable selection strategy	PROC LP	VARSELECT=
Delay examination of some active nodes	PROC LP	WOBJECTIVE=
Sensitivity/Parametric/Ranging Control Options:		
Inverse of RANGEPRICE	PROC LP	NORANGEPRICE
Inverse of RANGERHS	PROC LP	NORANGERHS
Limit perturbation of the price vector	PROC LP	PRICEPHI=
Range analysis on the price coefficients	PROC LP	RANGEPRICE
Range analysis on the RHS vector	PROC LP	RANGERHS
Limit perturbation of the RHS vector	PROC LP	RHSPHI=
Simplex Algorithm Control Options:		
Use devex method	PROC LP	DEVEX
General error tolerance	PROC LP	EPSILON=
Perform goal programming	PROC LP	GOALPROGRAM
Largest number used in computation	PROC LP	INFINITY=
Reinversion frequency	PROC LP	INVFREQ=
Reinversion tolerance	PROC LP	INVTOL=
Simultaneously set MAXIT1, MAXIT2, MAXIT3 and IMAXIT values	PROC LP	MAXIT=
Limit phase 1 iterations	PROC LP	MAXIT1=
Limit phase 2 iterations	PROC LP	MAXIT2=
Limit phase 3 iterations	PROC LP	MAXIT3=
Inverse of devex	PROC LP	NODEVEX
Restore basis after parametric programming	PROC LP	PARARESTORE
Weight of the phase 2 objective function in phase 1	PROC LP	PHASEMIX=
Multiple pricing strategy	PROC LP	PRICETYPE=
Number of columns to subset in multiple pricing	PROC LP	PRICE=
Limit the number of iterations randomly selecting each entering variable during phase 1	PROC LP	RANDOMPRICEMULT=
Zero tolerance in ratio test	PROC LP	REPSILON=
Scaling type to be performed	PROC LP	SCALE=
Zero tolerance in LU decomposition	PROC LP	SMALL=
Time pause limit	PROC LP	TIME=
Control pivoting during LU decomposition	PROC LP	U=

RESET Statement Options:

The RESET statement supports the same options as the PROC LP statement except for the DATA=, PRIMALIN=, and ACTIVEIN= options, and supports the following additional options:

Description	Statement	Option
New variable lower bound during phase 3	RESET	LOWER=
New variable upper bound during phase 3	RESET	UPPER=
PRINT Statement Options:		
Display the best integer solution	PRINT	BEST
Display variable summary for specified columns	PRINT	COLUMN
Display variable summary and price sensitivity analysis for specified columns	PRINT	COLUMN / SENSITIVITY
Display variable summary for integer variables	PRINT	INTEGER
Display variable summary for nonzero integer variables	PRINT	INTEGER_NONZEROS
Display variable summary for integer variables with zero activity	PRINT	INTEGER_ZEROS
Display submatrix for specified rows and columns	PRINT	MATRIX
Display formatted submatrix for specified rows and columns	PRINT	MATRIX / PICTURE
Display variable summary for continuous variables	PRINT	NONINTEGER
Display variable summary for nonzero continuous variables	PRINT	NONINTEGER_NONZEROS
Display variable summary for variables with nonzero activity	PRINT	NONZEROS
Display price sensitivity analysis or price parametric programming	PRINT	PRICESEN
Display price range analysis	PRINT	RANGEPRICE
Display RHS range analysis	PRINT	RANGERHS
Display RHS sensitivity analysis or RHS parametric programming	PRINT	RHSSEN
Display constraint summary for specified rows	PRINT	ROW
Display constraint summary and RHS sensitivity analysis for specified rows	PRINT	ROW / SENSITIVITY
Display solution, variable, and constraint summaries	PRINT	SOLUTION
Display current tableau	PRINT	TABLEAU
Display variables with zero activity	PRINT	ZEROS
SHOW Statement Options:		
Display options applied	SHOW	OPTIONS
Display status of the current solution	SHOW	STATUS
QUIT Statement Option:		
Save the defined output data sets and then terminate PROC LP	QUIT	/ SAVE

PROC LP Statement

PROC LP *options* **;**

This statement invokes the procedure. The following options can appear in the PROC LP statement.

Data Set Options

ACTIVEIN=*SAS-data-set*
names the SAS data set containing the active nodes in a branch-and-bound tree that is to be used to restart an integer program.

ACTIVEOUT=*SAS-data-set*
names the SAS data set in which to save the current branch-and-bound tree of active nodes.

DATA=*SAS-data-set*
names the SAS data set containing the problem data. If the DATA= option is not specified, PROC LP uses the most recently created SAS data set.

DUALOUT=*SAS-data-set*
names the SAS data set that contains the current dual solution (shadow prices) on termination of PROC LP. This data set contains the current dual solution only if PROC LP terminates successfully.

MPSOUT=*SAS-data-set*
names the SAS data set that contains converted sparse or dense format input data in MPS format. Invoking this option directs the LP procedure to halt before attempting optimization. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

PRIMALIN=*SAS-data-set*
names the SAS data set that contains a feasible solution to the problem defined by the DATA= data set. The data set specified in the PRIMALIN= option should have the same format as a data set saved using the PRIMALOUT= option. Specifying the PRIMALIN= option is particularly useful for continuing iteration on a problem previously attempted. It is also useful for performing sensitivity analysis on a previously solved problem.

PRIMALOUT=*SAS-data-set*
names the SAS data set that contains the current primal solution when PROC LP terminates.

SPARSEDATA
tells PROC LP that the data are in the sparse input format. If this option is not specified, PROC LP assumes that the data are in the dense input format. See the section "Sparse Data Input Format" on page 201 for information about the sparse input format.

TABLEAUOUT=*SAS-data-set*
names the SAS data set in which to save the final tableau.

Display Control Options

FLOW
requests that a journal of pivot information (the Iteration Log) be displayed after every PRINTFREQ= iterations. This includes the names of the variables entering and leaving the basis, the reduced cost of the entering variable, and the current objective value.

FUZZ=*e*
displays all numbers within *e* of zero as zeros. The default value is 1.0E−10.

NOFLOW
is the inverse of the FLOW option.

NOPARAPRINT
is the inverse of the PARAPRINT option.

NOPRINT
suppresses the display of the Variable, Constraint, and Sensitivity Analysis summaries. This option is equivalent to the PRINTLEVEL=0 option.

NOTABLEAUPRINT
is the inverse of the TABLEAUPRINT option.

PARAPRINT
indicates that the solution be displayed at each pivot when performing parametric programming.

PRINT
is the inverse of the NOPRINT option.

PRINTFREQ=*m*
indicates that after every *m*th iteration, a line in the (Integer) Iteration Log be displayed. The default value is 1.

PRINTLEVEL=*i*
indicates the amount of displaying that the procedure should perform.

PRINTLEVEL=-2	only messages to the SAS log are displayed
PRINTLEVEL=-1	is equivalent to NOPRINT unless the problem is infeasible. If it is infeasible, the infeasible rows are displayed in the Constraint Summary along with the Infeasible Information Summary.
PRINTLEVEL=0	is identical to NOPRINT
PRINTLEVEL=1	all output is displayed

The default value is 1.

TABLEAUPRINT
indicates that the final tableau be displayed.

Interactive Control Options

ENDPAUSE
requests that PROC LP pause before displaying the solution. When this pause occurs, you can enter the RESET, SHOW, PRINT, RUN, and QUIT statements.

FEASIBLEPAUSE
requests that PROC LP pause after a feasible (not necessarily integer feasible) solution has been found. At a pause, you can enter the RESET, SHOW, PRINT, PIVOT, RUN, and QUIT statements.

IFEASIBLEPAUSE=*n*
requests that PROC LP pause after every *n* integer feasible solutions. At a pause, you can enter the RESET, SHOW, PRINT, IPIVOT, PIVOT, RUN, and QUIT statements. The default value is 99999999.

IPAUSE=*n*
requests that PROC LP pause after every *n* integer iterations. At a pause, you can enter RESET, SHOW, PRINT, IPIVOT, PIVOT, RUN, and QUIT statements. The default value is 99999999.

NOENDPAUSE
is the inverse of the ENDPAUSE option.

NOFEASIBLEPAUSE
is the inverse of the FEASIBLEPAUSE option.

PAUSE=*n*
requests that PROC LP pause after every *n* iterations. At a pause, you can enter the RESET, SHOW, PRINT, IPIVOT, PIVOT, RUN, and QUIT statements. The default value is 99999999.

PROXIMITYPAUSE=*r*
causes the procedure to pause if at least one integer feasible solution has been found and the objective value of the current best integer solution can be determined to be within *r* units of the optimal integer solution. This distance, called proximity, is also displayed on the Integer Iteration Log. Note that the proximity is calculated using the minimum (maximum if the problem is maximization) objective value among the nodes that remain to be explored in the branch-and-bound tree as a bound on the value of the optimal integer solution. Following the first PROXIMITYPAUSE= pause, in order to avoid a pause at every iteration thereafter, it is recommended that you reduce this measure through the use of a RESET statement. Otherwise, if any other option or statement that causes the procedure to pause is used while the PROXIMITYPAUSE= option is in effect, pause interferences may occur. When this pause occurs, you can enter the RESET, SHOW, PRINT, IPIVOT, PIVOT, RUN, and QUIT statements. The default value is 0.

READPAUSE
requests that PROC LP pause after the data have been read and the initial basis inverted. When this pause occurs, you can enter the RESET, SHOW, PRINT, IPIVOT, PIVOT, RUN, and QUIT statements.

Preprocessing Control Options

NOPREPROCESS
is the inverse of the PREPROCESS option.

PEPSILON=*e*

specifies a positive number close to zero. This value is an error tolerance in the preprocessing. If the value is too small, any marginal changes may cause the preprocessing to repeat itself. However, if the value is too large, it may alter the optimal solution or falsely claim that the problem is infeasible. The default value is 1.0E−8.

PMAXIT=*n*

performs at most *n* preprocessings. Preprocessing repeats itself if it improves some bounds or fixes some variables. However when a problem is large and dense, each preprocessing may take a significant amount of CPU time. This option limits the number of preprocessings PROC LP performs. It can also reduce the build-up of round-off errors. The default value is 100.

PREPROCESS

performs preprocessing techniques. See the section "Preprocessing" on page 209 for further discussion.

Branch-and-Bound Algorithm Control Options

AUTO, AUTO(*m,n***)**

automatically sets and adjusts the value of the CONTROL= option. Initially, it sets CONTROL=0.70, concentrating on finding an integer feasible solution or an upper bound. When an upper bound is found, it sets CONTROL=0.5, concentrating on efficiency and lower bound improvement. When the number of active problems exceeds *m*, it starts to gradually increase the value of the CONTROL= option to keep the size of active problems under control. When total active problems exceed *n*, CONTROL=1 will keep the active problems from growing further. You can alter the automatic process by resetting the value of the CONTROL= option interactively.

The default values of *m* and *n* are 20000 and 250000, respectively. You can change the two values according to your computer's space and memory capacities.

BACKTRACK=*rule*

specifies the rule used to choose the next active problem when backtracking is required. One of the following can be specified:

- BACKTRACK=*LIFO*
- BACKTRACK=*FIFO*
- BACKTRACK=*OBJ*
- BACKTRACK=*PROJECT*
- BACKTRACK=*PSEUDOC*
- BACKTRACK=*ERROR*

The default value is *OBJ*. See the section "Integer Programming" on page 210 for further discussion.

BINFST

requests that PROC LP branch on binary variables first when integer and binary variables are present. The reasoning behind this is that a subproblem will usually be fathomed or found integer feasible after less than 20% of its variables have been fixed. Considering binary variables first attempts to reduce the size of the branch-and-bound tree. It is a heuristic technique.

CANSELECT=*rule*
specifies the rule used to choose the next active problem when backtracking is not required or used. One of the following can be specified:

- CANSELECT=*LIFO*
- CANSELECT=*FIFO*
- CANSELECT=*OBJ*
- CANSELECT=*PROJECT*
- CANSELECT=*PSEUDOC*
- CANSELECT=*ERROR*

The default value is *LIFO*. See the section "Integer Programming" on page 210 for further discussion.

CONTROL=*r*
specifies a number between 0 and 1. This option combines CANSELECT= and other rules to choose the next active problem. It takes into consideration three factors: efficiency, improving lower bounds, and improving upper bounds. When *r* is close to 0, PROC LP concentrates on improving lower bounds (upper bounds for maximization). However, the efficiency per integer iteration is usually the worst. When *r* is close to 1, PROC LP concentrates on improving upper bounds (lower bounds for maximization). In addition, the growth of active problems will be controlled and stopped at $r = 1$. When its value is around 0.5, PROC LP will be in the most efficient state in terms of CPU time and integer number of iterations. The CONTROL= option will be automatically adjusted when the AUTO option is applied.

DELTAIT=*r*
is used to modify the exploration of the branch-and-bound tree. If more than *r* integer iterations have occurred since the last integer solution was found, then the procedure uses the backtrack strategy in choosing the next node to be explored. The default value is 3 times the number of integer variables.

DOBJECTIVE=*r*
specifies that PROC LP should discard active nodes that cannot lead to an integer solution with the objective at least as small (or as large for maximizations) as the objective of the relaxed problem plus (minus) *r*. The default value is $+\infty$.

IEPSILON=*e*
requests that PROC LP consider an integer variable as having an integer value if its value is within *e* units of an integer. The default value is 1.0E−7.

IMAXIT=*n*
performs at most *n* integer iterations. The default value is 100.

IOBJECTIVE=*r*
specifies that PROC LP should discard active nodes unless the node could lead to an integer solution with the objective smaller (or larger for maximizations) than *r*. The default value is $+\infty$ for minimization ($-\infty$ for maximization).

LIFOTYPE=*c*
specifies the order in which to add the two newly branched active nodes to the LIFO list.

LIFOTYPE=0	add the node with minimum penalty first
LIFOTYPE=1	add the node with maximum penalty first
LIFOTYPE=2	add the node resulting from adding $x_i \geq \lceil x^{opt}(k)_i \rceil$ first
LIFOTYPE=3	add the node resulting from adding $x_i \leq \lfloor x^{opt}(k)_i \rfloor$ first

The default value is 0.

NOAUTO
is the inverse of the AUTO option.

NOBINFST
is the inverse of the BINFST option.

NOPOSTPROCESS
is the inverse of the POSTPROCESS option.

PENALTYDEPTH=*m*
requests that PROC LP examine *m* variables as branching candidates when VARSELECT=PENALTY. If the PENALTYDEPTH= option is not specified when VARSELECT=PENALTY, then all of the variables are considered branching candidates. The default value is the number of integer variables. See the section "Integer Programming" on page 210 for further discussion.

POBJECTIVE=*r*
specifies that PROC LP should discard active nodes that cannot lead to an integer solution with objective at least as small as $o + |o| \times r$ (at least as large as $o - |o| \times r$ for maximizations) where o is the objective of the relaxed noninteger constrained problem. The default value is $+\infty$.

POSTPROCESS
attempts to fix binary variables globally based on the relationships among the reduced cost and objective value of the relaxed problem and the objective value of the current best integer feasible solution.

PWOBJECTIVE=*r*
specifies a percentage for use in the automatic update of the WOBJECTIVE= option. If the WOBJECTIVE= option is not specified in PROC LP, then when an integer feasible solution is found, the value of the option is updated to be $b + q \times r$ where b is the best bound on the value of the optimal integer solution and q is the current proximity. Note that for maximizations, $b - q \times r$ is used. The default value is 0.95.

TREETYPE=*i*
specifies a data compression algorithm.

TREETYPE=0	no data compression
TREETYPE=1	Huffman coding compression routines
TREETYPE=2	adaptive Huffman coding compression routines
TREETYPE=3	adaptive arithmetic coding compression routines

For IP or MIP problems, the basis and bounds information of each active node is saved to a utility file. When the number of active nodes increases, the size of the utility file becomes larger and larger. If PROC LP runs into a disk problem, like "disk full ..." or "writing failure ...", you can use this option to compress the utility file. For more information on the data compression routines, refer to Nelson (1992). The default value is 0.

VARSELECT=*rule*
specifies the rule used to choose the branching variable on an integer iteration.

- VARSELECT=*CLOSE*
- VARSELECT=*PRIOR*
- VARSELECT=*PSEUDOC*
- VARSELECT=*FAR*
- VARSELECT=*PRICE*
- VARSELECT=*PENALTY*

The default value is FAR. See the section "Integer Programming" on page 210 for further discussion.

WOBJECTIVE=*r*
specifies that PROC LP should delay examination of active nodes that cannot lead to an integer solution with objective at least as small (as large for maximizations) as *r*, until all other active nodes have been explored. The default value is $+\infty$ for minimization ($-\infty$ for maximization).

Sensitivity/Parametric/Ranging Control Options

NORANGEPRICE
is the inverse of the RANGEPRICE option.

NORANGERHS
is the inverse of the RANGERHS option.

PRICEPHI=Φ
specifies the limit for parametric programming when perturbing the price vector. See the section "Parametric Programming" on page 220 for further discussion. See Example 5.5 for an illustration of this option.

RANGEPRICE
indicates that range analysis is to be performed on the price coefficients. See the section "Range Analysis" on page 219 for further discussion.

RANGERHS
indicates that range analysis is to be performed on the right-hand-side vector. See the section "Range Analysis" on page 219 for further discussion.

RHSPHI=Φ
specifies the limit for parametric programming when perturbing the right-hand-side vector. See the section "Parametric Programming" on page 220 for further discussion.

Simplex Algorithm Control Options

DEVEX
indicates that the devex method of weighting the reduced costs be used in pricing (Harris 1975).

EPSILON=*e*
specifies a positive number close to zero. It is used in the following instances:

During phase 1, if the sum of the basic artificial variables is within *e* of zero, the current solution is considered feasible. If this sum is not exactly zero, then there are artificial variables within *e* of zero in the current solution. In this case, a note is displayed on the SAS log.

During phase 1, if all reduced costs are $\leq e$ for nonbasic variables at their lower bounds and $\geq e$ for nonbasic variables at their upper bounds and the sum of infeasibilities is greater than *e*, then the problem is considered infeasible. If the maximum reduced cost is within *e* of zero, a note is displayed on the SAS log.

During phase 2, if all reduced costs are $\leq e$ for nonbasic variables at their lower bounds and $\geq e$ for nonbasic variables at their upper bounds, then the current solution is considered optimal.

During phases 1, 2, and 3, the EPSILON= option is also used to test if the denominator is different from zero before performing the ratio test to determine which basic variable should leave the basis.

The default value is 1.0E−8.

GOALPROGRAM
specifies that multiple objectives in the input data set are to be treated as sequential objectives in a goal-programming model. The value of the right-hand-side variable in the objective row gives the priority of the objective. Lower numbers have higher priority.

INFINITY=*r*
specifies the largest number PROC LP uses in computation. The INFINITY= option is used to determine when a problem has an unbounded variable value. The default value is the largest double precision number. [1]

INVFREQ=*m*
reinverts the current basis matrix after *m* major and minor iterations. The default value is 100.

INVTOL=*r*
reinverts the current basis matrix if the largest element in absolute value in the decomposed basis matrix is greater than *r*. If after reinversion this condition still holds, then the value of the INVTOL= option is increased by a factor of 10 and a note indicating this modification is displayed on the SAS log. When *r* is frequently exceeded, this may be an indication of a numerically unstable problem. The default value is 1000.

MAXIT=*n*
simultaneously sets the values of the MAXIT1=, MAXIT2=, MAXIT3=, and IMAXIT= options.

MAXIT1=*n*
performs at most $n \geq 0$ phase 1 iterations. The default value is 100.

[1] This value is system dependent.

MAXIT2=*n*
performs at most $n \geq 0$ phase 2 iterations. If MAXIT2=0, then only phase 1 is entered so that on successful termination PROC LP will have found a feasible, but not necessarily optimal, solution. The default value is 100.

MAXIT3=*n*
performs at most $n \geq 0$ phase 3 iterations. All dual pivots are counted as phase 3 pivots. The default value is 99999999.

NODEVEX
is the inverse of the DEVEX option.

PARARESTORE
indicates that following a parametric programming analysis, PROC LP should restore the basis.

PHASEMIX=*r*
specifies a number between 0 and 1. When the number is positive, PROC LP tries to improve the objective function of phase 2 during phase 1. The PHASEMIX= option is a weight factor of the phase 2 objective function in phase 1. The default value is 0.

PRICE=*m*
specifies the number of columns to subset when multiple pricing is used in selecting the column to enter the basis (Greenberg 1978). The type of suboptimization used is determined by the PRICETYPE= option. See the section "Pricing" on page 208 for a description of this process.

PRICETYPE=*pricetype*
specifies the type of multiple pricing to be performed. If this option is specified and the PRICE= option is not specified, then PRICE= is assumed to be 10. Valid values for the PRICETYPE= option are

- PRICETYPE=*COMPLETE*
- PRICETYPE=*DYNAMIC*
- PRICETYPE=*NONE*
- PRICETYPE=*PARTIAL*

The default value is PARTIAL. See the section "Pricing" on page 208 for a description of this process.

RANDOMPRICEMULT=*r*
specifies a number between 0 and 1. This option sets a limit, in phase 1, on the number of iterations when PROC LP will randomly pick the entering variables. The limit equals *r* times the number of nonbasic variables, or the number of basic variables, whichever is smaller. The default value of the RANDOMPRICEMULT= option is 0.01.

REPSILON=*e*
specifies a positive number close to zero. The REPSILON= option is used in the ratio test to determine which basic variable is to leave the basis. The default value is 1.0E−10.

SCALE=*scale*
specifies the type of scaling to be used. Valid values for the SCALE= option are

- SCALE=*BOTH*
- SCALE=*COLUMN*
- SCALE=*NONE*
- SCALE=*ROW*

The default value is BOTH. See the section "Scaling" on page 209 for further discussion.

SMALL=*e*
specifies a positive number close to zero. Any element in a matrix with a value less than *e* is set to zero. The default value is machine dependent.

TIME=*t*
checks at each iteration to see if *t* seconds have elapsed since PROC LP began. If more than *t* seconds have elapsed, the procedure pauses and displays the current solution. The default value is 120 seconds.

U=*r*
enables PROC LP to control the choice of pivots during LU decomposition and updating the basis matrix. The variable *r* should take values between EPSILON and 1.0 because small values of *r* bias the algorithm toward maintaining sparsity at the expense of numerical stability and vice versa. The more sparse the decomposed basis is, the less time each iteration takes. The default value is 0.1.

COEF Statement

COEF *variables* **;**

For the sparse input format, the COEF statement specifies the numeric variables in the problem data set that contain the coefficients in the model. The value of the coefficient variable in a given observation is the value of the coefficient in the column and row specified in the COLUMN and ROW variables in that observation. For multiple ROW variables, the LP procedure maps the ROW variables to the COEF variables on the basis of their order in the COEF and ROW statements. There must be the same number of COEF variables as ROW variables. If the COEF statement is omitted, the procedure looks for the default variable names that have the prefix _COEF.

COL Statement

COL *variable* **;**

For the sparse input format, the COL statement specifies a character variable in the problem data set that contains the names of the columns in the model. Columns in the model are either structural variables, right-hand-side vectors, right-hand-side change vectors, or a range vector. The COL variable must be a character variable. If the COL statement is omitted, the LP procedure looks for the default variable name _COL_.

ID Statement

ID *variable(s)* **;**

For the dense input format, the ID statement specifies a character variable in the problem data set that contains a name for each constraint coefficients row, objective coefficients row, and variable definition row. If the ID statement is omitted, the LP procedure looks for the default variable name, _ID_. If this variable is not in the problem data set, the procedure assigns the default name _OBSxx_ to each row, where xx specifies the observation number in the problem data set.

For the sparse input format, the ID statement specifies the character variables in the problem data set that contain the names of the rows in the model. Rows in the model are one of the following types: constraints, objective functions, bounding rows, or variable describing rows. The ID variables must be character variables. There must be the same number of ID variables as variables specified in the COEF statement. If the ID statement is omitted, the LP procedure looks for the default variable names having the prefix _ID.

NOTE: The ID statement is an alias for the ROW statement.

IPIVOT Statement

IPIVOT ;

The IPIVOT statement causes the LP procedure to execute one integer branch-and-bound pivot and pause. If you use the IPIVOT statement while the PROXIMITYPAUSE= option is in effect, pause interferences may occur. To avoid such interferences, you must either reset the PROXIMITYPAUSE value or submit IPIVOT; RUN; instead of IPIVOT;.

PIVOT Statement

PIVOT ;

The PIVOT statement causes the LP procedure to execute one simplex pivot and pause.

PRINT Statement

PRINT *options* **;**

The PRINT statement is useful for displaying part of a solution summary, examining intermediate tableaus, performing sensitivity analysis, and using parametric programming. In the options, the colnames and rownames lists can be empty, in which case the LP procedure displays tables with all columns or rows, or both. If a column name or a row name has spaces or other special characters in it, the name must be enclosed in single or double quotes when it appears in the argument.

The options that can be used with this statement are as follows.

BEST
displays a Solution, Variable, and Constraint Summary for the best integer solution found.

COLUMN(*colnames*) / SENSITIVITY
displays a Variable Summary containing the logical and structural variables listed in the colnames list. If the / SENSITIVITY option is included, then sensitivity analysis is performed on the price coefficients for the listed colnames structural variables.

INTEGER
displays a Variable Summary containing only the integer variables.

INTEGER_NONZEROS
displays a Variable Summary containing only the integer variables with nonzero activity.

INTEGER_ZEROS
displays a Variable Summary containing only the integer variables with zero activity.

MATRIX(*rownames,colnames*) / PICTURE
displays the submatrix of the matrix of constraint coefficients defined by the rownames and colnames lists. If the / PICTURE option is included, then the formatted submatrix is displayed. The format used is summarized in Table 5.2.

Table 5.2 Format Summary

Condition on the Coefficient x							Symbols Printed	
			abs(x)	=	0			" "
0		<	abs(x)	<		.000001	sgn(x)	"Z"
	.000001	≤	abs(x)	<		.00001	sgn(x)	"Y"
	.00001	≤	abs(x)	<		.0001	sgn(x)	"X"
	.0001	≤	abs(x)	<		.001	sgn(x)	"W"
	.001	≤	abs(x)	<		.01	sgn(x)	"V"
	.01	≤	abs(x)	<		.1	sgn(x)	"U"
	.1	≤	abs(x)	<	1		sgn(x)	"T"
			abs(x)	=	1		sgn(x)	"1"
1		<	abs(x)	<	10		sgn(x)	"A"
10		≤	abs(x)	<	100		sgn(x)	"B"
100		≤	abs(x)	<	1000		sgn(x)	"C"
1000		≤	abs(x)	<	10000		sgn(x)	"D"
10000		≤	abs(x)	<	100000		sgn(x)	"E"
100000		≤	abs(x)	<	1.0E06		sgn(x)	"F"

NONINTEGER
displays a Variable Summary containing only the continuous variables.

NONINTEGER_NONZEROS
displays a Variable Summary containing only the continuous variables with nonzero activity.

NONZEROS
displays a Variable Summary containing only the variables with nonzero activity.

PRICESEN
displays the results of parametric programming for the current value of the PRICEPHI= option, the price coefficients, and all of the price change vectors.

RANGEPRICE
performs range analysis on the price coefficients.

RANGERHS
performs range analysis on the right-hand-side vector.

RHSSEN
displays the results of parametric programming for the current value of the RHSPHI= option, the right-hand-side coefficients, and all of the right-hand-side change vectors.

ROW(*rownames*) / SENSITIVITY
displays a constraint summary containing the rows listed in the rowname list. If the / SENSITIVITY option is included, then sensitivity analysis is performed on the right-hand-side coefficients for the listed rownames.

SOLUTION
displays the Solution Summary, including the Variable Summary and the Constraint Summary.

TABLEAU
displays the current tableau.

ZEROS
displays a Variable Summary containing only the variables with zero activity. This may be useful in the analysis of ON/OFF, ZERO/ONE, scheduling, and assignment applications.

QUIT Statement

QUIT *options* **;**

The QUIT statement causes the LP procedure to terminate processing immediately. No further displaying is performed and no output data sets are created.

The QUIT/SAVE statement causes the LP procedure to save the output data sets, defined in the PROC LP statement or in the RESET statement, and then terminate the procedure.

RANGE Statement

RANGE *variable* **;**

For the dense input format, the RANGE statement identifies the variable in the problem data set that contains the range coefficients. These coefficients enable you to specify the feasible range of a row. For example, if the *i*th row is

$$a^T x \leq b_i$$

and the range coefficient for this row is $r_i > 0$, then all values of x that satisfy

$$b_i - r_i \leq a^T x \leq b_i$$

are feasible for this row. Table 5.3 shows the bounds on a row as a function of the row type and the sign on a nonmissing range coefficient r.

Table 5.3 Interpretation of the Range Coefficient

		Bounds	
r	_TYPE_	Lower	Upper
$\neq 0$	LE	$b - \|r\|$	b
$\neq 0$	GE	b	$b + \|r\|$
> 0	EQ	b	$b + r$
< 0	EQ	$b + r$	b

If you include a range variable in the model and have a missing value or zero for it in a constraint row, then that constraint is treated as if no range variable had been included.

If the RANGE statement is omitted, the LP procedure assumes that the variable named _RANGE_ contains the range coefficients.

For the sparse input format, the RANGE statement gives the name of a column in the problem data set that contains the range constants. If the RANGE statement is omitted, then the LP procedure assumes that the column named _RANGE_ or the column with the 'RANGE' keyword in the problem data set contains the range constants.

RESET Statement

RESET *options* **;**

The RESET statement is used to change options after the LP procedure has started execution.

All of the options that can be set in the PROC LP statement can also be reset with the RESET statement, except for the DATA=, the PRIMALIN=, and the ACTIVEIN= options. In addition to the options available with the PROC LP statement, the following two options can be used.

LOWER(*colnames*)=*n*;

During phase 3, this sets the lower bound on all of the structural variables listed in the colnames list to an integer value *n*. This may contaminate the branch-and-bound tree. All nodes that descend from the current problem have lower bounds that may be different from those input in the problem data set.

UPPER(*colnames*)=*n*;

During phase 3, this sets the upper bound on all of the structural variables listed in the colnames list to an integer value *n*. This may contaminate the branch-and-bound tree. All nodes that descend from the current problem have upper bounds that may be different from those input in the problem data set.

Note that the LOWER= and UPPER= options only apply to phase 3 for integer problems. Therefore, they should only be applied once the integer iterations have started; if they are applied before then, they will be ignored.

RHS Statement

RHS *variables* **;**

For the dense input format, the RHS statement identifies variables in the problem data set that contain the right-hand-side constants of the linear program. Only numeric variables can be specified. If more than one variable is included in the RHS statement, the LP procedure assumes that problems for several linear programs are defined in the problem data set. A new linear program is defined for each variable in the RHS list. If the RHS statement is omitted, the procedure assumes that a variable named _RHS_ contains the right-hand-side constants.

For the sparse input format, the RHS statement gives the names of one or more columns in the problem data set that are to be considered as right-hand-side constants. If the RHS statement is omitted, then the LP procedure assumes that the column named _RHS_ or columns with the 'RHS' keyword in the problem data set contain the right-hand-side constants. See the section "Sparse Data Input Format" on page 201 for further discussion.

As default, the LP procedure assumes that the RHS constant is a zero vector for the dense and sparse input formats.

RHSSEN Statement

RHSSEN *variables* **;**

For the dense input format, the RHSSEN statement identifies variables in the problem data set that define change vectors for examining the sensitivity of the optimal solution to changes in the RHS constants. If the RHSSEN statement is omitted, then the LP procedure assumes that a variable named _RHSSEN_ contains a right-hand-side change vector.

For the sparse input format, the RHSSEN statement gives the names of one or more columns in the problem data set that are to be considered as change vectors. If the RHSSEN statement is omitted, then the LP procedure assumes that the column named _RHSSEN_ or columns with the 'RHSSEN' keyword in the problem data set contain the right-hand-side change vectors. For further information, see the section "Sparse Data Input Format" on page 201, the section "Right-Hand-Side Sensitivity Analysis" on page 217, and the section "Right-Hand-Side Parametric Programming" on page 220.

ROW Statement

ROW *variable(s)* **;**

For the dense input format, the ROW statement specifies a character variable in the problem data set that contains a name for each row of constraint coefficients, each row of objective coefficients and each variable describing row. If the ROW statement is omitted, the LP procedure looks for the default variable name, _ROW_. If there is no such variable in the problem data set, the procedure assigns the default name _OBSxx_ to each row, where xx specifies the observation number in the problem data set.

For the sparse input format, the ROW statement specifies the character variables in the problem data set that contain the names of the rows in the model. Rows in the model are one of the following types: constraints,

objective functions, bounding rows, or variable describing rows. The ROW variables must be character variables. There must be the same number of ROW variables as variables specified in the COEF statement. If the ROW statement is omitted, the LP procedure looks for the default variable names having the prefix _ROW.

RUN Statement

RUN ;

The RUN statement causes optimization to be started or resumed.

The TITLE or OPTIONS statement should not appear between PROC LP and RUN statements.

SHOW Statement

SHOW *options* **;**

The SHOW statement specifies that the LP procedure display either the *current options* or the *current solution status* on the SAS log.

OPTIONS
requests that the current options be displayed on the SAS log.

STATUS
requests that the status of the current solution be displayed on the SAS log.

TYPE Statement

TYPE *variable* **;**

The TYPE statement specifies a character variable in the problem data set that contains the type identifier for each observation. This variable has keyword values that specify how the LP procedure should interpret the observation. If the TYPE statement is omitted, the procedure assumes that a variable named _TYPE_ contains the type keywords.

For the dense input format, the type variable identifies the constraint and objective rows and rows that contain information about the variables. The type variable should have nonmissing values in all observations.

For the sparse input format, the type variable identifies a model's rows and columns. In an observation, a nonmissing type is associated with either a row or a column. If there are many columns sharing the same type, you can define a row of that type. Then, any nonmissing values in that row set the types of the corresponding columns.

The following are valid values for the TYPE variable in an observation:

MIN	contains the price coefficients of an objective row, for example, c in the problem (MIP), to be minimized.
MAX	contains the price coefficients of an objective row, for example, c, to be maximized.

EQ (=)	contains coefficients of an equality constrained row.
LE (≤)	contains coefficients of an inequality, less than or equal to, constrained row.
GE (≥)	contains coefficients of an inequality, greater than or equal to, constrained row.
SOSEQ	identifies the row as specifying a special ordered set. The variables flagged in this row are members of a set *exactly one* of which must be above its lower bound in the optimal solution. Note that variables in this type of special ordered set must be *integer*.
SOSLE	identifies the row as specifying a special ordered set. The variables flagged in this row are members of a set in which only one can be above its lower bound in the optimal solution.
UNRSTRT UNRSTRCT	identifies those structural variables to be considered as unrestricted variables. These are variables for which $\ell_i = -\infty$ and $u_i = +\infty$. Any variable that has a 1 in this observation is considered an unrestricted variable.
LOWERBD	identifies lower bounds on the structural variables. If all structural variables are to be nonnegative, that is, $\ell_i = 0$, then you do not need to include an observation with the 'LOWERBD' keyword in a variable specified in the TYPE statement. Missing values for variables in a lower-bound row indicate that the variable has lower bound equal to zero. **NOTE:** A variable with lower or upper bounds cannot be identified as unrestricted.
UPPERBD	identifies upper bounds u_i on the structural variables. For each structural variable that is to have an upper bound $u_i = +\infty$, the observation must contain a missing value or the current value of INFINITY. All other values are interpreted as upper bounds, including 0.
FIXED	identifies variables that have fixed values. A nonmissing value in a row with 'FIXED' type keyword gives the constant value of that variable.
INTEGER	identifies variables that are integer-constrained. In a feasible solution, these variables must have integer values. A missing value in a row with 'INTEGER' type keyword indicates that the variable is not integer-constrained. The value of variables in the 'INTEGER' row gives an ordering to the integer-constrained variables that is used when the VARSELECT= option equals PRIOR. **NOTE:** Every integer-constrained variable must have an upper bound defined in a row with type 'UPPERBD'. See the section "Controlling the Branch-and-Bound Search" on page 213 for further discussion.

BINARY	identifies variables that are constrained to be either 0 or 1. This is equivalent to specifying that the variable is an integer variable and has a lower bound of 0 and an upper bound of 1. A missing value in a row with 'BINARY' type keyword indicates that the variable is not constrained to be 0 or 1. The value of variables in the 'BINARY' row gives an ordering to the integer-constrained variables that is used when the VARSELECT= option equals PRIOR. See the section "Controlling the Branch-and-Bound Search" on page 213 for further discussion.
BASIC	identifies variables that form an initial basic feasible solution. A missing value in a row with 'BASIC' type indicates that the variable is not basic.
PRICESEN	identifies a vector that is used to evaluate the sensitivity of the optimal solution to changes in the objective function. See the section "Price Sensitivity Analysis" on page 218 and the section "Price Parametric Programming" on page 221 for further discussion.
FREE	identifies a nonbinding constraint. Any number of FREE constraints can appear in a problem data set.
RHS	identifies a right-hand-side column in the sparse input format. This replaces the RHS statement. It is useful when converting the MPS format into the sparse format of PROC LP. See the section "Converting Standard MPS Format to Sparse Format" on page 204 for more information.
RHSSEN	identifies a right-hand-side sensitivity analysis vector in the sparse input format. This replaces the RHSSEN statement. It is useful when converting the MPS format into the sparse format of PROC LP. See the section "Converting Standard MPS Format to Sparse Format" on page 204 for more information.
RANGE	identifies a range vector in the sparse input format. This replaces the RANGE statement. It is useful when converting the MPS format into the sparse format of PROC LP. See the section "Converting Standard MPS Format to Sparse Format" on page 204 for more information.

VAR Statement

VAR *variables* **;**

For the dense input format, the VAR statement identifies variables in the problem data set that are to be interpreted as structural variables in the linear program. Only numeric variables can be specified. If no VAR statement is specified, the LP procedure uses all numeric variables not included in an RHS or RHSSEN statement as structural variables.

Details: LP Procedure

Missing Values

The LP procedure treats missing values as missing in all rows except those that identify either upper or lower bounds on structural variables. If the row is an upper-bound row, then the type identifier is 'UPPERBD' and the LP procedure treats missing values as $+\infty$. If the row is a lower-bound row, then the type identifier is 'LOWERBD' and the LP procedure treats missing values as 0, except for the variables that are identified as 'UNRSTRT'.

Dense Data Input Format

In the dense format, a model is expressed in a similar way as it is formulated. Each SAS variable corresponds to a model's column and each SAS observation corresponds to a model's row. A SAS variable in the input data set is one of the following:

- a type variable
- an id variable
- a structural variable
- a right-hand-side variable
- a right-hand-side sensitivity analysis variable
- a range variable

The type variable tells PROC LP how to interpret the observation as a part of the mathematical programming problem. It identifies and classifies objectives, constraints, and the rows that contain information of variables like types, bounds, and so on. PROC LP recognizes the following keywords as values for the type variable: MIN, MAX, EQ, LE, GE, SOSEQ, SOSLE, UNRSTRT, LOWERBD, UPPERBD, FIXED, INTEGER, BINARY, BASIC, PRICESEN, and FREE. The values of the id variable are the names of the rows in the model. The other variables identify and classify the columns with numerical values.

The TYPE, ID (or ROW), and RHS statements can be omitted if the input data set contains variables _TYPE_, _ID_ (or _ROW_), and _RHS_; otherwise, they must be used. The VAR statement is optional. When it is not specified, PROC LP uses as structural variables all numeric variables not explicitly or implicitly included in statement lists. The RHSSEN and RANGE statements are optional statements for sensitivity and range analyses. They can be omitted if the input data set contains the _RHSSEN_ and _RANGE_ variables.

Sparse Data Input Format

The sparse format to PROC LP is designed to enable you to specify only the nonzero coefficients in the description of linear programs, integer programs, and mixed-integer programs. The SAS data set that describes the sparse model must contain at least four SAS variables:

- a type variable
- a column variable
- a row variable
- a coefficient variable

Each observation in the data set associates a type with a row or column, and defines a coefficient or numerical value in the model. The value of the type variable is a keyword that tells PROC LP how to interpret the observation. In addition to the keywords in the dense format, PROC LP also recognizes the keywords RHS, RHSSEN, and RANGE as values of the type variable. Table 5.5 shows the keywords that are recognized by PROC LP and in which variables can appear in the problem data set.

The values of the row and column variables are the names of the rows and columns in the model. The values of the coefficient variables define basic coefficients and lower and upper bounds, and identify model variables with types BASIC, FIXED, BINARY, and INTEGER. All character values in the sparse data input format are case insensitive.

The SAS data set can contain multiple pairs of rows and coefficient variables. In this way, more information about the model can be specified in each observation in the data set. See Example 5.2 for details.

Table 5.5 Variable Keywords Used in the Problem Data Set

TYPE (_TYPE_)	COL (_COL_)
MIN	
MAX	
EQ	
LE	
GE	
SOSEQ	
SOSLE	
UNRSTRT	
LOWERBD	
UPPERBD	
FIXED	
INTEGER	
BINARY	
BASIC	
PRICESEN	
FREE	
RHS	_RHS_
RHSSEN	_RHSSEN_
RANGE	_RANGE_
*xxxxxxx	

Follow these rules for sparse data input:

- The order of the observations is unimportant.

- Each unique column name appearing in the COL variable defines a unique column in the model.
- Each unique row name appearing in the ROW variable defines a unique row in the model.
- The type of the row is identified when an observation in which the row name appears (in a ROW variable) has type MIN, MAX, LE, GE, EQ, SOSLE, SOSEQ, LOWERBD, UPPERBD, UNRSTRT, FIXED, BINARY, INTEGER, BASIC, FREE, or PRICESEN.
- The type of each row must be identified at least once. If a row is given a type more than once, the multiple definitions must be identical.
- When there are multiple rows named in an observation (that is, when there are multiple ROW variables), the TYPE variable applies to each row named in the observation.
- The type of a column is identified when an observation in which the column name but no row name appears has the type LOWERBD, UPPERBD, UNRSTRT, FIXED, BINARY, INTEGER, BASIC, RHS, RHSSEN, or RANGE. A column type can also be identified in an observation in which both column and row names appear and the row name has one of the preceding types.
- Each column is assumed to be a structural column in the model unless the column is identified as a right-hand-side vector, a right-hand-side change vector, or a range vector. A column can be identified as one of these types using either the keywords RHS, RHSSEN, or RANGE in the TYPE variable, the special column names _RHS_, _RHSSEN_, or _RANGE_, or the RHS, RHSSEN, or RANGE statements following the PROC LP statement.
- A TYPE variable beginning with the character ∗ causes the observation to be interpreted as a comment.

When the column names appear in the Variable Summary in the PROC LP output, they are listed in alphabetical order. The row names appear in the order in which they appear in the problem data set.

Converting Any PROC LP Format to an MPS-Format SAS Data Set

The MPSOUT= option enables you to convert an input data set for the LP procedure into an MPS-format SAS data set. The converted data set is readable by the OPTLP and OPTMILP procedures.

The conversion can handle both linear and mixed integer linear programs. The _TYPE_ values for sensitivity analysis (PRICESEN), parametric programming (RHSSEN), and input basis (BASIS) are dropped. When multiple objective rows are present, only the first row is marked as the objective row. The remaining rows are marked as free rows. When multiple right-hand side (RHS) columns are present, only the first RHS column is processed. Constraints with a _TYPE_ value of SOSEQ or SOSLE are ignored. The MPSOUT= option does not output branching priorities specified for the VARSELECT=PRIOR option to a BRANCH section in the MPS-format SAS data set.

For information about how the contents of the MPS-format SAS data set are interpreted, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*). For examples that demonstrate the use of the MPSOUT= option and migration to the OPTMODEL procedure, see the section "Examples: LP Procedure" on page 232.

Converting Standard MPS Format to Sparse Format

The MPS input format was introduced by IBM as a way of specifying data for linear and integer programs. Before you can solve a linear program specified in the MPS input format by using the LP procedure, the data must be converted to the sparse format of the LP procedure. If you want to solve a linear program specified in the sparse LP format by using the OPTLP procedure, you must convert the data into an MPS-format SAS data set. This section describes how to perform both conversions.

SASMPSXS is a SAS macro function that converts the standard MPS format to the sparse format of the LP procedure. The following is an example of the MPS format:

```
NAME          EXAMPLE
* THIS IS DATA FOR THE PRODUCT MIX PROBLEM.
ROWS
 N  PROFIT
 L  STAMP
 L  ASSEMB
 L  FINISH
 N  CHNROW
 N  PRICE
COLUMNS
    DESK      STAMP          3.00000   ASSEMB        10.00000
    DESK      FINISH        10.00000   PROFIT        95.00000
    DESK      PRICE        175.00000
    CHAIR     STAMP          1.50000   ASSEMB         6.00000
    CHAIR     FINISH         8.00000   PROFIT        41.00000
    CHAIR     PRICE         95.00000
    CABINET   STAMP          2.00000   ASSEMB         8.00000
    CABINET   FINISH         8.00000   PROFIT        84.00000
    CABINET   PRICE        145.00000
    BOOKCSE   STAMP          2.00000   ASSEMB         7.00000
    BOOKCSE   FINISH         7.00000   PROFIT        76.00000
    BOOKCSE   PRICE        130.00000   CHNROW         1.00000
RHS
    TIME      STAMP        800.00000   ASSEMB       1200.0000
    TIME      FINISH       800.00000
RANGES
    T1        ASSEMB       900.00000
BOUNDS
    UP        CHAIR         75.00000
    LO        BOOKCSE       50.00000
ENDATA
```

In this example, the company tries to find an optimal product mix of four items: a DESK, a CHAIR, a CABINET, and a BOOKCASE. Each item is processed in a stamping department (STAMP), an assembly department (ASSEMB), and a finishing department (FINISH). The time each item requires in each department is given in the input data. Because of resource limitations, each department has an upper limit on the time available for processing. Furthermore, because of labor constraints, the assembly department must work at least 300 hours. Finally, marketing tells you not to make more than 75 chairs, to make at least 50 bookcases, and to find the range over which the selling price of a bookcase can vary without changing the optimal product mix.

The SASMPSXS macro function uses MPSFILE='FILENAME' as an argument to read an MPS input file. It then converts the file and saves the conversion to a default SAS data set, PROB. The FILENAME should include the path.

Running the following statements on the preceding example

```
%sasmpsxs(mpsfile='filename');

proc print data=prob;
run;
```

produces the sparse input form of the LP procedure:

OBS	_TYPE_	_COL_	_ROW1_	_COEF1_	_ROW2_	_COEF2_
1	*OW			.		.
2	FREE		PROFIT	.		.
3	LE		STAMP	.		.
4	LE		ASSEMB	.		.
5	LE		FINISH	.		.
6	FREE		CHNROW	.		.
7	FREE		PRICE	.		.
8	*OL	MNS		.		.
9		DESK	STAMP	3.0	ASSEMB	10
10		DESK	FINISH	10.0	PROFIT	95
11		DESK	PRICE	175.0		.
12		CHAIR	STAMP	1.5	ASSEMB	6
13		CHAIR	FINISH	8.0	PROFIT	41
14		CHAIR	PRICE	95.0		.
15		CABINET	STAMP	2.0	ASSEMB	8
16		CABINET	FINISH	8.0	PROFIT	84
17		CABINET	PRICE	145.0		.
18		BOOKCSE	STAMP	2	ASSEMB	7
19		BOOKCSE	FINISH	7	PROFIT	76
20		BOOKCSE	PRICE	130	CHNROW	1
21	*HS			.		.
22	RHS	TIME	STAMP	800	ASSEMB	1200
23	RHS	TIME	FINISH	800		.
24	*AN	ES		.		.
25	RANGE	T1	ASSEMB	900		.
26	*OU	DS		.		.
27	UPPERBDD	CHAIR	UP	75		.
28	LOWERBDD	BOOKCSE	LO	50		.

SASMPSXS recognizes four MPS row types: E, L, G, and N. It converts them into types EQ, LE, GE, and FREE. Since objective rows, price change rows and free rows all share the same type N in the MPS format, you need a DATA step to assign proper types to the objective rows and price change rows.

```
data;
   set prob;
   if _type_='free' and _row1_='profit' then _type_='max';
   if _type_='free' and _row1_='chnrow' then _type_='pricesen';
run;

proc lp sparsedata;
run;
```

In the MPS format, the variable types include LO, UP, FX, FR, MI, and BV. The SASMPSXS macro converts them into types LOWERBD, UPPERBD, FIXED, UNRESTRICTED, -INFINITY, and BINARY, respectively. Occasionally, you may need to define your own variable types, in which case, you must add corresponding type handling entries in the SASMPSXS.SAS program and use the SAS %INCLUDE macro to include the file at the beginning of your program. The SASMPSXS macro function can be found in the SAS sample library. Information on the MPS format can be obtained from Murtagh (1981).

SASMPSXS can take no arguments, or it can take one or two arguments. If no arguments are present, SASMPSXS assumes that the MPS input file has been saved to a SAS data set named RAW. The macro then takes information from that data set and converts it into the sparse form of the LP procedure. The RAW data set should have the following six variables:

```
data RAW;
   infile ...;
   input field1 $ 2-3   field2 $ 5-12
         field3 $ 15-22 field4   25-36
         field5 $ 40-47 field6   50-61;
   ...
run;
```

If the preceding MPS input data set has a name other than RAW, you can use MPSDATA=*SAS-data-set* as an argument in the SASMPSXS macro function. If you want the converted sparse form data set to have a name other than PROB, you can use LPDATA=*SAS-data-set* as an argument. The order of the arguments in the SASMPSXS macro function is not important.

The Reduced Costs, Dual Activities, and Current Tableau

The evaluation of reduced costs and the dual activities is independent of problem structure. For a basic solution, let B be the matrix composed of the basic columns of A and let N be the matrix composed of the nonbasic columns of A. The reduced cost associated with the ith variable is

$$(c^T - c_B^T B^{-1} A)_i$$

and the dual activity of the jth row is

$$(c_B^T B^{-1})_j$$

The Current Tableau is a section displayed when you specify either the TABLEAUPRINT option in the PROC LP statement or the TABLEAU option in the PRINT statement. The output contains a row for each basic variable and a column for each nonbasic variable. In addition, there is a row for the reduced costs and a column for the product

$$B^{-1}b$$

This column is labeled INV(B)*R. The body of the tableau contains the matrix

$$B^{-1}N$$

Macro Variable _ORLP_

The LP procedure defines a macro variable named _ORLP_. This variable contains a character string that indicates the status of the procedure. It is set whenever the user gets control, at breakpoints, and at procedure termination. The form of the _ORLP_ character string is STATUS= PHASE= OBJECTIVE= P_FEAS= D_FEAS= INT_ITER= INT_FEAS= ACTIVE= INT_BEST= PHASE1_ITER= PHASE2_ITER= PHASE3_ITER=. The terms are interpreted as follows:

STATUS=	the status of the current solution
PHASE=	the phase the procedure is in (1, 2, or 3)
OBJECTIVE=	the current objective value
P_FEAS=	whether the current solution is primal feasible
D_FEAS=	whether the current solution is dual feasible
INT_ITER=	the number of integer iterations performed
INT_FEAS=	the number of integer feasible solutions found
ACTIVE=	the number of active nodes in the current branch-and-bound tree
INT_BEST=	the best integer objective value found
PHASE1_ITER=	the number of iterations performed in phase 1
PHASE2_ITER=	the number of iterations performed in phase 2
PHASE3_ITER=	the number of iterations performed in phase 3

Table 5.7 shows the possible values for the nonnumeric terms in the string.

Table 5.7 Possible Values for Nonnumeric Terms

STATUS=	P_FEAS=	D_FEAS=
SUCCESSFUL	YES	YES
UNBOUNDED	NO	NO
INFEASIBLE		
MAX_TIME		
MAX_ITER		
PIVOT		
BREAK		
INT_FEASIBLE		
INT_INFEASIBLE		
INT_MAX_ITER		
PAUSE		
FEASIBLEPAUSE		
IPAUSE		
PROXIMITYPAUSE		
ACTIVE		
RELAXED		
FATHOMED		
IPIVOT		
UNSTABLE		
SINGULAR		
MEMORY_ERROR		
IO_ERROR		
SYNTAX_ERROR		
SEMANTIC_ERROR		
BADDATA_ERROR		
UNKNOWN_ERROR		

This information can be used when PROC LP is one step in a larger program that needs to identify how the LP procedure terminated. Because _ORLP_ is a standard SAS macro variable, it can be used in the ways that all macro variables can be used (see the *SAS Guide to Macro Processing*).

Pricing

PROC LP performs multiple pricing when determining which variable will enter the basis at each pivot (Greenberg 1978). This heuristic can shorten execution time in many problems. The specifics of the multiple pricing algorithm depend on the value of the PRICETYPE= option. However, in general, when some form of multiple pricing is used, during the first iteration PROC LP places the PRICE= nonbasic columns yielding the greatest marginal improvement to the objective function in a candidate list. This list identifies a subproblem of the original. On subsequent iterations, only the reduced costs for the nonbasic variables in the candidate list are calculated. This accounts for the potential time savings. When either the candidate list is empty or the subproblem is optimal, a new candidate list must be identified and the process repeats. Because identification

of the subproblem requires pricing the complete problem, an iteration in which this occurs is called a *major iteration*. A *minor iteration* is an iteration in which only the subproblem is to be priced.

The value of the PRICETYPE= option determines the type of multiple pricing that is to be used. The types of multiple pricing include partial suboptimization (PRICETYPE=PARTIAL), complete suboptimization (PRICETYPE=COMPLETE), and complete suboptimization with dynamically varying the value of the PRICE= option (PRICETYPE=DYNAMIC).

When partial suboptimization is used, in each minor iteration the nonbasic column in the subproblem yielding the greatest marginal improvement to the objective is brought into the basis and removed from the candidate list. The candidate list now has one less entry. At each subsequent iteration, another column from the subproblem is brought into the basis and removed from the candidate list. When there are either no remaining candidates or the remaining candidates do not improve the objective, the subproblem is abandoned and a major iteration is performed. If the objective cannot be improved on a major iteration, the current solution is optimal and PROC LP terminates.

Complete suboptimization is identical to partial suboptimization with one exception. When a nonbasic column from the subproblem is brought into the basis, it is replaced in the candidate list by the basic column that is leaving the basis. As a result, the candidate list does not diminish at each iteration.

When PRICETYPE=DYNAMIC, complete suboptimization is performed, but the value of the PRICE= option changes so that the ratio of minor to major iterations is within two units of the PRICE= option.

These heuristics can shorten execution time for small values of the PRICE= option. Care should be exercised in choosing a value from the PRICE= option because too large a value can use more time than if pricing were not used.

Scaling

Based on the SCALE= option specified, the procedure scales the coefficients of both the constraints and objective rows before iterating. This technique can improve the numerical stability of an ill-conditioned problem. If you want to modify the default matrix scaling used, which is SCALE=BOTH, use the SCALE=COLUMN, SCALE=ROW, or SCALE=NONE option in the PROC LP statement. If SCALE=BOTH, the matrix coefficients are scaled so that the largest element in absolute value in each row or column equals 1. They are scaled by columns first and then by rows. If SCALE=COLUMN (ROW), the matrix coefficients are scaled so that the largest element in absolute value in each column (row) equals 1. If SCALE=NONE, no scaling is performed.

Preprocessing

With the preprocessing option, you can identify redundant and infeasible constraints, improve lower and upper bounds of variables, fix variable values and improve coefficients and RHS values before solving a problem. Preprocessing can be applied to LP, IP and MIP problems. For an LP problem, it may significantly reduce the problem size. For an IP or MIP problem, it can often reduce the gap between the optimal solution and the solution of the relaxed problem, which could lead to a smaller search tree in the branch-and-bound algorithm. As a result, the CPU time may be reduced on many problems. Although there is no guarantee that preprocessing will always yield a faster solution, it does provide a highly effective approach to solving large and difficult problems.

Preprocessing is especially useful when the original problem causes numerical difficulties to PROC LP. Since preprocessing could identify redundant constraints and tighten lower and upper bounds of variables, the reformulated problem may eliminate the numerical difficulties in practice.

When a constraint is identified as redundant, its type is marked as 'FREE' in the Constraint Summary. If a variable is fixed, its type is marked as 'FIXED' in the Variables Summary. If a constraint is identified as infeasible, PROC LP stops immediately and displays the constraint name in the SAS log file. This capability sometimes gives valuable insight into the model or the formulation and helps establish if the model is reasonable and the formulation is correct.

For a large and dense problem, preprocessing may take a longer time for each iteration. To limit the number of preprocessings, use the PMAXIT= option. To stop any further preprocessings during the preprocessing stage, press the CTRL-BREAK key. PROC LP will enter phase 1 at the end of the current iteration.

Integer Programming

Formulations of mathematical programs often require that some of the decision variables take only integer values. Consider the formulation

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \{\geq, =, \leq\} b \\ & \ell \leq x \leq u \\ & x_i \text{ is integer}, i \in \mathcal{S} \end{array}$$

The set of indices $\mathcal{S}$ identifies those variables that must take only integer values. When $\mathcal{S}$ does not contain all of the integers between 1 and n, inclusive, this problem is called a mixed-integer program (MIP). Otherwise, it is known as an integer program. Let x^{opt}(MIP) denote an optimal solution to (MIP). An integer variable with bounds between 0 and 1 is also called a binary variable.

Specifying the Problem

An integer or mixed-integer problem can be solved with PROC LP. To solve this problem, you must identify the integer variables. You can do this with a row in the input data set that has the keyword 'INTEGER' for the type variable. Any variable that has a nonmissing and nonzero value for this row is interpreted as an integer variable. It is important to note that integer variables must have upper bounds explicitly defined using the 'UPPERBD' keyword. The values in the 'INTEGER' row not only identify those variables that must be integers, but they also give an ordering to the integer variables that can be used in the solution technique.

You can follow the same steps to identify binary variables. For the binary variables, there is no need to supply any upper bounds.

Following the rules of sparse data input format, you can also identify individual integer or binary variables.

The Branch-and-Bound Technique

The branch-and-bound approach is used to solve integer and mixed-integer problems. The following discussion outlines the approach and explains how to use several options to control the procedure.

The branch-and-bound technique solves an integer program by solving a sequence of linear programs. The sequence can be represented by a tree, with each node in the tree being identified with a linear program that is derived from the problems on the path leading to the root of the tree. The root of the tree is identified with a linear program that is identical to (MIP), except that $\mathcal{S}$ is empty. This relaxed version of (MIP), called (LP(0)), can be written as

$$\begin{array}{ll} x^{opt}(0) = & \min c^T x \\ \text{subject to} & Ax \{\geq, =, \leq\} b \\ & \ell \leq x \leq u \end{array}$$

The branch-and-bound approach generates linear programs along the nodes of the tree using the following scheme. Consider $x^{opt}(0)$, the optimal solution to (LP(0)). If $x^{opt}(0)_i$ is integer for all $i \in \mathcal{S}$, then $x^{opt}(0)$ is optimal in (MIP). Suppose for some $i \in \mathcal{S}$, $x^{opt}(0)_i$ is nonintegral. In that case, define two new problems (LP(1)) and (LP(2)), descendants of the parent problem (LP(0)). The problem (LP(1)) is identical to (LP(0)) except for the additional constraint

$$x_i \leq \lfloor x^{opt}(0)_i \rfloor$$

and the problem (LP(2)) is identical to (LP(0)) except for the additional constraint

$$x_i \geq \lceil x^{opt}(0)_i \rceil$$

The notation $\lceil y \rceil$ means the smallest integer greater than or equal to y, and the notation $\lfloor y \rfloor$ means the largest integer less than or equal to y. Note that the two new problems do not have $x^{opt}(0)$ as a feasible solution, but because the solution to (MIP) must satisfy one of the preceding constraints, x_i^{opt}(MIP) must satisfy one of the new constraints. The two problems thus defined are called active nodes in the branch-and-bound tree, and the variable x_i is called the branching variable.

Next, the algorithm chooses one of the problems associated with an active node and attempts to solve it using the dual simplex algorithm. The problem may be infeasible, in which case the problem is dropped. If it can be solved, and it in turn does not have an integer solution (that is, a solution for which x_i is integer for all $i \in \mathcal{S}$), then it defines two new problems. These new problems each contain all of the constraints of the parent problems plus the appropriate additional one.

Branching continues in this manner until either there are no active nodes or an integer solution is found. When an integer solution is found, its objective value provides a bound for the objective of (MIP). In particular, if z is the objective value of the current best integer solution, then any active problems whose parent problem has objective value $\geq z$ can be discarded (assuming that the problem is a minimization). This can be done because all problems that descend from this parent will also have objective value $\geq z$. This technique is known as *fathoming*. When there are no active nodes remaining to be solved, the current integer solution is optimal in (MIP). If no integer solution has been found, then (MIP) is (integer) infeasible.

It is important to realize that integer programs are NP-complete. Roughly speaking, this means that the effort required to solve them grows exponentially with the size of the problem. For example, a problem with 10 binary variables can, in the worst case, generate $2^{10} = 1024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can, in the worst case, generate $2^{20} = 1048576$ nodes in the branch-and-bound tree. Although the algorithm is unlikely to have to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource intensive.

The Integer Iteration Log

To help monitor the growth of the branch-and-bound tree, the LP procedure reports on the status of each problem that is solved. The report, displayed in the Integer Iteration Log, can be used to reconstruct the branch-and-bound tree. Each row in the report describes the results of the attempted solution of the linear program at a node in the tree. In the following discussion, a problem on a given line in the log is called the current problem. The following columns are displayed in the report:

Iter	identifies the number of the branch-and-bound iteration.
Problem	identifies how the current problem fits in the branch-and-bound tree.
Condition	reports the result of the attempted solution of the current problem. Values for Condition are: • ACTIVE: The current problem was solved successfully. • INFEASIBLE: The current problem is infeasible. • FATHOMED: The current problem cannot lead to an improved integer solution and therefore it is dropped. • SINGULAR: A singular basis was encountered in attempting to solve the current problem. Solution of this relaxed problem is suspended and will be attempted later if necessary. • SUBOPTIMAL: The current problem has an integer feasible solution.
Objective	reports the objective value of the current problem.
Branched	names the variable that is branched in subtrees defined by the descendants of this problem.
Value	gives the current value of the variable named in the column labeled Branched.
Sinfeas	gives the sum of the integer infeasibilities in the optimal solution to the current problem.
Active	reports the total number of nodes currently active in the branch-and-bound tree.
Proximity	reports the gap between the best integer solution and the current lower (upper for maximizations) bound of all active nodes.

To reconstruct the branch-and-bound tree from this report, consider the interpretation of iteration j. If Iter=j and Problem=k, then the problem solved on iteration j is identical to the problem solved on iteration $|k|$ with an additional constraint. If $k > 0$, then the constraint is an upper bound on the variable named in the Branched column on iteration $|k|$. If $k < 0$, then the constraint is a lower bound on that variable. The value

of the bound can be obtained from the value of Value in iteration $|\,k\,|$ as described in the previous section.

Example 5.8 in the section "Examples: LP Procedure" on page 232 shows an Integer Iteration Log in its output.

Controlling the Branch-and-Bound Search

There are several options you can use to control branching. This is accomplished by controlling the program's choice of the branching variable and of the next active node. In the discussion that follows, let

$$f_i(k) = x^{opt}(k)_i - \lfloor x^{opt}(k)_i \rfloor$$

where $x^{opt}(k)$ is the optimal solution to the problem solved in iteration k.

The CANSELECT= option directs the choice of the next active node. Valid keywords for this option include LIFO, FIFO, OBJ, PROJECT, PSEUDOC, and ERROR. The following list describes the action that each of these causes when the procedure must choose for solution a problem from the list of active nodes.

LIFO
: chooses the last problem added to the tree of active nodes. This search has the effect of a depth-first search of the branch-and-bound tree.

FIFO
: chooses the first node added to the tree of active nodes. This search has the effect of a breadth-first search of the branch-and-bound tree.

OBJ
: chooses the problem whose parent has the smallest (largest if the problem is a maximization) objective value.

PROJECT
: chooses the problem with the largest (smallest if the problem is a maximization) projected objective value. The projected objective value is evaluated using the sum of integer infeasibilities, $s(k)$, associated with an active node (LP(k)), defined by

$$s(k) = \sum_{i \in \mathcal{S}} \min\{f_i(k), 1 - f_i(k)\}$$

An empirical measure of the rate of increase (decrease) in the objective value is defined as

$$\lambda = (z^* - z(0))/s(0)$$

where

- $z(k)$ is the optimal objective value for (LP(k))
- z^* is the objective value of the current best integer solution

The projected objective value for problems (LP($k+1$)) and (LP($k+2$)) is defined as

$$z(k) + \lambda s(k)$$

PSEUDOC chooses the problem with the largest (least if the problem is a maximization) projected pseudocost) The projected pseudocost is evaluated using the weighted sum of infeasibilities $s_w(k)$ associated with an active problem (LP(k)), defined by

$$s_w(k) = \sum_{i \in \mathcal{S}} \min\{d_i(k) f_i(k), u_i(k)(1 - f_i(k))\}$$

The weights u_i and d_i are initially equal to the absolute value of the ith objective coefficient and are updated at each integer iteration. They are modified by examining the empirical marginal change in the objective as additional constraints are placed on the variables in $\mathcal{S}$ along the path from (LP(0)) to a node associated with an integer feasible solution. In particular, if the definition of problems (LP($k+1$)) and (LP($k+2$)) from parent (LP(k)) involve the addition of constraints $x_i \leq \lfloor x^{opt}(k)_i \rfloor$ and $x_i \geq \lceil x^{opt}(k)_i \rceil$, respectively, and one of them is on a path to an integer feasible solution, then only one of the following is true:

$$d_i(k) = (z(k+1) - z(k))/f_i(k)$$

$$u_i(k) = (z(k+2) - z(k))/(1 - f_i(k))$$

Note the similarity between $s_w(k)$ and $s(k)$. The weighted quantity $s_w(k)$ accounts to some extent for the influence of the objective function. The projected pseudocost for problems (LP($k+1$)) and (LP($k+2$)) is defined as

$$z_w(k) \equiv z(k) + s_w(k)$$

ERROR chooses the problem with the largest error. The error associated with problems (LP($k+1$)) and (LP($k+2$)) is defined as

$$(z^* - z_w(k))/(z^* - z(k))$$

The BACKTRACK= option controls the search for the next problem. This option can take the same values as the CANSELECT= option. In addition to the case outlined under the DELTAIT= option, backtracking is required as follows based on the CANSELECT= option in effect:

- If CANSELECT=LIFO and there is no active node in the portion of the active tree currently under exploration with a bound better than the value of WOBJECTIVE=, then the procedure must backtrack.
- If CANSELECT=FIFO, PROJECT, PSEUDOC, or ERROR and the bound corresponding to the node under consideration is not better than the value of WOBJECTIVE=, then the procedure must backtrack.

The default value is OBJ.

The VARSELECT= option directs the choice of branching variable. Valid keywords for this option include CLOSE, FAR, PRIOR, PSEUDOC, PRICE, and PENALTY. The following list describes the action that each of these causes when $x^{opt}(k)$, an optimal solution of problem (LP(k)), is used to define active problems (LP($k+1$)) and (LP($k+2$)).

CLOSE chooses as branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and } \text{IEPSILON} \leq f_i(k) \leq 1 - \text{IEPSILON}\}$$

FAR chooses as branching variable the variable x_i such that i maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and } \text{IEPSILON} \leq f_i(k) \leq 1 - \text{IEPSILON}\}$$

PRIOR chooses as branching variable x_i such that $i \in \mathcal{S}$, $x^{opt}(k)_i$ is nonintegral, and variable x_i has the minimum value in the INTEGER row in the input data set. This choice for the VARSELECT= option is recommended when you have enough insight into the model to identify those integer variables that have the most significant effect on the objective value.

PENALTY chooses as branching variable x_i such that $i \in \mathcal{S}$ and a bound on the increase in the objective of (LP(k)) (penalty) resulting from adding the constraint

$$x_i \leq \lfloor x^{opt}(k)_i \rfloor \quad \text{or} \quad x_i \geq \lceil x^{opt}(k)_i \rceil$$

is maximized. The bound is calculated without pivoting using techniques of sensitivity analysis (Garfinkel and Nemhauser 1972). Because the cost of calculating the maximum penalty can be large if $\mathcal{S}$ is large, you may want to limit the number of variables in $\mathcal{S}$ for which the penalty is calculated. The penalty is calculated for PENALTYDEPTH= variables in $\mathcal{S}$.

PRICE chooses as branching variable x_i such that $i \in \mathcal{S}$, $x^{opt}(k)_i$ is nonintegral, and variable x_i has the minimum price coefficient (maximum for maximization).

PSEUDOC chooses as branching variable the variable x_i such that i maximizes

$$\{\min\{d_i f_i(k), u_i(1 - f_i(k))\} \mid i \in \mathcal{S} \text{ and } \text{IEPSILON} \leq f_i(k) \leq 1 - \text{IEPSILON}\}$$

The weights u_i and d_i are initially equal to the absolute value of the ith objective coefficient and are updated whenever an integer feasible solution is encountered. See the discussion on the CANSELECT= option for details on the method of updating the weights.

Customizing Search Heuristics

Often a good heuristic for searching the branch-and-bound tree of a problem can be found. You are tempted to continue using this heuristic when the problem data changes but the problem structure remains constant. The ability to reset procedure options interactively enables you to experiment with search techniques in an attempt to identify approaches that perform well. Then you can easily reapply these techniques to subsequent problems.

For example, the PIP branch-and-bound strategy (Crowder, Johnson, and Padberg 1983) describes one such heuristic. The following program uses a similar strategy. Here, the OBJ rule (choose the active node with least parent objective function in the case of a minimization problem) is used for selecting the next active node to be solved until an integer feasible solution is found. Once such a solution is found, the search procedure is changed to the LIFO rule: choose the problem most recently placed in the list of active nodes.

```
proc lp canselect=obj ifeasiblepause=1;
run;
   reset canselect=lifo ifeasiblepause=9999999;
run;
```

Further Discussion on AUTO and CONTROL= options

Consider a minimization problem. At each integer iteration, PROC LP will select a node to solve from a pool of active nodes. The best bound strategy (CANSELECT=OBJ) will pick the node with the smallest projected objective value. This strategy improves the lower bound of the integer program and usually takes fewer integer iterations. One disadvantage is that PROC LP must recalculate the inverse of the basis matrix at almost every integer iteration; such recalculation is relatively expensive. Another disadvantage is that this strategy does not pay attention to improving the upper bound of the integer program. Thus the number of active nodes tends to grow rapidly if PROC LP cannot quickly find an optimal integer solution.

On the other hand, the LIFO strategy is very efficient and does not need to calculate the inverse of the basis matrix unless the previous node is fathomed. It is a depth-first strategy so it tends to find an integer feasible solution quickly. However, this strategy will pick nodes locally and usually will take longer integer iterations than the best bound strategy.

There is another strategy that is often overlooked. Here it is called the **best upper bound** strategy. With this strategy, each time you select an active node, instead of picking the node with the smallest projected objective value, you select the one with the largest projected objective value. This strategy is as efficient as the LIFO strategy. Moreover, it selects active nodes globally. This strategy tries to improve the upper bound of the integer program by searching for new integer feasible solutions. It also fathoms active nodes quickly and keeps the total number of active nodes below the current level. A disadvantage is that this strategy may evaluate more nodes that do not have any potential in finding an optimal integer solution.

The best bound strategy has the advantage of improving the lower bound. The LIFO strategy has the advantages of efficiency and finding a local integer feasible solution. The best upper bound strategy has the advantages of keeping the size of active nodes under control and at the same time trying to identify any potential integer feasible solution globally.

Although the best bound strategy is generally preferred, in some instances other strategies may be more effective. For example, if you have found an integer optimal solution but you do not know it, you still have to enumerate all possible active nodes. Then the three strategies will basically take the same number of integer iterations after an optimal solution is found but not yet identified. Since the LIFO and best upper bound strategies are very efficient per integer iteration, both will outperform the best bound strategy.

Since no one strategy suits all situations, a hybrid strategy has been developed to increase applicability. The CONTROL= option combines the above three strategies naturally and provides a simple control parameter in [0, 1] dealing with different integer programming problems and different solution situations. The AUTO option automatically sets and adjusts the CONTROL= parameter so that you do not need to know any problem structure or decide a node selection strategy in advance.

Since the LIFO strategy is less costly, you should use it as much as possible in the combinations. The following process is called a **diving process**. Starting from an active node, apply the LIFO strategy as much as you can until the current node becomes feasible or is fathomed, or exceeds a preset limit. During this process, there is no inverse matrix calculation involved except for the first node. When the diving process is over, apply one of the three strategies to select the next starting node. One set of combinations is called a cycle.

The control parameter *r* controls the frequency of the three strategies being applied and the depth of the diving process in a cycle. It starts with a pure best bound strategy at *r*=0, and then gradually increases the frequency of the diving processes and their depths as *r* increases. At *r*=0.5, one cycle contains a best bound strategy plus a full diving process. After *r*=0.5, the number of the diving processes will gradually increase in a cycle. In addition, the best upper bound strategy will join the cycle. As *r* continues to increase, the frequency of the best upper bound strategy will increase. At *r*=1, it becomes a pure best upper bound strategy.

The AUTO option will automatically adjust the value of the CONTROL= option. At the start, it sets CONTROL=0.7, which emphasizes finding an upper bound. After an integer feasible solution is found, it sets CONTROL=0.5, which emphasizes efficiency and lower bound improvement. When the number of active nodes grows over the default or user defined limit *m*, the number indicates that a better upper bound is needed. The AUTO option will start to increase the value of CONTROL= from 0.5. If the size of the active nodes continues to grow, so will the value of the CONTROL= option. When the size of active nodes reaches to the default or user-defined limit *n*, CONTROL= will be set to 1. At this moment, the growth of active nodes is stopped. When the size of active nodes reduces, AUTO will decrease the value of CONTROL= option.

You can use other strategies to improve the lower bound by setting CANSELECT= to other options.

Saving and Restoring the List of Active Nodes

The list of active nodes can be saved in a SAS data set for use at a subsequent invocation of PROC LP. The ACTIVEOUT= option in the PROC LP statement names the data set into which the current list of active nodes is saved when the procedure terminates due to an error termination condition. Examples of such conditions are time limit exceeded, integer iterations exceeded, and phase 3 iterations exceeded. The ACTIVEIN= option in the PROC LP statement names a data set that can be used to initialize the list of active nodes. To achieve the greatest benefit when restarting PROC LP, use the PRIMALOUT= and PRIMALIN= options in conjunction with the ACTIVEOUT= and ACTIVEIN= options. See Example 5.10 in the section "Examples: LP Procedure" on page 232 for an illustration.

Sensitivity Analysis

Sensitivity analysis is a technique for examining the effects of changes in model parameters on the optimal solution. The analysis enables you to examine the size of a perturbation to the right-hand-side or objective vector by an arbitrary change vector for which the basis of the current optimal solution remains optimal.

NOTE: When sensitivity analysis is performed on integer-constrained problems, the integer variables are fixed at the value they obtained in the integer optimal solution. Therefore, care must be used when interpreting the results of such analyses. Care must also be taken when preprocessing is enabled, because preprocessing usually alters the original formulation.

Right-Hand-Side Sensitivity Analysis

Consider the problem $(lpr(\phi))$:

$$\begin{array}{ll} x^{opt}(\phi) = & \min c^T x \\ \text{subject to} & Ax \{\geq, =, \leq\} b + \phi r \\ & \ell \leq x \leq u \end{array}$$

where *r* is a right-hand-side change vector.

Let $x^{opt}(\phi)$ denote an optimal basic feasible solution to $(lpr(\phi))$. PROC LP can be used to examine the effects of changes in ϕ on the solution $x^{opt}(0)$ of problem $(lpr(0))$. For the basic solution $x^{opt}(0)$, let B be the matrix composed of the basic columns of A and let N be the matrix composed of the nonbasic columns of A. For the basis matrix B, the basic components of $x^{opt}(0)$, written as $x^{opt}(0)_B$, can be expressed as

$$x^{opt}(0)_B = B^{-1}(b - Nx^{opt}(0)_N)$$

Furthermore, because $x^{opt}(0)$ is feasible,

$$\ell_B \leq B^{-1}(b - Nx^{opt}(0)_N) \leq u_B$$

where ℓ_B is a column vector of the lower bounds on the structural basic variables, and u_B is a column vector of the upper bounds on the structural basic variables. For each right-hand-side change vector r identified in the RHSSEN statement, PROC LP finds an interval $[\phi_{min}, \phi_{max}]$ such that

$$\ell_B \leq B^{-1}(b + \phi r - Nx^{opt}(0)_N) \leq u_B$$

for $\phi \in [\phi_{min}, \phi_{max}]$. Furthermore, because changes in the right-hand side do not affect the reduced costs, for $\phi \in [\phi_{min}, \phi_{max}]$,

$$x^{opt}(\phi)^T = ((B^{-1}(b + \phi r - Nx^{opt}(0)_N))^T, x^{opt}(0)_N^T)$$

is optimal in $(lpr(\phi))$.

For $\phi = \phi_{min}$ and $\phi = \phi_{max}$, PROC LP reports the following:

- the names of the leaving variables
- the value of the optimal objective in the modified problems
- the RHS values in the modified problems
- the solution status, reduced costs and activities in the modified problems

The leaving variable identifies the basic variable x_i that first reaches either the lower bound ℓ_i or the upper bound u_i as ϕ reaches ϕ_{min} or ϕ_{max}. This is the basic variable that would leave the basis to maintain primal feasibility. Multiple RHSSEN variables can appear in a problem data set.

Price Sensitivity Analysis

Consider the problem $(lpp(\phi))$:

$$\begin{array}{ll} x^{opt}(\phi) = & \min(c + \phi r)^T x \\ \text{subject to} & Ax \{\geq, =, \leq\} b \\ & \ell \leq x \leq u \end{array}$$

where r is a price change vector.

Let $x^{opt}(\phi)$ denote an optimal basic feasible solution to $(lpp(\phi))$. PROC LP can be used to examine the effects of changes in ϕ on the solution $x^{opt}(0)$ of problem $(lpp(0))$. For the basic solution $x^{opt}(0)$, let B be

the matrix composed of the basic columns of A and let N be the matrix composed of the nonbasic columns of A. For basis matrix B, the reduced cost associated with the ith variable can be written as

$$rc_i(\phi) = ((c + \phi r)_N^T - (c + \phi r)_B^T B^{-1} N)_i$$

where $(c + \phi r)_N$ and $(c + \phi r)_B$ is a partition of the vector of price coefficients into nonbasic and basic components. Because $x^{opt}(0)$ is optimal in $(lpp(0))$, the reduced costs satisfy

$$rc_i(\phi) \geq 0$$

if the nonbasic variable in column i is at its lower bound, and

$$rc_i(\phi) \leq 0$$

if the nonbasic variable in column i is at its upper bound.

For each price coefficient change vector r identified with the keyword PRICESEN in the TYPE variable, PROC LP finds an interval $[\phi_{min}, \phi_{max}]$ such that for $\phi \in [\phi_{min}, \phi_{max}]$,

$$rc_i(\phi) \geq 0$$

if the nonbasic variable in column i is at its lower bound, and

$$rc_i(\phi) \leq 0$$

if the nonbasic variable in column i is at its upper bound. Because changes in the price coefficients do not affect feasibility, for $\phi \in [\phi_{min}, \phi_{max}]$, $x^{opt}(\phi)$ is optimal in $(lpp(\phi))$. For $\phi = \phi_{min}$ and $\phi = \phi_{max}$, PROC LP reports the following:

- the names of entering variables
- the value of the optimal objective in the modified problems
- the price coefficients in the modified problems
- the solution status, reduced costs, and activities in the modified problems

The entering variable identifies the variable whose reduced cost first goes to zero as ϕ reaches ϕ_{min} or ϕ_{max}. This is the nonbasic variable that would enter the basis to maintain optimality (dual feasibility). Multiple PRICESEN variables may appear in a problem data set.

Range Analysis

Range analysis is sensitivity analysis for specific change vectors. As with the sensitivity analysis case, care must be used in interpreting the results of range analysis when the problem has integers or the preprocessing option is enabled.

Right-Hand-Side Range Analysis

The effects on the optimal solution of changes in each right-hand-side value can be studied using the RANGERHS option in the PROC LP or RESET statement. This option results in sensitivity analysis for the m right-hand-side change vectors specified by the columns of the $m \times m$ identity matrix.

Price Range Analysis

The effects on the optimal solution of changes in each price coefficient can be studied using the RANGEPRICE option in the PROC LP or RESET statement. This option results in sensitivity analysis for the n price change vectors specified by the rows of the $n \times n$ identity matrix.

Parametric Programming

Sensitivity analysis and range analysis examine how the optimal solution behaves with respect to perturbations of model parameter values. These approaches assume that the basis at optimality is not allowed to change. When greater flexibility is desired and a change of basis is acceptable, parametric programming can be used.

As with the sensitivity analysis case, care must be used in interpreting the results of parametric programming when the problem has integers or the preprocessing option is enabled.

Right-Hand-Side Parametric Programming

As discussed in the section "Right-Hand-Side Sensitivity Analysis" on page 217, for each right-hand-side change vector r, PROC LP finds an interval $[\phi_{min}, \phi_{max}]$ such that for $\phi \in [\phi_{min}, \phi_{max}]$,

$$x^{opt}(\phi)^T = ((B^{-1}(b + \phi r - N x^{opt}(0)_N))^T, x^{opt}(0)_N^T)$$

is optimal in $(lpr(\phi))$ for the fixed basis B. Leaving variables that inhibit further changes in ϕ without a change in the basis B are associated with the quantities ϕ_{min} and ϕ_{max}. By specifying RHSPHI=Φ in either the PROC LP statement or the RESET statement, you can examine the solution $x^{opt}(\phi)$ as ϕ increases or decreases from 0 to Φ.

When RHSPHI=Φ is specified, the procedure first finds the interval $[\phi_{min}, \phi_{max}]$ as described previously. Then, if $\Phi \in [\phi_{min}, \phi_{max}]$, no further investigation is needed. However, if $\Phi > \phi_{max}$ or $\Phi < \phi_{min}$, then the procedure attempts to solve the new problem $(lpr(\Phi))$. To accomplish this, it pivots the leaving variable out of the basis while maintaining dual feasibility. If this new solution is primal feasible in $(lpr(\Phi))$, no further investigation is needed; otherwise, the procedure identifies the new leaving variable and pivots it out of the basis, again maintaining dual feasibility. Dual pivoting continues in this manner until a solution that is primal feasible in $(lpr(\Phi))$ is identified. Because dual feasibility is maintained at each pivot, the $(lpr(\Phi))$ primal feasible solution is optimal.

At each pivot, the procedure reports on the variables that enter and leave the basis, the current range of ϕ , and the objective value. When $x^{opt}(\Phi)$ is found, it is displayed. If you want the solution $x^{opt}(\phi)$ at each pivot, then specify the PARAPRINT option in either the PROC LP or the RESET statement.

Price Parametric Programming

As discussed in the section "Price Sensitivity Analysis" on page 218, for each price change vector r, PROC LP finds an interval $[\phi_{min}, \phi_{max}]$ such that for each $\phi \in [\phi_{min}, \phi_{max}]$,

$$rc_i(\phi) = ((c + \phi r)_N^T - (c + \phi r)_B^T B^{-1} N)_i$$

satisfies the conditions for optimality in $(lpp(\phi))$ for the fixed basis B. Entering variables that inhibit further changes in ϕ without a change in the basis B are associated with the quantities ϕ_{min} and ϕ_{max}. By specifying PRICEPHI=Φ in either the PROC LP statement or the RESET statement, you can examine the solution $x^{opt}(\phi)$ as ϕ increases or decreases from 0 to Φ.

When PRICEPHI=Φ is specified, the procedure first finds the interval $[\phi_{min}, \phi_{max}]$, as described previously. Then, if $\Phi \in [\phi_{min}, \phi_{max}]$, no further investigation is needed. However, if $\Phi > \phi_{max}$ or $\Phi < \phi_{min}$, the procedure attempts to solve the new problem $(lpp(\Phi))$. To accomplish this, it pivots the entering variable into the basis while maintaining primal feasibility. If this new solution is dual feasible in $(lpp(\Phi))$, no further investigation is needed; otherwise, the procedure identifies the new entering variable and pivots it into the basis, again maintaining primal feasibility. Pivoting continues in this manner until a solution that is dual feasible in $(lpp(\Phi))$ is identified. Because primal feasibility is maintained at each pivot, the $(lpp(\Phi))$ dual feasible solution is optimal.

At each pivot, the procedure reports on the variables that enter and leave the basis, the current range of ϕ , and the objective value. When $x^{opt}(\Phi)$ is found, it is displayed. If you want the solution $x^{opt}(\phi)$ at each pivot, then specify the PARAPRINT option in either the PROC LP or the RESET statement.

Interactive Facilities

The interactive features of the LP procedure enable you to examine intermediate results, perform sensitivity analysis, parametric programming, and range analysis, and control the solution process.

Controlling Interactive Features

You can gain control of the LP procedure for interactive processing by setting a breakpoint or pressing the CTRL-BREAK key combination, or when certain error conditions are encountered:

- when a feasible solution is found
- at each pivot of the simplex algorithm
- when an integer feasible solution is found
- at each integer pivot of the branch-and-bound algorithm
- after the data are read but before iteration begins
- after at least one integer feasible solution has been found which is within desirable proximity of optimality
- after the problem has been solved but before results are displayed

When the LP procedure pauses, you can enter any of the interactive statements RESET, PIVOT, IPIVOT, PRINT, SHOW, QUIT, and RUN.

Breakpoints are set using the FEASIBLEPAUSE, PAUSE=, IFEASIBLEPAUSE=, IPAUSE=, PROXIMITY-PAUSE=, READPAUSE, and ENDPAUSE options. The LP procedure displays a message on the SAS log when it gives you control because of encountering one of these breakpoints.

During phase 1, 2, or 3, the CTRL-BREAK key pauses the LP procedure and releases the control at the beginning of the next iteration.

The error conditions, which usually cause the LP procedure to pause, include time limit exceeded, phase 1 iterations exceeded, phase 2 iterations exceeded, phase 3 iterations exceeded, and integer iterations exceeded. You can use the RESET statement to reset the option that caused the error condition.

The PIVOT and IPIVOT statements result in control being returned to you after a single simplex algorithm pivot and an integer pivot. The PRINT and SHOW statements display current solution information and return control to you. On the other hand, the QUIT statement requests that you leave the LP procedure immediately. If you want to quit but save output data sets, then type QUIT/SAVE. The RUN statement requests the LP procedure to continue its execution immediately.

Displaying Intermediate Results

Once you have control of the procedure, you can examine the current values of the options and the status of the problem being solved using the SHOW statement. All displaying done by the SHOW statement goes to the SAS log.

Details about the current status of the solution are obtained using the PRINT statement. The various display options enable you to examine parts of the variable and constraint summaries, display the current tableau, perform sensitivity analysis on the current solution, and perform range analysis.

Interactive Facilities in Batch Mode

All of the interactive statements can be used when processing in batch mode. This is particularly convenient when the interactive facilities are used to combine different search strategies in solving integer problems.

Sensitivity Analysis

Two features that enhance the ability to perform sensitivity analysis need further explanation. When you specify /SENSITIVITY in a PRINT COLUMN(*colnames*) statement, the LP procedure defines a new change row to use in sensitivity analysis and parametric programming. This new change row has a +1 entry for each variable listed in the PRINT statement. This enables you to define new change rows interactively.

When you specify /SENSITIVITY in a PRINT ROW (*rownames*) statement, the LP procedure defines a new change column to use in sensitivity analysis and parametric programming. This new change column has a +1 entry for each right-hand-side coefficient listed in the PRINT statement. This enables you to define new change columns interactively.

In addition, you can interactively change the RHSPHI= and PRICEPHI= options using the RESET statement. This enables you to perform parametric programming interactively.

Memory Management

There are no restrictions on the problem size in the LP procedure. The number of constraints and variables in a problem that PROC LP can solve depends on the host platform, the available memory, and the available disk space for utility data sets.

Memory usage is affected by a great many factors including the density of the technological coefficient matrix, the model structure, and the density of the decomposed basis matrix. The algorithm requires that the decomposed basis fit completely in memory. Any additional memory is used for nonbasic columns. The partition between the decomposed basis and the nonbasic columns is dynamic so that as the inverse grows, which typically happens as iterations proceed, more memory is available to it and less is available for the nonbasic columns.

The LP procedure determines the initial size of the decomposed basis matrix. If the area used is too small, PROC LP must spend time compressing this matrix, which degrades performance. If PROC LP must compress the decomposed basis matrix on the average more than 15 times per iteration, then the size of the memory devoted to the basis is increased. If the work area cannot be made large enough to invert the basis, an error return occurs. On the other hand, if PROC LP compresses the decomposed basis matrix on the average once every other iteration, then memory devoted to the decomposed basis is decreased, freeing memory for the nonbasic columns.

For many models, memory constraints are not a problem because both the decomposed basis and all the nonbasic columns will have no problem fitting. However, when the models become large relative to the available memory, the algorithm tries to adjust memory distribution in order to solve the problem. In the worst cases, only one nonbasic column fits in memory with the decomposed basis matrix.

Problems involving memory use can occur when solving mixed-integer problems. Data associated with each node in the branch-and-bound tree must be kept in memory. As the tree grows, competition for memory by the decomposed basis, the nonbasic columns, and the branch-and-bound tree may become critical. If the situation becomes critical, the procedure automatically switches to branching strategies that use less memory. However, it is possible to reach a point where no further processing is possible. In this case, PROC LP terminates on a memory error.

Output Data Sets

The LP procedure can optionally produce five output data sets. These are the ACTIVEOUT=, PRIMALOUT=, DUALOUT=, TABLEAUOUT=, and MPSOUT= data sets. Each contains two variables that identify the particular problem in the input data set. These variables are

_OBJ_ID_	identifies the objective function ID.
_RHS_ID_	identifies the right-hand-side variable.

Additionally, each data set contains other variables, which are discussed below.

ACTIVEOUT= Data Set

The ACTIVEOUT= data set contains a representation of the current active branch-and-bound tree. You can use this data set to initialize the branch-and-bound tree to continue iterations on an incompletely solved problem. Each active node in the tree generates two observations in this data set. The first is a 'LOWERBD' observation that is used to reconstruct the lower-bound constraints on the currently described active node. The second is an 'UPPERBD' observation that is used to reconstruct the upper-bound constraints on the currently described active node. In addition to these, an observation that describes the current best integer solution is included. The data set contains the following variables:

STATUS contains the keywords LOWERBD, UPPERBD, and INTBEST for identifying the type of observation.

PROB contains the problem number for the current observation.

OBJECT contains the objective value of the parent problem that generated the current observation's problem.

SINFEA contains the sum of the integer infeasibilities of the current observation's problem.

PROJEC contains the data needed for CANSELECT=PROJECT when the branch-and-bound tree is read using the ACTIVEIN= option.

PSEUDO contains the data needed for CANSELECT=PSEUDOC when the branch-and-bound tree is read using the ACTIVEIN= option.

INTEGER VARIABLES Integer-constrained structural variables are also included in the ACTIVEOUT= data set. For each observation, these variables contain values for defining the active node in the branch-and-bound tree.

PRIMALOUT= Data Set

The PRIMALOUT= data set contains the current primal solution. If the problem has integer-constrained variables, the PRIMALOUT= data set contains the current best integer feasible solution. If none have been found, the PRIMALOUT= data set contains the relaxed solution. In addition to _OBJ_ID_ and _RHS_ID_, the data set contains the following variables:

VAR identifies the variable name.

TYPE identifies the type of the variable as specified in the input data set. Artificial variables are labeled as type 'ARTIFCL'.

STATUS identifies whether the variable is basic, nonbasic, or at an upper bound in the current solution.

LBOUND contains the input lower bound on the variable unless the variable is integer-constrained and an integer solution is given. In this case, _LBOUND_ contains the lower bound on the variable needed to realize the integer solution on subsequent calls to PROC LP when using the PRIMALIN= option.

VALUE identifies the value of the variable in the current solution or the current best integer feasible solution.

UBOUND contains the input upper bound on the variable unless the variable is integer-constrained and an integer solution is given. In this case, _UBOUND_ contains the upper bound on

the variable needed to realize the integer solution on subsequent calls to PROC LP when using the PRIMALIN= option.

PRICE contains the input price coefficient of the variable.

_R_COST_ identifies the value of the reduced cost in the current solution. Example 5.3 in the section "Examples: LP Procedure" on page 232 shows a typical PRIMALOUT= data set. Note that it is necessary to include the information on objective function and right-hand side in order to distinguish problems in multiple problem data sets.

DUALOUT= Data Set

The DUALOUT= data set contains the dual solution for the current solution. If the problem has integer-constrained variables, the DUALOUT= data set contains the dual for the current best integer solution, if any. Otherwise it contains the dual for the relaxed solution. In addition to _OBJ_ID_ and _RHS_ID_, it contains the following variables:

_ROW_ID_ identifies the row or constraint name.

TYPE identifies the type of the row as specified in the input data set.

RHS gives the value of the right-hand side on input.

_L_RHS_ gives the lower bound for the row evaluated from the input right-hand-side value, the TYPE of the row, and the value of the RANGE variable for the row.

VALUE gives the value of the row, at optimality, excluding logical variables.

_U_RHS_ gives the upper bound for the row evaluated from the input right-hand-side value, the TYPE of the row, and the value of the RANGE variable for the row.

DUAL gives the value of the dual variable associated with the row.

TABLEAUOUT= Data Set

The TABLEAUOUT= data set contains the current tableau. Each observation, except for the first, corresponds to a basic variable in the solution. The observation labeled R_COSTS contains the reduced costs $c_N^T - c_B^T B^{-1} N$. In addition to _OBJ_ID_ and _RHS_ID_, it contains the following variables:

BASIC gives the names of the basic variables in the solution.

INVB_R gives the values of $B^{-1}r$, where r is the right-hand-side vector.

STRUCTURAL VARIABLES give the values in the tableau, namely $B^{-1}A$.

MPSOUT= Data Set

The MPSOUT= data set contains problem data converted from a PROC LP format into an MPS-format SAS data set. The six fields, FIELD1 to FIELD6, in the MPSOUT= data set correspond to the six columns in MPS standard. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

Input Data Sets

In addition to the DATA= input data set, PROC LP recognizes the ACTIVEIN= and the PRIMALIN= data sets.

ACTIVEIN= Data Set

The ACTIVEIN= data set contains a representation of the current active tree. The format is identical to that of the ACTIVEOUT= data set.

PRIMALIN= Data Set

The format of the PRIMALIN= data set is identical to the PRIMALOUT= data set. PROC LP uses the PRIMALIN= data set to identify variables at their upper bounds in the current solution and variables that are basic in the current solution.

You can add observations to the end of the problem data set if they define cost (right-hand-side) sensitivity change vectors and have PRICESEN (RHSSEN) types. This enables you to solve a problem, save the solution in a SAS data set, and perform sensitivity analysis later. You can also use the PRIMALIN= data set to restart problems that have not been completely solved or to which new variables have been added.

Displayed Output

The output from the LP procedure is discussed in the following six sections:

- Problem Summary
- Solution Summary including a Variable Summary and a Constraint Summary
- Infeasible Information Summary
- RHS Sensitivity Analysis Summary (the RHS Range Analysis Summary is not discussed)
- Price Sensitivity Analysis Summary (the Price Range Analysis Summary is not discussed)
- Iteration Log

For integer-constrained problems, the procedure also displays an Integer Iteration Log. The description of this Log can be found in the section "Integer Programming" on page 210. When you request that the tableau be displayed, the procedure displays the Current Tableau. The description of this can be found in the section "The Reduced Costs, Dual Activities, and Current Tableau" on page 206.

A problem data set can contain a set of constraints with several right-hand sides and several objective functions. PROC LP considers each combination of right-hand side and objective function as defining a new linear programming problem and solves each, performing all specified sensitivity analysis on each problem. For each problem defined, PROC LP displays a new sequence of output sections. Example 5.1 in the section "Examples: LP Procedure" on page 232 discusses each of these elements.

The LP procedure produces the following displayed output by default.

The Problem Summary

The problem summary includes the

- type of optimization and the name of the objective row (as identified by the ID or ROW variable)
- name of the SAS variable that contains the right-hand-side constants
- name of the SAS variable that contains the type keywords
- density of the coefficient matrix (the ratio of the number of nonzero elements to the number of total elements) after the slack and surplus variables have been appended
- number of each type of variable in the mathematical program
- number of each type of constraint in the mathematical program

The Solution Summary

The solution summary includes the

- termination status of the procedure
- objective value of the current solution
- number of phase 1 iterations that were completed
- number of phase 2 iterations that were completed
- number of phase 3 iterations that were completed
- number of integer iterations that were completed
- number of integer feasible solutions that were found
- number of initial basic feasible variables identified
- time used in solving the problem excluding reading the data and displaying the solution
- number of inversions of the basis matrix
- current value of several of the options

The Variable Summary

The variable summary includes the

- column number associated with each structural or logical variable in the problem
- name of each structural or logical variable in the problem. (PROC LP gives the logical variables the name of the constraint ID. If no ID variable is specified, the procedure names the logical variable _OBS*n*_, where *n* is the observation that describes the constraint.)

- variable's status in the current solution. The status can be BASIC, DEGEN, ALTER, blank, LOWBD, or UPPBD, depending upon whether the variable is a basic variable, a degenerate variable (that is, a basic variable whose activity is at its input lower bound), a nonbasic variable that can be brought into the basis to define an alternate optimal solution, a nonbasic variable at its default lower bound 0, a nonbasic variable at its lower bound, or a nonbasic variable at its upper bound.
- type of variable (whether it is logical or structural, and, if structural, its bound type, or other value restriction). See Example 5.1 for a list of possible types in the variable summary.
- value of the objective coefficient associated with each variable
- activity of the variable in the current solution
- variable's reduced cost in the current solution

The Constraint Summary

The constraint summary includes the

- constraint row number and its ID
- kind of constraint (whether it is an OBJECTIVE, LE, EQ, GE, RANGELE, RANGEEQ, RANGEGE, or FREE row)
- number of the slack or surplus variable associated with the constraint row
- value of the right-hand-side constant associated with the constraint row
- current activity of the row (excluding logical variables)
- current activity of the dual variable (shadow price) associated with the constraint row

The Infeasible Information Summary

The infeasible information summary includes the

- name of the infeasible row or variable
- current activity for the row or variable
- type of the row or variable
- value of right-hand-side constant
- name of each nonzero and nonmissing variable in the row
- activity and upper and lower bounds for the variable

The RHS Sensitivity Analysis Summary

The RHS sensitivity analysis summary includes the

- value of ϕ_{min}
- leaving variable when $\phi = \phi_{min}$
- objective value when $\phi = \phi_{min}$
- value of ϕ_{max}
- leaving variable when $\phi = \phi_{max}$
- objective value when $\phi = \phi_{max}$
- column number and name of each logical and structural variable
- variable's status when $\phi \in [\phi_{min}, \phi_{max}]$
- variable's reduced cost when $\phi \in [\phi_{min}, \phi_{max}]$
- value of right-hand-side constant when $\phi = \phi_{min}$
- activity of the variable when $\phi = \phi_{min}$
- value of right-hand-side constant when $\phi = \phi_{max}$
- activity of the variable when $\phi = \phi_{max}$

The Price Sensitivity Analysis Summary

The price sensitivity analysis summary includes the

- value of ϕ_{min}
- entering variable when $\phi = \phi_{min}$
- objective value when $\phi = \phi_{min}$
- value of ϕ_{max}
- entering variable when $\phi = \phi_{max}$
- objective value when $\phi = \phi_{max}$
- column number and name of each logical and structural variable
- variable's status when $\phi \in [\phi_{min}, \phi_{max}]$
- activity of the variable when $\phi \in [\phi_{min}, \phi_{max}]$
- price of the variable when $\phi = \phi_{min}$
- variable's reduced cost when $\phi = \phi_{min}$
- price of the variable when $\phi = \phi_{max}$
- variable's reduced cost when $\phi = \phi_{max}$

The Iteration Log

The iteration log includes the

- phase number
- iteration number in each phase
- name of the leaving variable
- name of the entering variable
- variable's reduced cost
- objective value

ODS Table and Variable Names

PROC LP assigns a name to each table it creates. You can use these names to select output tables when using the Output Delivery System (ODS).

Table 5.9 ODS Tables Produced in PROC LP

Table Name	Description	Statement/Option
ProblemSummary	Problem summary	Default
SolutionSummary	Solution summary	Default
VariableSummary	Variable summary	Default
ConstraintSummary	Constraint summary	Default
IterationLog	Iteration log	FLOW
IntegerIterationLog	Integer iteration log	Default
PriceSensitivitySummary	Price sensitivity analysis summary	Default, PRINT PRICESEN, or PRINT COLUMN/SENSITIVITY
PriceActivities	Price activities at ϕ_{min} and ϕ_{max}	Default, PRINT PRICESEN, or PRINT COLUMN/SENSITIVITY
PriceActivity	Price activity at ϕ_{min} or ϕ_{max}	PRICEPHI= and PARAPRINT
PriceParametricLog	Price parametric programming log	PRICEPHI=
PriceRangeSummary	Price range analysis	RANGEPRICE or PRINT RANGEPRICE
RhsSensitivitySummary	RHS sensitivity analysis summary	Default, PRINT RHSSEN, or PRINT ROW/SENSITIVITY
RhsActivities	RHS activities at ϕ_{min} and ϕ_{max}	Default, PRINT RHSSEN, or PRINT ROW/SENSITIVITY
RhsActivity	RHS activity at ϕ_{min} or ϕ_{max}	RHSPHI= and PARAPRINT
RhsParametricLog	RHS parametric programming log	RHSPHI=
RhsRangeSummary	RHS range analysis	RANGERHS or PRINT RANGERHS
InfeasibilitySummary	Infeasible row or variable summary	Default

Table 5.9 (continued)

Table Name	Description	Statement/Option
InfeasibilityActivity	Variable activity in an infeasible row	Default
CurrentTableau	Current tableau	TABLEAUPRINT or PRINT TABLEAU
Matrix	Technological matrix	PRINT MATRIX
MatrixPicture	Technological matrix picture	PRINT MATRIX/PICTURE
MatrixPictureLegend	Technological matrix picture legend	PRINT MATRIX/PICTURE

The following table lists the variable names of the preceding tables used in the ODS template of the LP procedure.

Table 5.10 Variable Names for the ODS Tables Produced in PROC LP

Table Name	Variables
VariableSummary	VarName, Status, Type, Price, Activity, ReducedCost
ConstraintSummary	Row, RowName, Type, SSCol, Rhs, Activity, Dual
IterationLog	Phase, Iteration, EnterVar, EnterCol, LeaveVar, LeaveCol, ReducedCost, ObjValue
IntegerIterationLog	Iteration, Problem, Condition, Objective, Branch, Value, SumOfInfeas, Active, Proximity
PriceActivities	Col, VarName, Status, Activity, MinPrice, MinReducedCost, MaxPrice, MaxReducedCost
PriceActivity	Col, VarName, Status, Activity, Price, ReducedCost
PriceParametricLog	LeaveVar, LeaveCol, EnterVar, EnterCol, ObjValue, CurrentPhi
PriceRangeSummary	Col, VarName, MinPrice, MinEnterVar, MinObj, MaxPrice, MaxEnterVar, MaxObj
RhsActivities	Col, VarName, Status, ReducedCost, MinRhs, MinActivity, MaxRhs, MaxActivity
RhsActivity	Col, VarName, Status, ReducedCost, Rhs, Activity,
RhsParametricLog	LeaveVar, LeaveCol, EnterVar, EnterCol, ObjValue, CurrentPhi
RhsRangeSummary	RowName, MinRhs, MinLeaveVar, MinObj, MaxRhs, MaxLeaveVar, MaxObj
InfeasibilityActivity	VarName, Coefficient, Activity, Lower, Upper

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory used by the SAS System. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit, which is sufficient for many applications. However, to solve most realistic optimization problems, the LP procedure might require more memory. Increasing the memory limit can reduce the chance of an out-of-memory condition.

NOTE: The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify -MEMSIZE 0 to indicate all available memory should be used, but this setting should be used with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify -MEMSIZE 0. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, -MEMSIZE 500M might be sufficient to allow the procedure to run without an out-of-memory condition. When problems have millions of variables, -MEMSIZE 1000M or higher might be needed. These are "rules of thumb"—problems with atypical structure, density, or other characteristics can increase the optimizer's memory requirements.

The MEMSIZE option can be specified at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the *SAS Companion* book for your operating system.

To report a procedure's memory consumption, you can use the FULLSTIMER option. The syntax is described in the *SAS Companion* book for your operating system.

Examples: LP Procedure

The following examples illustrate some of the capabilities of PROC LP. These examples, together with the other SAS/OR examples, can be found in the SAS sample library. A description of the features of PROC LP as shown in the examples are

- Example 5.1 shows dense input format.
- Example 5.2 shows sparse input format.
- Example 5.3 uses the RANGEPRICE option to show you the range over which each objective coefficient can vary without changing the variables in the basis.
- Example 5.4 shows more sensitivity analysis and restarting a problem.
- Example 5.5 shows parametric programming.
- Example 5.6 shows special ordered sets.
- Example 5.7 shows goal programming.
- Example 5.8 shows integer programming.
- Example 5.9 shows an infeasible problem.
- Example 5.10 shows restarting integer programs.
- Example 5.11 controls the search of the branch-and-bound tree.
- Example 5.12 shows matrix generation and report writing for an assignment problem.
- Example 5.13 shows matrix generation and report writing for a scheduling problem.
- Example 5.14 shows a multicommodity transshipment problem.

- Example 5.15 shows migration to PROC OPTLP via the MPSOUT= option.
- Example 5.16 shows migration to PROC OPTMODEL.
- Example 5.17 shows migration to PROC OPTMODEL.

Example 5.1: An Oil Blending Problem

The blending problem presented in the introduction is a good example for demonstrating some of the features of the LP procedure. Recall that a step in refining crude oil into finished oil products involves a distillation process that splits crude into various streams. Suppose that there are three types of crude available: Arabian light, Arabian heavy, and Brega. These are distilled into light naphtha, intermediate naphtha, and heating oil. Using one of two recipes, these in turn are blended into jet fuel.

Assume that you can sell as much fuel as is produced. What production strategy maximizes the profit from jet fuel sales? The following SAS code demonstrates a way of answering this question using linear programming. The SAS data set is a representation of the formulation for this model given in the introductory section.

```
data;
   input _row_ $17.
         a_light a_heavy brega naphthal naphthai heatingo jet_1
         jet_2  _type_ $ _rhs_;
   datalines;
profit             -175 -165 -205   0  0  0  300  300  max      .
naphtha_l_conv      .035 .030 .045  -1  0  0    0    0   eq       0
naphtha_i_conv      .100 .075 .135   0 -1  0    0    0   eq       0
heating_o_conv      .390 .300 .430   0  0 -1    0    0   eq       0
recipe_1               0    0    0   0 .3 .7   -1    0   eq       0
recipe_2               0    0    0  .2  0 .8    0   -1   eq       0
available            110  165   80   .  .  .    .    .   upperbd .
;
```

The _ROW_ variable contains the names of the rows in the model; the variables A_LIGHT to JET_2 are the names of the structural variables in the model; the _TYPE_ variable contains the keywords that tell the LP procedure how to interpret each row in the model; and the _RHS_ variable gives the value of the right-hand-side constants.

The structural variables are interpreted as the quantity of each type of constituent or finished product. For example, the value of A_HEAVY in the solution is the amount of Arabian heavy crude to buy while the value of JET_1 in the solution is the amount of recipe 1 jet fuel that is produced. As discussed previously, the values given in the model data set are the technological coefficients whose interpretation depends on the model. In this example, the coefficient -175 in the PROFIT row for the variable A_LIGHT gives a cost coefficient (because the row with _ROW_=PROFIT has _TYPE_=MAX) for the structural variable A_LIGHT. This means that for each unit of Arabian heavy crude purchased, a cost of 175 units is incurred.

The coefficients 0.035, 0.100, and 0.390 for the A_LIGHT variable give the percentages of each unit of Arabian light crude that is distilled into the light naphtha, intermediate naphtha, and heating oil components. The 110 value in the row _ROW_=AVAILABLE gives the quantity of Arabian light that is available.

PROC LP produces the following Problem Summary output. Included in the summary is an identification of the objective, defined by the first observation of the problem data set; the right-hand-side variable, defined by the variable _RHS_; and the type identifier, defined by the variable _TYPE_. See Output 5.1.1.

Output 5.1.1 Problem Summary for the Oil Blending Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

The next section of output (Output 5.1.2) contains the Solution Summary, which indicates whether or not an optimal solution was found. In this example, the procedure terminates successfully (with an optimal solution), with 1544 as the value of the objective function. Also included in this section of output is the number of phase 1 and phase 2 iterations, the number of variables used in the initial basic feasible solution, and the time used to solve the problem. For several options specified in the PROC LP statement, the current option values are also displayed.

Output 5.1.2 Solution Summary for the Oil Blending Problem

The LP Procedure

Solution Summary Terminated Successfully	
Objective Value	1544
Phase 1 Iterations	0
Phase 2 Iterations	4
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	5
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

The next section of output (Output 5.1.3) contains the Variable Summary. A line is displayed for each variable in the mathematical program with the variable name, the status of the variable in the solution, the type of variable, the variable's price coefficient, the activity of the variable in the solution, and the reduced cost for the variable. The status of a variable can be

BASIC	if the variable is a basic variable in the solution.
DEGEN	if the variable is a basic variable whose activity is at its input lower bound.
ALTER	if the variable is nonbasic and is basic in an alternate optimal solution.
LOWBD	if the variable is nonbasic and is at its lower bound.
UPPBD	if the variable is nonbasic and is at its upper bound.

The TYPE column shows how PROC LP interprets the variable in the problem data set. Types include the following:

NON-NEG	if the variable is a nonnegative variable with lower bound 0 and upper bound $+\infty$.
LOWERBD	if the variable has a lower bound specified in a LOWERBD observation and upper bound $+\infty$.
UPPERBD	if the variable has an upper bound that is less than $+\infty$ and lower bound 0. This upper bound is specified in an UPPERBD observation.
UPLOWBD	if the variable has a lower bound specified in a LOWERBD observation and an upper bound specified in an UPPERBD observation.
INTEGER	if the variable is constrained to take integer values. If this is the case, then it must also be upper and lower bounded.
BINARY	if the variable is constrained to take value 0 or 1.
UNRSTRT	if the variable is an unrestricted variable having bounds of $-\infty$ and $+\infty$.
SLACK	if the variable is a slack variable that PROC LP has appended to a LE constraint. For variables of this type, the variable name is the same as the name of the constraint (given in the ROW variable) for which this variable is the slack. A nonzero slack variable indicates that the constraint is not tight. The slack is the amount by which the right-hand side of the constraint exceeds the left-hand side.
SURPLUS	if the variable is a surplus variable that PROC LP has appended to a GE constraint. For variables of this type, the variable name is the same as the name of the constraint (given in the ROW variable) for which this variable is the surplus. A nonzero surplus variable indicates that the constraint is not tight. The surplus is the amount by which the left-hand side of the constraint exceeds the right-hand side.

The Variable Summary gives the value of the structural variables at optimality. In this example, it tells you how to produce the jet fuel to maximize your profit. You should buy 110 units of A_LIGHT and 80 units of BREGA. These are used to make 7.45 units of NAPHTHAL, 21.8 units of NAPHTHAI, and 77.3 units of HEATINGO. These in turn are used to make 60.65 units of JET_1 using recipe 1 and 63.33 units of JET_2 using recipe 2.

Output 5.1.3 Variable Summary for the Oil Blending Problem

The LP Procedure

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	a_light	UPPBD	UPPERBD	-175	110	11.6
2	a_heavy		UPPERBD	-165	0	-21.45
3	brega	UPPBD	UPPERBD	-205	80	3.35
4	naphthal	BASIC	NON-NEG	0	7.45	0
5	naphthai	BASIC	NON-NEG	0	21.8	0
6	heatingo	BASIC	NON-NEG	0	77.3	0
7	jet_1	BASIC	NON-NEG	300	60.65	0
8	jet_2	BASIC	NON-NEG	300	63.33	0

The reduced cost associated with each nonbasic variable is the marginal value of that variable if it is brought into the basis. In other words, the objective function value would (assuming no constraints were violated) increase by the reduced cost of a nonbasic variable if that variable's value increased by one. Similarly, the objective function value would (assuming no constraints were violated) decrease by the reduced cost of a nonbasic variable if that variable's value were decreased by one. Basic variables always have a zero reduced cost. At optimality, for a maximization problem, nonbasic variables that are not at an upper bound have nonpositive reduced costs (for example, A_HEAVY has a reduced cost of -21.45). The objective would decrease if they were to increase beyond their optimal values. Nonbasic variables at upper bounds have nonnegative reduced costs, showing that increasing the upper bound (if the reduced cost is not zero) does not decrease the objective. For a nonbasic variable at its upper bound, the reduced cost is the marginal value of increasing its upper bound, often called its shadow price.

For minimization problems, the definition of reduced costs remains the same but the conditions for optimality change. For example, at optimality the reduced costs of all non-upper-bounded variables are nonnegative, and the reduced costs of upper-bounded variables at their upper bound are nonpositive.

The next section of output (Output 5.1.4) contains the Constraint Summary. For each constraint row, free row, and objective row, a line is displayed in the Constraint Summary. Included on the line are the constraint name, the row type, the slack or surplus variable associated with the row, the right-hand-side constant associated with the row, the activity of the row (not including the activity of the slack and surplus variables), and the dual activity (shadow prices).

A dual variable is associated with each constraint row. At optimality, the value of this variable, the dual activity, tells you the marginal value of the right-hand-side constant. For each unit increase in the right-hand-side constant, the objective changes by this amount. This quantity is also known as the shadow price. For example, the marginal value for the right-hand-side constant of constraint HEATING_O_CONV is -450.

Output 5.1.4 Constraint Summary for the Oil Blending Problem

The LP Procedure

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1544	.
2	naphtha_l_conv	EQ	.	0	0	-60
3	naphtha_i_conv	EQ	.	0	0	-90
4	heating_o_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

Example 5.2: A Sparse View of the Oil Blending Problem

Typically, mathematical programming models are very sparse. This means that only a small percentage of the coefficients are nonzero. The sparse problem input is ideal for these models. The oil blending problem in the section "An Introductory Example" on page 171 has a sparse form. This example shows the same problem in a sparse form with the data given in a different order. In addition to representing the problem in a concise form, the sparse format

- allows long column names
- enables easy matrix generation (see Example 5.12, Example 5.13, and Example 5.14)
- is compatible with MPS sparse format

The model in the sparse format is solved by invoking PROC LP with the SPARSEDATA option as follows.

```
data oil;
      format _type_ $8. _col_ $14. _row_ $16. ;
      input _type_ $ _col_ $ _row_ $ _coef_ ;
datalines;
max       .              profit                    .
.         arabian_light  profit                 -175
.         arabian_heavy  profit                 -165
.         brega          profit                 -205
.         jet_1          profit                  300
.         jet_2          profit                  300
eq        .              napha_l_conv              .
.         arabian_light  napha_l_conv           .035
.         arabian_heavy  napha_l_conv           .030
.         brega          napha_l_conv           .045
.         naphtha_light  napha_l_conv             -1
eq        .              napha_i_conv              .
.         arabian_light  napha_i_conv           .100
.         arabian_heavy  napha_i_conv           .075
.         brega          napha_i_conv           .135
.         naphtha_inter  napha_i_conv             -1
eq        .              heating_oil_conv          .
```

```
.        arabian_light heating_oil_conv       .390
.        arabian_heavy heating_oil_conv       .300
.        brega         heating_oil_conv       .430
.        heating_oil   heating_oil_conv        -1
eq       .             recipe_1                 .
.        naphtha_inter recipe_1                .3
.        heating_oil   recipe_1                .7
eq       .             recipe_2                 .
.        jet_1         recipe_1                -1
.        naphtha_light recipe_2                .2
.        heating_oil   recipe_2                .8
.        jet_2         recipe_2                -1
.        _rhs_         profit                   0
upperbd  .             available                .
.        arabian_light available               110
.        arabian_heavy available               165
.        brega         available                80
;

proc lp SPARSEDATA;
run;
```

The output from PROC LP follows.

Output 5.2.1 Output for the Sparse Oil Blending Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

The LP Procedure

Solution Summary	
Terminated Successfully	
Objective Value	1544
Phase 1 Iterations	0
Phase 2 Iterations	5
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	5
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

The LP Procedure

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	arabian_heavy		UPPERBD	-165	0	-21.45
2	arabian_light	UPPBD	UPPERBD	-175	110	11.6
3	brega	UPPBD	UPPERBD	-205	80	3.35
4	heating_oil	BASIC	NON-NEG	0	77.3	0
5	jet_1	BASIC	NON-NEG	300	60.65	0
6	jet_2	BASIC	NON-NEG	300	63.33	0
7	naphtha_inter	BASIC	NON-NEG	0	21.8	0
8	naphtha_light	BASIC	NON-NEG	0	7.45	0

The LP Procedure

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1544	.
2	napha_l_conv	EQ	.	0	0	-60
3	napha_i_conv	EQ	.	0	0	-90
4	heating_oil_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

Example 5.3: Sensitivity Analysis: Changes in Objective Coefficients

Simple solution of a linear program is often not enough. A manager needs to evaluate how sensitive the solution is to changing assumptions. The LP procedure provides several tools that are useful for "what if," or sensitivity, analysis. One tool studies the effects of changes in the objective coefficients.

For example, in the oil blending problem, the cost of crude and the selling price of jet fuel can be highly variable. If you want to know the range over which each objective coefficient can vary without changing the variables in the basis, you can use the RANGEPRICE option in the PROC LP statement.

```
proc lp data=oil sparsedata
   rangeprice primalout=solution;
run;
```

In addition to the Problem and Solution summaries, the LP procedure produces a Price Range Summary, shown in Output 5.3.1.

For each structural variable, the upper and lower ranges of the price (objective function coefficient) and the objective value are shown. The blocking variables, those variables that would enter the basis if the objective coefficient were perturbed further, are also given. For example, the output shows that if the cost of ARABIAN_LIGHT crude were to increase from 175 to 186.6 per unit (remember that you are maximizing profit so the ARABIAN_LIGHT objective coefficient would decrease from -175 to -186.6), then it would become optimal to use less of this crude for any fractional increase in its cost. Increasing the unit cost to 186.6 would drive its reduced cost to zero. Any additional increase would drive its reduced cost negative and would destroy the optimality conditions; thus, you would want to use less of it in your processing. The output shows that, at the point where the reduced cost is zero, you would only be realizing a profit of 268 = 1544 - (110 $\times$ 11.6) and that ARABIAN_LIGHT enters the basis, that is, leaves its upper bound. On the other hand, if the cost of ARABIAN_HEAVY were to decrease to 143.55, you would want to stop using the formulation of 110 units of ARABIAN_LIGHT and 80 units of BREGA and switch to a production scheme that included ARABIAN_HEAVY, in which case the profit would increase from the 1544 level.

Output 5.3.1 Price Range Summary for the Oil Blending Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

Solution Summary Terminated Successfully	
Objective Value	1544
Phase 1 Iterations	0
Phase 2 Iterations	5
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	5
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Output 5.3.1 *continued*

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	arabian_heavy		UPPERBD	-165	0	-21.45
2	arabian_light	UPPBD	UPPERBD	-175	110	11.6
3	brega	UPPBD	UPPERBD	-205	80	3.35
4	heating_oil	BASIC	NON-NEG	0	77.3	0
5	jet_1	BASIC	NON-NEG	300	60.65	0
6	jet_2	BASIC	NON-NEG	300	63.33	0
7	naphtha_inter	BASIC	NON-NEG	0	21.8	0
8	naphtha_light	BASIC	NON-NEG	0	7.45	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1544	.
2	napha_l_conv	EQ	.	0	0	-60
3	napha_i_conv	EQ	.	0	0	-90
4	heating_oil_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

Price Range Analysis

		Minimum Phi			Maximum Phi		
Col	Variable Name	Price	Entering	Objective	Price	Entering	Objective
1	arabian_heavy	-INFINITY	.	1544	-143.55	arabian_heavy	1544
2	arabian_light	-186.6	arabian_light	268	INFINITY	.	INFINITY
3	brega	-208.35	brega	1276	INFINITY	.	INFINITY
4	heating_oil	-7.790698	brega	941.77907	71.5	arabian_heavy	7070.95
5	jet_1	290.19034	brega	949.04392	392.25806	arabian_heavy	7139.4516
6	jet_2	290.50992	brega	942.99292	387.19512	arabian_heavy	7066.0671
7	naphtha_inter	-24.81481	brega	1003.037	286	arabian_heavy	7778.8
8	naphtha_light	-74.44444	brega	989.38889	715	arabian_heavy	6870.75

Note that in the PROC LP statement, the PRIMALOUT= SOLUTION option was given. This caused the procedure to save the optimal solution in a SAS data set named SOLUTION. This data set can be used to perform further analysis on the problem without having to restart the solution process. Example 5.4 shows how this is done. A display of the data follows in Output 5.3.2.

Output 5.3.2 The PRIMALOUT= Data Set for the Oil Blending Problem

Obs	_OBJ_ID_	_RHS_ID_	_VAR_	_TYPE_	_STATUS_	_LBOUND_	_VALUE_
1	profit	_rhs_	arabian_heavy	UPPERBD		0	0.00
2	profit	_rhs_	arabian_light	UPPERBD	_UPPER_	0	110.00
3	profit	_rhs_	brega	UPPERBD	_UPPER_	0	80.00
4	profit	_rhs_	heating_oil	NON-NEG	_BASIC_	0	77.30
5	profit	_rhs_	jet_1	NON-NEG	_BASIC_	0	60.65
6	profit	_rhs_	jet_2	NON-NEG	_BASIC_	0	63.33
7	profit	_rhs_	naphtha_inter	NON-NEG	_BASIC_	0	21.80
8	profit	_rhs_	naphtha_light	NON-NEG	_BASIC_	0	7.45
9	profit	_rhs_	PHASE_1_OBJECTIV	OBJECT	_DEGEN_	0	0.00
10	profit	_rhs_	profit	OBJECT	_BASIC_	0	1544.00

Obs	_UBOUND_	_PRICE_	_R_COST_
1	165	-165	-21.45
2	110	-175	11.60
3	80	-205	3.35
4	1.7977E308	0	0.00
5	1.7977E308	300	0.00
6	1.7977E308	300	0.00
7	1.7977E308	0	-0.00
8	1.7977E308	0	0.00
9	0	0	0.00
10	1.7977E308	0	0.00

Example 5.4: Additional Sensitivity Analysis

The objective coefficient ranging analysis, discussed in the last example, is useful for assessing the effects of changing costs and returns on the optimal solution if each objective function coefficient is modified in isolation. However, this is often not the case.

Suppose you anticipate that the cost of crude will be increasing and you want to examine how that will affect your optimal production plans. Furthermore, you estimate that if the price of ARABIAN_LIGHT goes up by 1 unit, then the price of ARABIAN_HEAVY will rise by 1.2 units and the price of BREGA will increase by 1.5 units. However, you plan on passing some of your increased overhead on to your jet fuel customers, and you decide to increase the price of jet fuel 1 unit for each unit of increased cost of ARABIAN_LIGHT.

An examination of the solution sensitivity to changes in the cost of crude is a two-step process. First, add the information on the proportional rates of change in the crude costs and the jet fuel price to the problem data set. Then, invoke the LP procedure. The following program accomplishes this. First, it adds a new row, named CHANGE, to the model. It gives this row a type of PRICESEN. That tells PROC LP to perform objective function coefficient sensitivity analysis using the given rates of change. The program then invokes PROC LP to perform the analysis. Notice that the PRIMALIN= SOLUTION option is used in the PROC LP statement. This tells the LP procedure to use the saved solution. Although it is not necessary to do this, it will eliminate the need for PROC LP to re-solve the problem and can save computing time.

```
data sen;
   format _type_ $8. _col_ $14. _row_ $6.;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
pricesen .             change   .
.        arabian_light change   1
.        arabian_heavy change 1.2
.        brega         change 1.5
.        jet_1         change  -1
.        jet_2         change  -1
;

data;
   set oil sen;
run;

proc lp sparsedata primalin=solution;
run;
```

Output 5.4.1 shows the range over which the current basic solution remains optimal so that the current production plan need not change. The objective coefficients are modified by adding ϕ times the change vector given in the SEN data set, where ϕ ranges from a minimum of -4.15891 to a maximum of 29.72973. At the minimum value of ϕ, the profit decreases to 1103.073. This value of ϕ corresponds to an increase in the cost of ARABIAN_HEAVY to 169.99 (namely, $-175 + \phi \times 1.2$), ARABIAN_LIGHT to 179.16 ($= -175 + \phi \times 1$), and BREGA to 211.24 ($= -205 + \phi \times 1.5$), and corresponds to an increase in the price of JET_1 and JET_2 to 304.16 (= 300 + $\phi \times$ (-1)). These values can be found in the Price column under the section labeled Minimum Phi.

Output 5.4.1 The Price Sensitivity Analysis Summary for the Oil Blending Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

Output 5.4.1 *continued*

Solution Summary	
Terminated Successfully	
Objective Value	1544
Phase 1 Iterations	0
Phase 2 Iterations	0
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	7
Time Used (seconds)	0
Number of Inversions	2
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	arabian_heavy		UPPERBD	-165	0	-21.45
2	arabian_light	UPPBD	UPPERBD	-175	110	11.6
3	brega	UPPBD	UPPERBD	-205	80	3.35
4	heating_oil	BASIC	NON-NEG	0	77.3	0
5	jet_1	BASIC	NON-NEG	300	60.65	0
6	jet_2	BASIC	NON-NEG	300	63.33	0
7	naphtha_inter	BASIC	NON-NEG	0	21.8	0
8	naphtha_light	BASIC	NON-NEG	0	7.45	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1544	.
2	napha_l_conv	EQ	.	0	0	-60
3	napha_i_conv	EQ	.	0	0	-90
4	heating_oil_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

Output 5.4.1 *continued*

The LP Procedure

Price Sensitivity Analysis Summary	
Sensitivity Vector change	
Minimum Phi	-4.158907511
Entering Variable	brega
Optimal Objective	1103.0726257
Maximum Phi	29.72972973
Entering Variable	arabian_heavy
Optimal Objective	4695.9459459

				Minimum Phi		Maximum Phi	
Col	Variable Name	Status	Activity	Price	Reduced Cost	Price	Reduced Cost
1	arabian_heavy		0	-169.9907	-24.45065	-129.3243	0
2	arabian_light	UPPBD	110	-179.1589	10.027933	-145.2703	22.837838
3	brega	UPPBD	80	-211.2384	0	-160.4054	27.297297
4	heating_oil	BASIC	77.3	0	0	0	0
5	jet_1	BASIC	60.65	304.15891	0	270.27027	0
6	jet_2	BASIC	63.33	304.15891	0	270.27027	0
7	naphtha_inter	BASIC	21.8	0	0	0	0
8	naphtha_light	BASIC	7.45	0	0	0	0

The Price Sensitivity Analysis Summary also shows the effects of lowering the cost of crude and lowering the price of jet fuel. In particular, at the maximum ϕ of 29.72973, the current optimal production plan yields a profit of 4695.95. Any increase or decrease in ϕ beyond the limits given results in a change in the production plan. More precisely, the columns that constitute the basis change.

Example 5.5: Price Parametric Programming for the Oil Blending Problem

This example continues to examine the effects of a change in the cost of crude and the selling price of jet fuel. Suppose that you know the cost of ARABIAN_LIGHT crude is likely to increase 30 units, with the effects on oil and fuel prices as described in Example 5.4. The analysis in the last example only accounted for an increase of a little over 4 units (because the minimum ϕ was -4.15891). Because an increase in the cost of ARABIAN_LIGHT beyond 4.15891 units requires a change in the optimal basis, it may also require a change in the optimal production strategy. This type of analysis, where you want to find how the solution changes with changes in the objective function coefficients or right-hand-side vector, is called parametric programming.

You can answer this question by using the PRICEPHI= option in the PROC LP statement. The following program instructs PROC LP to continually increase the cost of the crudes and the return from jet fuel using the ratios given previously, until the cost of ARABIAN_LIGHT increases at least 30 units.

```
proc lp sparsedata primalin=solution pricephi=-30;
run;
```

The PRICEPHI= option in the PROC LP statement tells PROC LP to perform parametric programming on any price change vectors specified in the problem data set. The value of the PRICEPHI= option tells PROC LP how far to change the value of ϕ and in what direction. A specification of PRICEPHI=-30 tells PROC LP to continue pivoting until the problem has objective function equal to (original objective function value) $-$ 30 $\times$ (change vector).

Output 5.5.1 shows the result of this analysis. The first page is the Price Sensitivity Analysis Summary, as discussed in Example 5.4. The next page is an accounting for the change in basis as a result of decreasing ϕ beyond -4.1589. It shows that BREGA left the basis at an upper bound and entered the basis at a lower bound. The interpretation of these basis changes can be difficult (Hadley 1962; Dantzig 1963).

The last page of output shows the optimal solution at the displayed value of ϕ, namely -30.6878. At an increase of 30.6878 units in the cost of ARABIAN_LIGHT and the related changes to the other crudes and the jet fuel, it is optimal to modify the production of jet fuel as shown in the activity column. Although this plan is optimal, it results in a profit of 0. This may suggest that the ratio of a unit increase in the price of jet fuel per unit increase in the cost of ARABIAN_LIGHT is lower than desirable.

Output 5.5.1 Price Parametric Programming for the Oil Blending Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

Output 5.5.1 *continued*

Solution Summary	
Terminated Successfully	
Objective Value	1544
Phase 1 Iterations	0
Phase 2 Iterations	0
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	7
Time Used (seconds)	0
Number of Inversions	2
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	arabian_heavy		UPPERBD	-165	0	-21.45
2	arabian_light	UPPBD	UPPERBD	-175	110	11.6
3	brega	UPPBD	UPPERBD	-205	80	3.35
4	heating_oil	BASIC	NON-NEG	0	77.3	0
5	jet_1	BASIC	NON-NEG	300	60.65	0
6	jet_2	BASIC	NON-NEG	300	63.33	0
7	naphtha_inter	BASIC	NON-NEG	0	21.8	0
8	naphtha_light	BASIC	NON-NEG	0	7.45	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1544	.
2	napha_l_conv	EQ	.	0	0	-60
3	napha_i_conv	EQ	.	0	0	-90
4	heating_oil_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

Output 5.5.1 *continued*

The LP Procedure

Price Sensitivity Analysis Summary	
Sensitivity Vector change	
Minimum Phi	-4.158907511
Entering Variable	brega
Optimal Objective	1103.0726257
Maximum Phi	29.72972973
Entering Variable	arabian_heavy
Optimal Objective	4695.9459459

				Minimum Phi		Maximum Phi	
Col	**Variable Name**	**Status**	**Activity**	**Price**	**Reduced Cost**	**Price**	**Reduced Cost**
1	arabian_heavy		0	-169.9907	-24.45065	-129.3243	0
2	arabian_light	UPPBD	110	-179.1589	10.027933	-145.2703	22.837838
3	brega	UPPBD	80	-211.2384	0	-160.4054	27.297297
4	heating_oil	BASIC	77.3	0	0	0	0
5	jet_1	BASIC	60.65	304.15891	0	270.27027	0
6	jet_2	BASIC	63.33	304.15891	0	270.27027	0
7	naphtha_inter	BASIC	21.8	0	0	0	0
8	naphtha_light	BASIC	7.45	0	0	0	0

The LP Procedure

Price Parametric Programming Log			
Sensitivity Vector change			
Leaving Variable	**Entering Variable**	**Objective**	**Current Phi**
brega	brega	1103.0726	-4.158908

The LP Procedure

Price Sensitivity Analysis Summary	
Sensitivity Vector change	
Minimum Phi	-30.68783069
Entering Variable	arabian_light
Optimal Objective	0

Output 5.5.1 *continued*

Col	Variable Name	Status	Activity	Minimum Phi Price	Minimum Phi Reduced Cost
1	arabian_heavy		0	-201.8254	-43.59127
2	arabian_light	ALTER	110	-205.6878	0
3	brega		0	-251.0317	-21.36905
4	heating_oil	BASIC	42.9	0	0
5	jet_1	BASIC	33.33	330.68783	0
6	jet_2	BASIC	35.09	330.68783	0
7	naphtha_inter	BASIC	11	0	0
8	naphtha_light	BASIC	3.85	0	0

What is the optimal return if ϕ is exactly -30? Because the change in the objective is linear as a function of ϕ, you can calculate the objective for any value of ϕ between those given by linear interpolation. For example, for any ϕ between -4.1589 and -30.6878, the optimal objective value is

$$\phi \times (1103.0726 - 0)/(-4.1589 - 30.6878) + b$$

where

$$b = 30.6878 \times (1103.0726 - 0)/(-4.1589 - 30.6878)$$

For ϕ =-30, this is 28.5988.

Example 5.6: Special Ordered Sets and the Oil Blending Problem

Often managers want to evaluate the cost of making a choice among alternatives. In particular, they want to make the most profitable choice. Suppose that only one oil crude can be used in the production process. This identifies a set of variables of which only one can be above its lower bound. This additional restriction could be included in the model by adding a binary integer variable for each of the three crudes. Constraints would be needed that would drive the appropriate binary variable to 1 whenever the corresponding crude is used in the production process. Then a constraint limiting the total of these variables to only one would be added. A similar formulation for a fixed charge problem is shown in Example 5.8.

The SOSLE type implicitly does this. The following DATA step adds a row to the model that identifies which variables are in the set. The SOSLE type tells the LP procedure that only one of the variables in this set can be above its lower bound. If you use the SOSEQ type, it tells PROC LP that exactly one of the variables in the set must be above its lower bound. Only integer variables can be in an SOSEQ set.

```
data special;
   format _type_ $6. _col_ $14. _row_ $8. ;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
SOSLE .             special .
```

```
.     arabian_light special 1
.     arabian_heavy special 1
.     brega         special 1
;

data;
   set oil special;
run;

proc lp sparsedata;
run;
```

Output 5.6.1 includes an Integer Iteration Log. This log shows the progress that PROC LP is making in solving the problem. This is discussed in some detail in Example 5.8.

Output 5.6.1 The Oil Blending Problem with a Special Ordered Set

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.00
Variables	Number
Non-negative	5
Upper Bounded	3
Total	8
Constraints	Number
EQ	5
Objective	1
Total	6

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
1	0	ACTIVE	1544	arabian_light	110	0	2	.
2	-1	SUBOPTIMAL	1276	.	.	.	1	268
3	1	FATHOMED	268	.	.	.	0	.

Output 5.6.1 *continued*

Solution Summary	
Integer Optimal Solution	
Objective Value	1276
Phase 1 Iterations	0
Phase 2 Iterations	5
Phase 3 Iterations	0
Integer Iterations	3
Integer Solutions	1
Initial Basic Feasible Variables	5
Time Used (seconds)	0
Number of Inversions	5
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	arabian_heavy		UPPERBD	-165	0	-21.45
2	arabian_light	UPPBD	UPPERBD	-175	110	11.6
3	brega		UPPERBD	-205	0	3.35
4	heating_oil	BASIC	NON-NEG	0	42.9	0
5	jet_1	BASIC	NON-NEG	300	33.33	0
6	jet_2	BASIC	NON-NEG	300	35.09	0
7	naphtha_inter	BASIC	NON-NEG	0	11	0
8	naphtha_light	BASIC	NON-NEG	0	3.85	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	1276	.
2	napha_l_conv	EQ	.	0	0	-60
3	napha_i_conv	EQ	.	0	0	-90
4	heating_oil_conv	EQ	.	0	0	-450
5	recipe_1	EQ	.	0	0	-300
6	recipe_2	EQ	.	0	0	-300

The solution shows that only the ARABIAN_LIGHT crude is purchased. The requirement that only one crude be used in the production is met, and the profit is 1276. This tells you that the value of purchasing crude from an additional source, namely BREGA, is worth $1544 - 1276 = 268$.

Example 5.7: Goal-Programming a Product Mix Problem

This example shows how to use PROC LP to solve a linear goal-programming problem. PROC LP has the ability to solve a series of linear programs, each with a new objective function. These objective functions are ordered by priority. The first step is to solve a linear program with the highest priority objective function constrained only by the formal constraints in the model. Then, the problem with the next highest priority objective function is solved, constrained by the formal constraints in the model and by the value that the highest priority objective function realized. That is, the second problem optimizes the second highest priority objective function among the alternate optimal solutions to the first optimization problem. The process continues until a linear program is solved for each of the objectives.

This technique is useful for differentiating among alternate optimal solutions to a linear program. It also fits into the formal paradigm presented in goal programming. In goal programming, the objective functions typically take on the role of driving a linear function of the structural variables to meet a target level as closely as possible. The details of this can be found in many books on the subject, including Ignizio (1976).

Consider the following problem taken from Ignizio (1976). A small paint company manufactures two types of paint, latex and enamel. In production, the company uses 10 hours of labor to produce 100 gallons of latex and 15 hours of labor to produce 100 gallons of enamel. Without hiring outside help or requiring overtime, the company has 40 hours of labor available each week. Furthermore, each paint generates a profit at the rate of $1.00 per gallon. The company has the following objectives listed in decreasing priority:

- avoid the use of overtime
- achieve a weekly profit of $1000
- produce at least 700 gallons of enamel paint each week

The program to solve this problem follows.

```
data object;
   input _row_ $ latex enamel n1 n2 n3 p1 p2 p3 _type_ $ _rhs_;
   datalines;
overtime   .   .  .  .  .  1  .  . min   1
profit     .   .  .  1  .  .  .  . min   2
enamel     .   .  .  .  1  .  .  . min   3
overtime  10  15  1  .  . -1  .  . eq   40
profit   100 100  .  1  .  . -1  . eq 1000
enamel     .   1  .  .  1  .  . -1 eq    7
;

proc lp data=object goalprogram;
run;
```

The data set called OBJECT contains the model. Its first three observations are the objective rows, and the next three observations are the constraints. The values in the right-hand-side variable _RHS_ in the objective rows give the priority of the objectives. The objective in the first observation with _ROW_='OVERTIME' has the highest priority, the objective named PROFIT has the next highest, and the objective named ENAMEL has the lowest. Note that the value of the right-hand-side variable determines the priority, not the order, in the data set.

Because this example is set in the formal goal-programming scheme, the model has structural variables representing negative (n1, n2, and n3) and positive (p1, p2, and p3) deviations from target levels. For example, n1+p1 is the deviation from the objective of avoiding the use of overtime and underusing the normal work time, namely using exactly 40 work hours. The other objectives are handled similarly.

Notice that the PROC LP statement includes the GOALPROGRAM option. Without this option, the procedure would solve three separate problems: one for each of the three objective functions. In that case, however, the procedure would not constrain the second and third programs using the results of the preceding programs; also, the values 1, 2, and 3 for _RHS_ in the objective rows would have no effect.

Output 5.7.1 shows the solution of the goal program, apparently as three linear program outputs. However, examination of the constraint summaries in the second and third problems shows that the constraints labeled by the objectives OVERTIME and PROFIT have type FIXEDOBJ. This indicates that these objective rows have become constraints in the subsequent problems.

Output 5.7.1 Goal Programming

The LP Procedure

Problem Summary	
Objective Function	Min overtime
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.83
Variables	Number
Non-negative	8
Total	8
Constraints	Number
EQ	3
Objective	3
Total	6

Output 5.7.1 *continued*

Solution Summary	
Terminated Successfully	
Objective Value	0
Phase 1 Iterations	2
Phase 2 Iterations	0
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	7
Time Used (seconds)	0
Number of Inversions	2
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	latex	ALTER	NON-NEG	0	0	0
2	enamel	ALTER	NON-NEG	0	0	0
3	n1	BASIC	NON-NEG	0	40	0
4	n2	BASIC	NON-NEG	0	1000	0
5	n3	BASIC	NON-NEG	0	7	0
6	p1		NON-NEG	1	0	1
7	p2	ALTER	NON-NEG	0	0	0
8	p3	ALTER	NON-NEG	0	0	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	overtime	OBJECTVE	.	0	0	.
2	profit	FREE_OBJ	.	0	1000	.
3	enamel	FREE_OBJ	.	0	7	.
4	overtime	EQ	.	40	40	0
5	profit	EQ	.	1000	1000	0
6	enamel	EQ	.	7	7	0

Output 5.7.1 *continued*

Problem Summary	
Objective Function	Min profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.83
Variables	Number
Non-negative	8
Total	8
Constraints	Number
EQ	3
Objective	3
Total	6

Solution Summary Terminated Successfully	
Objective Value	600
Phase 1 Iterations	0
Phase 2 Iterations	3
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	7
Time Used (seconds)	0
Number of Inversions	5
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Output 5.7.1 *continued*

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	latex	BASIC	NON-NEG	0	4	0
2	enamel		NON-NEG	0	0	50
3	n1		NON-NEG	0	0	10
4	n2	BASIC	NON-NEG	1	600	0
5	n3	BASIC	NON-NEG	0	7	0
6	p1	DEGEN	NON-NEG	0	0	0
7	p2		NON-NEG	0	0	1
8	p3	ALTER	NON-NEG	0	0	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	overtime	FIXEDOBJ	.	0	0	.
2	profit	OBJECTVE	.	0	600	.
3	enamel	FREE_OBJ	.	0	7	.
4	overtime	EQ	.	40	40	-10
5	profit	EQ	.	1000	1000	1
6	enamel	EQ	.	7	7	0

Problem Summary

Objective Function	Min enamel
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	45.83
Variables	Number
Non-negative	8
Total	8
Constraints	Number
EQ	3
Objective	3
Total	6

Output 5.7.1 *continued*

Solution Summary	
Terminated Successfully	
Objective Value	7
Phase 1 Iterations	0
Phase 2 Iterations	1
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	7
Time Used (seconds)	0
Number of Inversions	8
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	latex	BASIC	NON-NEG	0	4	0
2	enamel	DEGEN	NON-NEG	0	0	0
3	n1		NON-NEG	0	0	0.2
4	n2	BASIC	NON-NEG	0	600	0
5	n3	BASIC	NON-NEG	1	7	0
6	p1	DEGEN	NON-NEG	0	0	0
7	p2		NON-NEG	0	0	0.02
8	p3		NON-NEG	0	0	1

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	overtime	FIXEDOBJ	.	0	0	.
2	profit	FIXEDOBJ	.	0	600	.
3	enamel	OBJECTVE	.	0	7	.
4	overtime	EQ	.	40	40	-0.2
5	profit	EQ	.	1000	1000	0.02
6	enamel	EQ	.	7	7	1

The solution to the last linear program shows a value of 4 for the variable LATEX and a value of 0 for the variable ENAMEL. This tells you that the solution to the linear goal program is to produce 400 gallons of latex and no enamel paint.

The values of the objective functions in the three linear programs tell you whether you can achieve the three objectives. The activities of the constraints labeled OVERTIME, PROFIT, and ENAMEL tell you values of

the three linear program objectives. Because the first linear programming objective OVERTIME is 0, the highest priority objective, which is to avoid using additional labor, is accomplished. However, because the second and third objectives are nonzero, the second and third priority objectives are not satisfied completely. The PROFIT objective is 600. Because the PROFIT objective is to minimize the negative deviation from the profit constraint, this means that a profit of only 400 = 1000 − 600 is realized. Similarly, the ENAMEL objective is 7, indicating that there is a negative deviation from the ENAMEL target of 7 units.

Example 5.8: A Simple Integer Program

Recall the linear programming problem presented in Chapter 3, "Introduction to Optimization" (*SAS/OR User's Guide: Mathematical Programming*). In that problem, a firm produces two products, chocolates and gumdrops, that are processed by four processes: cooking, color/flavor, condiments, and packaging. The objective is to determine the product mix that maximizes the profit to the firm while not exceeding manufacturing capacities. The problem is extended to demonstrate a use of integer-constrained variables.

Suppose that you must manufacture only one of the two products. In addition, there is a setup cost of 100 if you make the chocolates and 75 if you make the gumdrops. To identify which product will maximize profit, you define two zero-one integer variables, ICHOCO and IGUMDR, and you also define two new constraints, CHOCOLATE and GUM. The constraint labeled CHOCOLATE forces ICHOCO to equal one when chocolates are manufactured. Similarly, the constraint labeled GUM forces IGUMDR to equal 1 when gumdrops are manufactured. Also, you should include a constraint labeled ONLY_ONE that requires the sum of ICHOCO and IGUMDR to equal 1. (Note that this could be accomplished more simply by including ICHOCO and IGUMDR in a SOSEQ set.) Since ICHOCO and IGUMDR are integer variables, this constraint eliminates the possibility of both products being manufactured. Notice the coefficients -10000, which are used to force ICHOCO or IGUMDR to 1 whenever CHOCO and GUMDR are nonzero. This technique, which is often used in integer programming, can cause severe numerical problems. If this driving coefficient is too large, then arithmetic overflows and underflow may result. If the driving coefficient is too small, then the integer variable may not be driven to 1 as desired by the modeler.

The objective coefficients of the integer variables ICHOCO and IGUMDR are the negatives of the setup costs for the two products. The following is the data set that describes this problem and the call to PROC LP to solve it:

```
data;
   format _row_ $10. ;
   input _row_ $ choco gumdr ichoco igumdr _type_ $ _rhs_;
   datalines;
object         .25     .75   -100    -75 max         .
cooking         15      40      0      0 le      27000
color            0   56.25      0      0 le      27000
package      18.75       0      0      0 le      27000
condiments      12      50      0      0 le      27000
chocolate        1       0 -10000      0 le          0
gum              0       1      0 -10000 le          0
only_one         0       0      1      1 eq          1
binary           .       .      1      2 binary      .
;
```

```
proc lp;
run;
```

The solution shows that gumdrops are produced. See Output 5.8.1.

Output 5.8.1 Summaries and an Integer Programming Iteration Log

The LP Procedure

Problem Summary	
Objective Function	Max object
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	25.71
Variables	Number
Non-negative	2
Binary	2
Slack	6
Total	10
Constraints	Number
LE	6
EQ	1
Objective	1
Total	8

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
1	0	ACTIVE	397.5	ichoco	0.1	0.2	2	.
2	-1	SUBOPTIMAL	260	.	.	.	1	70
3	1	SUBOPTIMAL	285	.	.	.	0	.

Output 5.8.1 *continued*

Solution Summary	
Integer Optimal Solution	
Objective Value	285
Phase 1 Iterations	0
Phase 2 Iterations	5
Phase 3 Iterations	5
Integer Iterations	3
Integer Solutions	2
Initial Basic Feasible Variables	9
Time Used (seconds)	0
Number of Inversions	5
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	choco	DEGEN	NON-NEG	0.25	0	0
2	gumdr	BASIC	NON-NEG	0.75	480	0
3	ichoco		BINARY	-100	0	2475
4	igumdr	BASIC	BINARY	-75	1	0
5	cooking	BASIC	SLACK	0	7800	0
6	color		SLACK	0	0	-0.013333
7	package	BASIC	SLACK	0	27000	0
8	condiments	BASIC	SLACK	0	3000	0
9	chocolate		SLACK	0	0	-0.25
10	gum	BASIC	SLACK	0	9520	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	object	OBJECTVE	.	0	285	.
2	cooking	LE	5	27000	19200	0
3	color	LE	6	27000	27000	0.0133333
4	package	LE	7	27000	0	0
5	condiments	LE	8	27000	24000	0
6	chocolate	LE	9	0	0	0.25
7	gum	LE	10	0	-9520	0
8	only_one	EQ	.	1	1	-75

The branch-and-bound tree can be reconstructed from the information contained in the integer iteration log. The column labeled Iter numbers the integer iterations. The column labeled Problem identifies the Iter number of the parent problem from which the current problem is defined. For example, Iter=2 has Problem=-1. This means that problem 2 is a direct descendant of problem 1. Furthermore, because problem 1 branched on ICHOCO, you know that problem 2 is identical to problem 1 with an additional constraint on variable ICHOCO. The minus sign in the Problem=-1 in Iter=2 tells you that the new constraint on variable ICHOCO is a lower bound. Moreover, because Value=0.1 in Iter=1, you know that ICHOCO=0.1 in Iter=1 so that the added constraint in Iter=2 is ICHOCO $\geq \lceil 0.1 \rceil$. In this way, the information in the log can be used to reconstruct the branch-and-bound tree. In fact, when you save an ACTIVEOUT= data set, it contains information in this format that is used to reconstruct the tree when you restart a problem using the ACTIVEIN= data set. See Example 5.10.

Note that if you defined a SOSEQ special ordered set containing the variables CHOCO and GUMDR, the integer variables ICHOCO and IGUMDR and the three associated constraints would not have been needed.

Example 5.9: An Infeasible Problem

This is an example of the Infeasible Information Summary that is displayed when an infeasible problem is encountered. Consider the following problem:

$$\begin{array}{ll} \max & x + y + z + w \\ \text{subject to} & x + 3y + 2z + 4w \leq 5 \\ & 3x + y + 2z + w \leq 4 \\ & 5x + 3y + 3z + 3w = 9 \\ & x,\ y,\ z,\ w \geq 0 \end{array}$$

Examination of this problem reveals that it is unsolvable. Consequently, PROC LP identifies it as infeasible. The following program attempts to solve it.

```
data infeas;
   format _id_ $6.;
   input _id_ $ x1-x4 _type_ $ _rhs_;
   datalines;
profit   1    1    1    1    max   .
const1   1    3    2    4    le    5
const2   3    1    2    1    le    4
const3   5    3    3    3    eq    9
;
```

The results are shown in Output 5.9.1.

Output 5.9.1 The Solution of an Infeasible Problem

The LP Procedure

Problem Summary	
Objective Function	Max profit
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	77.78
Variables	Number
Non-negative	4
Slack	2
Total	6
Constraints	Number
LE	2
EQ	1
Objective	1
Total	4

```
ERROR: Infeasible problem. Note the constraints in the constraint summary
       that are identified as infeasible. If none of the constraints are
       flagged then check the implicit bounds on the variables.
```

The LP Procedure

Solution Summary	
Infeasible Problem	
Objective Value	2.5
Phase 1 Iterations	2
Phase 2 Iterations	0
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	5
Time Used (seconds)	0
Number of Inversions	2
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

The LP Procedure

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	x1	BASIC	NON-NEG	1	0.75	0
2	x2	BASIC	NON-NEG	1	1.75	0
3	x3		NON-NEG	1	0	0.5
4	x4		NON-NEG	1	0	0
INF	const1	BASIC	SLACK	0	-1	0
6	const2		SLACK	0	0	0.5

The LP Procedure

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	profit	OBJECTVE	.	0	2.5	.
INF	const1	LE	5	5	6	0
3	const2	LE	6	4	4	-0.5
4	const3	EQ	.	9	9	0.5

The LP Procedure

Infeasible Information Summary	
Infeasible Row	const1
Constraint Activity	6
Row Type	LE
Rhs Data	5

Variable	Coefficient	Activity	Lower Bound	Upper Bound
x1	1	0.75	0	INFINITY
x2	3	1.75	0	INFINITY
x3	2	0	0	INFINITY
x4	4	0	0	INFINITY

Note the information given in the Infeasible Information Summary for the infeasible row CONST1. It shows that the inequality row CONST1 with right-hand side 5 was found to be infeasible with activity 6. The summary also shows each variable that has a nonzero coefficient in that row and its activity level at the infeasibility. Examination of these model parameters might give you a clue as to the cause of infeasibility, such as an incorrectly entered coefficient or right-hand-side value.

Example 5.10: Restarting an Integer Program

The following example is attributed to Haldi (Garfinkel and Nemhauser 1972) and is used in the literature as a test problem. Notice that the ACTIVEOUT= and the PRIMALOUT= options are used when invoking PROC LP. These cause the LP procedure to save the primal solution in the data set named P and the active tree in the data set named A. If the procedure fails to find an optimal integer solution on the initial call, it can be called later using the A and P data sets as starting information.

```
data haldi10;
   input x1-x12 _type_ $ _rhs_;
   datalines;
  0   0   0   0   0   0   1   1   1   1   1   1  MAX   .
  9   7  16   8  24   5   3   7   8   4   6   5  LE  110
 12   6   6   2  20   8   4   6   3   1   5   8  LE   95
 15   5  12   4   4   5   5   5   6   2   1   5  LE   80
 18   4   4  18  28   1   6   4   2   9   7   1  LE  100
-12   0   0   0   0   0   1   0   0   0   0   0  LE    0
  0 -15   0   0   0   0   0   1   0   0   0   0  LE    0
  0   0 -12   0   0   0   0   0   1   0   0   0  LE    0
  0   0   0 -10   0   0   0   0   0   1   0   0  LE    0
  0   0   0   0 -11   0   0   0   0   0   1   0  LE    0
  0   0   0   0   0 -11   0   0   0   0   0   1  LE    0
  1   1   1   1   1   1 1000 1000 1000 1000 1000 1000 UPPERBD .
  1   2   3   4   5   6   7   8   9  10  11  12  INTEGER .
  ;
```

The ACTIVEOUT= data set contains a representation of the current active problems in the branch-and-bound tree. The PRIMALOUT= data set contains a representation of the solution to the current problem. These two can be used to restore the procedure to an equivalent state to the one it was in when it stopped.

The results from the call to PROC LP is shown in Output 5.10.1. Notice that the procedure performed 100 iterations and then terminated on maximum integer iterations. This is because, by default, IMAXIT=100. The procedure reports the current best integer solution.

Output 5.10.1 Output from the HALDI10 Problem

The LP Procedure

Problem Summary	
Objective Function	Max _OBS1_
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	31.82
Variables	Number
Integer	6
Binary	6
Slack	10
Total	22
Constraints	Number
LE	10
Objective	1
Total	11

The LP Procedure

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
1	0	ACTIVE	18.709524	x9	1.543	1.11905	2	.
2	1	ACTIVE	18.467723	x12	9.371	0.88948	3	.
3	2	ACTIVE	18.460133	x8	0.539	1.04883	4	.
4	-3	ACTIVE	18.453638	x12	8.683	1.12993	5	.
5	4	ACTIVE	18.439678	x10	7.448	1.20125	6	.
6	5	ACTIVE	18.403728	x6	0.645	1.3643	7	.
7	-6	ACTIVE	18.048289	x4	0.7	1.18395	8	.
8	-7	ACTIVE	17.679087	x8	1.833	0.52644	9	.
9	8	ACTIVE	17.52	x10	6.667	0.70111	10	.
10	9	ACTIVE	17.190085	x12	7.551	1.37615	11	.
11	-10	ACTIVE	17.02	x1	0.085	0.255	12	.
12	11	ACTIVE	16.748	x11	0.748	0.47	13	.
13	-12	ACTIVE	16.509091	x9	0.509	0.69091	14	.
14	13	ACTIVE	16.261333	x11	1.261	0.44267	15	.
15	14	ACTIVE	16	x3	0.297	0.45455	16	.
16	15	ACTIVE	16	x5	0.091	0.15758	16	.
17	-16	INFEASIBLE	-0.4	.	.	.	15	.
18	-15	ACTIVE	11.781818	x10	1.782	0.37576	15	.
19	18	ACTIVE	11	x5	0.091	0.15758	15	.
20	-19	INFEASIBLE	-6.4	.	.	.	14	.
21	-14	ACTIVE	11.963636	x5	0.182	0.28485	14	.
22	-21	INFEASIBLE	-4.4	.	.	.	13	.
23	-13	ACTIVE	15.281818	x10	4.282	0.52273	13	.
24	23	ACTIVE	15.041333	x5	0.095	0.286	14	.
25	-24	INFEASIBLE	-2.9	.	.	.	13	.
26	24	INFEASIBLE	14	.	.	.	12	.
27	12	ACTIVE	16	x3	0.083	0.15	13	.
28	-27	ACTIVE	15.277778	x9	0.278	0.34444	14	.
29	-28	ACTIVE	13.833333	x10	3.833	0.23333	14	.
30	29	ACTIVE	13	x2	0.4	0.4	15	.
31	30	INFEASIBLE	12	.	.	.	14	.
32	-30	SUBOPTIMAL	10	.	.	.	13	8
33	28	ACTIVE	15	x2	0.067	0.06667	13	8
34	-33	SUBOPTIMAL	12	.	.	.	12	6
35	27	ACTIVE	15	x2	0.067	0.06667	12	6
36	-35	SUBOPTIMAL	15	.	.	.	11	3
37	-11	FATHOMED	14.275	.	.	.	10	3
38	10	ACTIVE	16.804848	x1	0.158	0.50313	11	3
39	-38	FATHOMED	14.784	.	.	.	10	3
40	38	ACTIVE	16.40381	x11	1.404	0.68143	11	3
41	-40	ACTIVE	16.367677	x10	5.368	0.69949	12	3
42	41	ACTIVE	16.113203	x11	2.374	1.00059	12	3
43	42	ACTIVE	16	x5	0.182	0.33182	12	3
44	-43	FATHOMED	13.822222	.	.	.	11	3
45	-41	FATHOMED	12.642424	.	.	.	10	3
46	40	ACTIVE	16	x5	0.229	0.37857	10	3

The LP Procedure

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
47	46	FATHOMED	15	.	.	.	9	3
48	-9	ACTIVE	17.453333	x7	0.453	0.64111	10	3
49	48	ACTIVE	17.35619	x11	0.356	0.53857	11	3
50	49	ACTIVE	17	x5	0.121	0.27143	12	3
51	50	ACTIVE	17	x3	0.083	0.15	13	3
52	-51	FATHOMED	15.933333	.	.	.	12	3
53	51	ACTIVE	16	x2	0.067	0.06667	12	3
54	-53	SUBOPTIMAL	16	.	.	.	8	2
55	-8	ACTIVE	17.655399	x12	7.721	0.56127	9	2
56	55	ACTIVE	17.519375	x10	6.56	0.76125	10	2
57	56	ACTIVE	17.256874	x2	0.265	0.67388	11	2
58	57	INFEASIBLE	17.167622	.	.	.	10	2
59	-57	FATHOMED	16.521755	.	.	.	9	2
60	-56	FATHOMED	17.03125	.	.	.	8	2
61	-55	ACTIVE	17.342857	x9	0.343	0.50476	8	2
62	61	ACTIVE	17.2225	x7	0.16	0.37333	9	2
63	62	ACTIVE	17.1875	x8	2.188	0.33333	9	2
64	63	ACTIVE	17.153651	x11	0.154	0.30095	10	2
65	-64	FATHOMED	12.381818	.	.	.	9	2
66	64	ACTIVE	17	x2	0.133	0.18571	9	2
67	-66	FATHOMED	13	.	.	.	8	2
68	-62	FATHOMED	14.2	.	.	.	7	2
69	7	FATHOMED	15.428583	.	.	.	6	2
70	6	FATHOMED	16.75599	.	.	.	5	2
71	-5	ACTIVE	17.25974	x6	0.727	0.82078	5	2
72	-71	FATHOMED	17.142857	.	.	.	4	2
73	-4	ACTIVE	18.078095	x4	0.792	0.70511	5	2
74	-73	ACTIVE	17.662338	x10	7.505	0.91299	5	2
75	74	ACTIVE	17.301299	x9	0.301	0.57489	5	2
76	75	ACTIVE	17.210909	x7	0.211	0.47697	5	2
77	76	FATHOMED	17.164773	.	.	.	4	2
78	73	FATHOMED	12.872727	.	.	.	3	2
79	3	ACTIVE	18.368316	x10	7.602	1.20052	4	2
80	79	ACTIVE	18.198323	x7	1.506	1.85351	5	2
81	80	ACTIVE	18.069847	x12	8.517	1.67277	6	2
82	-81	ACTIVE	17.910909	x4	0.7	0.73015	7	2
83	-82	ACTIVE	17.790909	x7	0.791	0.54015	8	2
84	-83	ACTIVE	17.701299	x9	0.701	0.62229	8	2
85	84	ACTIVE	17.17619	x6	0.818	0.45736	8	2
86	-85	ACTIVE	17.146667	x11	0.147	0.24333	8	2
87	86	ACTIVE	17	x1	0.167	0.16667	8	2
88	87	INFEASIBLE	16	.	.	.	7	2
89	83	ACTIVE	17.58	x11	0.58	0.73788	8	2
90	-89	FATHOMED	17.114286	.	.	.	7	2
91	-80	ACTIVE	18.044048	x12	8.542	1.71158	8	2
92	91	ACTIVE	17.954536	x11	0.477	1.90457	9	2

The LP Procedure

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
93	92	ACTIVE	17.875084	x4	0.678	1.16624	10	2
94	93	FATHOMED	13.818182	.	.	.	9	2
95	-93	ACTIVE	17.231221	x6	0.727	0.76182	9	2
96	-95	FATHOMED	17.085714	.	.	.	8	2
97	-92	FATHOMED	17.723058	.	.	.	7	2
98	-91	FATHOMED	16.378788	.	.	.	6	2
99	89	ACTIVE	17	x6	0.818	0.26515	6	2
100	-99	ACTIVE	17	x3	0.083	0.08333	6	2

WARNING: The maximum number of integer iterations has been exceeded. Increase this limit with the 'IMAXIT=' option on the RESET statement.

The LP Procedure

Solution Summary

Terminated on Maximum Integer Iterations	
Integer Feasible Solution	
Objective Value	16
Phase 1 Iterations	0
Phase 2 Iterations	13
Phase 3 Iterations	161
Integer Iterations	100
Integer Solutions	4
Initial Basic Feasible Variables	12
Time Used (seconds)	0
Number of Inversions	37
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

The LP Procedure

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	x1	DEGEN	BINARY	0	0	0
2	x2	ALTER	BINARY	0	1	0
3	x3		BINARY	0	0	12
4	x4	ALTER	BINARY	0	1	0
5	x5	ALTER	BINARY	0	0	0
6	x6	ALTER	BINARY	0	1	0
7	x7		INTEGER	1	0	1
8	x8		INTEGER	1	1	1
9	x9	DEGEN	INTEGER	1	0	0
10	x10		INTEGER	1	7	1
11	x11		INTEGER	1	0	1
12	x12		INTEGER	1	8	1
13	_OBS2_	BASIC	SLACK	0	15	0
14	_OBS3_	BASIC	SLACK	0	2	0
15	_OBS4_	BASIC	SLACK	0	7	0
16	_OBS5_	BASIC	SLACK	0	2	0
17	_OBS6_	ALTER	SLACK	0	0	0
18	_OBS7_	BASIC	SLACK	0	14	0
19	_OBS8_		SLACK	0	0	-1
20	_OBS9_	BASIC	SLACK	0	3	0
21	_OBS10_	DEGEN	SLACK	0	0	0
22	_OBS11_	BASIC	SLACK	0	3	0

The LP Procedure

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	_OBS1_	OBJECTVE	.	0	16	.
2	_OBS2_	LE	13	110	95	0
3	_OBS3_	LE	14	95	93	0
4	_OBS4_	LE	15	80	73	0
5	_OBS5_	LE	16	100	98	0
6	_OBS6_	LE	17	0	0	0
7	_OBS7_	LE	18	0	-14	0
8	_OBS8_	LE	19	0	0	1
9	_OBS9_	LE	20	0	-3	0
10	_OBS10_	LE	21	0	0	0
11	_OBS11_	LE	22	0	-3	0

To continue with the solution of this problem, invoke PROC LP with the ACTIVEIN= and PRIMALIN= options and reset the IMAXIT= option. This restores the branch-and-bound tree and simplifies calculating a basic feasible solution from which to start processing.

```
proc lp data=haldi10 activein=a primalin=p imaxit=250;
run;
```

The procedure picks up iterating from a equivalent state to where it left off. The problem will still not be solved when IMAXIT=250 occurs.

Example 5.11: Alternative Search of the Branch-and-Bound Tree

In this example, the HALDI10 problem from Example 5.10 is solved. However, here the default strategy for searching the branch-and-bound tree is modified. By default, the search strategy has VARSELECT=FAR. This means that when searching for an integer variable on which to branch, the procedure uses the one that has a value farthest from an integer value. An alternative strategy has VARSELECT=PENALTY. This strategy causes PROC LP to look at the cost, in terms of the objective function, of branching on an integer variable. The procedure looks at PENALTYDEPTH= integer variables before choosing the one with the largest cost. This is a much more expensive strategy (in terms of execution time) than the VARSELECT=FAR strategy, but it can be beneficial if fewer integer iterations must be done to find an optimal solution.

```
proc lp data=haldi10 varselect=penalty;
run;
```

Compare the number of integer iterations needed to solve the problem using this heuristic with the default strategy used in Example 5.10. In this example, the difference is profound; in general, solution times can vary significantly with the search technique. See Output 5.11.1.

Output 5.11.1 Summaries and an Integer Programming Iteration Log: Using VARSELECT=PENALTY

The LP Procedure

Problem Summary	
Objective Function	Max _OBS1_
Rhs Variable	_rhs_
Type Variable	_type_
Problem Density (%)	31.82
Variables	Number
Integer	6
Binary	6
Slack	10
Total	22
Constraints	Number
LE	10
Objective	1
Total	11

Output 5.11.1 *continued*

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
1	0	ACTIVE	18.709524	x4	0.8	1.11905	2	.
2	1	ACTIVE	16.585187	x1	0.447	2.33824	3	.
3	2	ACTIVE	14.86157	x5	0.221	2.09584	4	.
4	3	ACTIVE	14.807195	x2	0.897	1.31729	5	.
5	-4	ACTIVE	14.753205	x8	14.58	0.61538	6	.
6	5	ACTIVE	14.730078	x6	0.043	0.79446	7	.
7	-6	ACTIVE	13.755102	x3	0.051	0.58163	8	.
8	-7	ACTIVE	11.6	x8	11.6	0.4	9	.
9	8	ACTIVE	11.6	x12	0.6	0.4	10	.
10	-9	ACTIVE	11.6	x8	10.6	0.4	11	.
11	10	ACTIVE	11.6	x12	1.6	0.4	12	.
12	-11	ACTIVE	11.6	x8	9.6	0.4	13	.
13	12	ACTIVE	11.6	x12	2.6	0.4	14	.
14	-13	ACTIVE	11.571429	x9	0.143	0.57143	15	.
15	14	ACTIVE	11.5	x8	8.5	0.5	16	.
16	-15	INFEASIBLE	9	.	.	.	15	.
17	15	ACTIVE	11.375	x12	3.375	0.375	16	.
18	-17	ACTIVE	11.166667	x8	7.167	0.16667	17	.
19	18	ACTIVE	11.125	x12	4.125	0.125	18	.
20	19	SUBOPTIMAL	11	.	.	.	7	7
21	7	ACTIVE	13.5	x8	13.5	0.5	8	7
22	-21	INFEASIBLE	11	.	.	.	7	7
23	21	ACTIVE	13.375	x12	0.375	0.375	8	7
24	-23	ACTIVE	13.166667	x8	12.17	0.16667	9	7
25	24	ACTIVE	13.125	x12	1.125	0.125	10	7
26	25	SUBOPTIMAL	13	.	.	.	4	5
27	6	ACTIVE	14.535714	x3	0.045	0.50893	5	5
28	-27	FATHOMED	12.625	.	.	.	4	5
29	27	SUBOPTIMAL	14	.	.	.	1	4
30	-1	ACTIVE	18.309524	x3	0.129	1.31905	2	4
31	30	ACTIVE	17.67723	x6	0.886	0.43662	3	4
32	31	ACTIVE	15.485156	x2	0.777	1.50833	4	4
33	-32	ACTIVE	15.2625	x1	0.121	1.38333	4	4
34	33	ACTIVE	15.085106	x10	3.532	0.91489	4	4
35	34	FATHOMED	14.857143	.	.	.	3	4
36	32	FATHOMED	11.212121	.	.	.	2	4
37	-31	ACTIVE	17.56338	x10	7.93	0.43662	3	4
38	37	ACTIVE	17.225962	x8	2.38	0.69231	4	4
39	38	ACTIVE	17.221818	x1	0.016	0.37111	5	4
40	-39	FATHOMED	14.43662	.	.	.	4	4
41	39	ACTIVE	17.172375	x2	0.133	0.31948	5	4
42	41	ACTIVE	16.890196	x5	0.086	0.19608	6	4
43	42	ACTIVE	16.75	x12	9.75	0.25	7	4
44	-43	SUBOPTIMAL	15	.	.	.	6	3
45	43	SUBOPTIMAL	16	.	.	.	3	2
46	-38	FATHOMED	17.138028	.	.	.	2	2

Output 5.11.1 *continued*

Integer Iteration Log

Iter	Problem	Condition	Objective	Branched	Value	Sinfeas	Active	Proximity
47	-37	SUBOPTIMAL	17	.	.	.	1	1
48	-30	FATHOMED	16.566667	.	.	.	0	.

Solution Summary

Integer Optimal Solution

Objective Value	17
Phase 1 Iterations	0
Phase 2 Iterations	13
Phase 3 Iterations	79
Integer Iterations	48
Integer Solutions	6
Initial Basic Feasible Variables	12
Time Used (seconds)	0
Number of Inversions	17
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Output 5.11.1 *continued*

Variable Summary

Col	Variable Name	Status	Type	Price	Activity	Reduced Cost
1	x1	DEGEN	BINARY	0	0	0
2	x2		BINARY	0	0	-4
3	x3		BINARY	0	0	-4
4	x4		BINARY	0	1	-18
5	x5	DEGEN	BINARY	0	0	0
6	x6		BINARY	0	1	-1
7	x7		INTEGER	1	0	-6.5
8	x8		INTEGER	1	0	-3
9	x9		INTEGER	1	0	-1
10	x10		INTEGER	1	8	-8
11	x11		INTEGER	1	0	-8.545455
12	x12	BASIC	INTEGER	1	9	0
13	_OBS2_	BASIC	SLACK	0	20	0
14	_OBS3_	BASIC	SLACK	0	5	0
15	_OBS4_	BASIC	SLACK	0	10	0
16	_OBS5_		SLACK	0	0	-1
17	_OBS6_		SLACK	0	0	-1.5
18	_OBS7_	DEGEN	SLACK	0	0	0
19	_OBS8_	DEGEN	SLACK	0	0	0
20	_OBS9_	BASIC	SLACK	0	2	0
21	_OBS10_		SLACK	0	0	-2.545455
22	_OBS11_	BASIC	SLACK	0	2	0

Constraint Summary

Row	Constraint Name	Type	S/S Col	Rhs	Activity	Dual Activity
1	_OBS1_	OBJECTVE	.	0	17	.
2	_OBS2_	LE	13	110	90	0
3	_OBS3_	LE	14	95	90	0
4	_OBS4_	LE	15	80	70	0
5	_OBS5_	LE	16	100	100	1
6	_OBS6_	LE	17	0	0	1.5
7	_OBS7_	LE	18	0	0	0
8	_OBS8_	LE	19	0	0	0
9	_OBS9_	LE	20	0	-2	0
10	_OBS10_	LE	21	0	0	2.5454545
11	_OBS11_	LE	22	0	-2	0

Although the VARSELECT=PENALTY strategy works well in this example, there is no guarantee that it will work well with your model. Experimentation with various strategies is necessary to find the one that works well with your model and data, particularly if a model is solved repeatedly with few changes to either the structure or the data.

Example 5.12: An Assignment Problem

This example departs somewhat from the emphasis of previous ones. Typically, linear programming models are large, have considerable structure, and are solved with some regularity. Some form of automatic model building, or matrix generation as it is commonly called, is a useful aid. The sparse input format provides a great deal of flexibility in model specification so that, in many cases, the DATA step can be used to generate the matrix.

The following assignment problem illustrates some techniques in matrix generation. In this example, you have four machines that can produce any of six grades of cloth, and you have five customers that demand various amounts of each grade of cloth. The return from supplying a customer with a demanded grade depends on the machine on which the cloth was made. In addition, the machine capacity depends both upon the specific machine used and the grade of cloth made.

To formulate this problem, let i denote customer, j denote grade, and k denote machine. Then let x_{ijk} denote the amount of cloth of grade j made on machine k for customer i; let r_{ijk} denote the return from selling one unit of grade j cloth made on machine k to customer i; let d_{ij} denote the demand for grade j cloth by customer i; let c_{jk} denote the number of units of machine k required to produce one unit of grade j cloth; and let a_k denote the number of units of machine k available. Then, you get

$$\begin{array}{lll} \max & \sum_{ijk} r_{ijk} x_{ijk} & \\ \text{subject to} & \sum_{k} x_{ijk} = d_{ij} & \text{for all } i \text{ and } j \\ & \sum_{ij} c_{jk} x_{ijk} \le a_k & \text{for all } k \\ & x_{ijk} \ge 0 & \text{for all } i, j \text{ and } k \end{array}$$

The data are saved in three data sets. The OBJECT data set contains the returns for satisfying demand, the DEMAND data set contains the amounts demanded, and the RESOURCE data set contains the conversion factors for each grade and the total amounts of machine resources available.

```
title 'An Assignment Problem';

data object;
   input machine customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 1 102 140 105  105  125  148
1 2 115 133 118  118  143  166
1 3  70 108  83   83   88   86
1 4  79 117  87   87  107  105
1 5  77 115  90   90  105  148
2 1 123 150 125  124  154   .
2 2 130 157 132  131  166   .
2 3 103 130 115  114  129   .
2 4 101 128 108  107  137   .
2 5 118 145 130  129  154   .
3 1  83  .   .    97  122  147
3 2 119  .   .   133  163  180
3 3  67  .   .    91  101  101
3 4  85  .   .   104  129  129
3 5  90  .   .   114  134  179
4 1 108 121  79   .   112  132
4 2 121 132  92   .   130  150
```

```
4 3  78  91  59   .    77   72
4 4 100 113  76   .   109  104
4 5  96 109  77   .   105  145
;

data demand;
   input customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 100 100 150  150  175  250
2 300 125 300  275  310  325
3 400   0 400  500  340    0
4 250   0 750  750    0    0
5   0 600 300    0  210  360
;

data resource;
   input machine
         grade1 grade2 grade3 grade4 grade5 grade6 avail;
   datalines;
1 .250 .275 .300  .350  .310  .295  744
2 .300 .300 .305  .315  .320  .     244
3 .350 .    .     .320  .315  .300  790
4 .280 .275 .260  .     .250  .295  672
;
```

The linear program is built using the DATA step. The model is saved in a SAS data set in the sparse input format for PROC LP. Each section of the following DATA step generates a piece of the linear program. The first section generates the objective function; the next section generates the demand constraints; and the last section generates the machine resource availability constraints.

```
/*  build the linear programming model */

data model;
   array grade{6} grade1-grade6;
   length _type_ $ 8 _row_ $ 8 _col_ $ 8;
   keep _type_ _row_ _col_ _coef_;

   ncust=5;
   nmach=4;
   ngrade=6;

/* generate the objective function */

_type_='MAX';
_row_='OBJ';
do k=1 to nmach;
   do i=1 to ncust;
      link readobj;     /* read the objective coefficient data */
      do j=1 to ngrade;
         if grade{j}^=. then do;
            _col_='X'||put(i,1.)||put(j,1.)||put(k,1.);
            _coef_=grade{j};
            output;
```

```
         end;
      end;
   end;
end;

/* generate the demand constraints */

do i=1 to ncust;
   link readdmd;        /* read the demand data */
   do j=1 to ngrade;
      if grade{j}^=. then do;
         _type_='EQ';
         _row_='DEMAND'||put(i,1.)||put(j,1.);
         _col_='_RHS_';
         _coef_=grade{j};
         output;
         _type_=' ';
         do k=1 to nmach;
            _col_='X'||put(i,1.)||put(j,1.)||put(k,1.);
            _coef_=1.0;
            output;
         end;
      end;
   end;
end;

/* generate the machine constraints */

do k=1 to nmach;
   link readres;       /* read the machine data */
   _type_='LE';
   _row_='MACHINE'||put(k,1.);
   _col_='_RHS_';
   _coef_=avail;
   output;
   _type_=' ';
   do i=1 to ncust;
      do j=1 to ngrade;
         if grade{j}^=. then do;
            _col_='X'||put(i,1.)||put(j,1.)||put(k,1.);
            _coef_=grade{j};
            output;
            end;
         end;
      end;
   end;

readobj: set object;
return;
readdmd: set demand;
return;
readres: set resource;
return;
run;
```

With the model built and saved in a data set, it is ready for solution using PROC LP. The following program solves the model and saves the solution in the data set called PRIMAL:

```
/* solve the linear program */

proc lp data=model sparsedata noprint primalout=primal;
run;
```

The following output is produced by PROC LP.

Output 5.12.1 An Assignment Problem

An Assignment Problem

The LP Procedure

Problem Summary	
Objective Function	Max OBJ
Rhs Variable	_RHS_
Type Variable	_type_
Problem Density (%)	5.31
Variables	Number
Non-negative	120
Slack	4
Total	124
Constraints	Number
LE	4
EQ	30
Objective	1
Total	35

Output 5.12.1 *continued*

Solution Summary	
Terminated Successfully	
Objective Value	871426.03763
Phase 1 Iterations	0
Phase 2 Iterations	40
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	36
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

The solution is prepared for reporting using the DATA step, and a report is written using PROC TABULATE.

```
   /* report the solution */

   data solution;
      set primal;
      keep customer grade machine amount;
      if substr(_var_,1,1)='X' then do;
        if _value_^=0 then do;
          customer = substr(_var_,2,1);
          grade    = substr(_var_,3,1);
          machine  = substr(_var_,4,1);
          amount   = _value_;
          output;
        end;
     end;
   run;

proc tabulate data=solution;
   class customer grade machine;
   var   amount;
   table (machine*customer), (grade*amount);
run;
```

The report shown in Output 5.12.2 gives the assignment of customer, grade of cloth, and machine that maximizes the return and does not violate the machine resource availability.

Output 5.12.2 An Assignment Problem

An Assignment Problem

		grade					
		1	2	3	4	5	6
		amount	amount	amount	amount	amount	amount
		Sum	Sum	Sum	Sum	Sum	Sum
machine	customer						
1	1	.	100.00	150.00	150.00	175.00	250.00
	2	.	.	300.00	.	.	.
	3	.	.	256.72	210.31	.	.
	4	.	.	750.00	.	.	.
	5	.	92.27	.	.	.	.
2	3	.	.	143.28	.	340.00	.
	5	.	.	300.00	.	.	.
3	2	.	.	.	275.00	310.00	325.00
	3	.	.	.	289.69	.	.
	4	.	.	.	750.00	.	.
	5	.	.	.	.	210.00	360.00
4	1	100.00	.	.	.	.	.
	2	300.00	125.00	.	.	.	.
	3	400.00	.	.	.	.	.
	4	250.00	.	.	.	.	.
	5	.	507.73	.	.	.	.

Example 5.13: A Scheduling Problem

Scheduling is an application area where techniques in model generation can be valuable. Problems involving scheduling are often solved with integer programming and are similar to assignment problems. In this example, you have eight one-hour time slots in each of five days. You have to assign four people to these time slots so that each slot is covered on every day. You allow the people to specify preference data for each slot on each day. In addition, there are constraints that must be satisfied:

- Each person has some slots for which they are unavailable.
- Each person must have either slot 4 or 5 off for lunch.
- Each person can work only two time slots in a row.
- Each person can work only a specified number of hours in the week.

To formulate this problem, let i denote person, j denote time slot, and k denote day. Then, let $x_{ijk} = 1$ if person i is assigned to time slot j on day k, and 0 otherwise; let p_{ijk} denote the preference of person i for slot

j on day k; and let h_i denote the number of hours in a week that person i will work. Then, you get

$$
\begin{array}{lll}
\max & \sum_{ijk} p_{ijk} x_{ijk} & \\
\text{subject to} & \sum_i x_{ijk} = 1 & \text{for all } j \text{ and } k \\
& x_{i4k} + x_{i5k} \leq 1 & \text{for all } i \text{ and } k \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \leq 2 & \text{for all } i \text{ and } k, \text{ and } \ell = 1, \ldots, 6 \\
& \sum_{jk} x_{ijk} \leq h_i & \text{for all } i \\
& x_{ijk} = 0 \text{ or } 1 & \text{for all } i \text{ and } k \text{ such that } p_{ijk} > 0, \\
& & \quad \text{otherwise } x_{ijk} = 0
\end{array}
$$

To solve this problem, create a data set that has the hours and preference data for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available.

```
data raw;
   input name $ hour slot mon tue wed thu fri;
   datalines;
marc  20   1    10 10 10 10 10
marc  20   2     9  9  9  9  9
marc  20   3     8  8  8  8  8
marc  20   4     1  1  1  1  1
marc  20   5     1  1  1  1  1
marc  20   6     1  1  1  1  1
marc  20   7     1  1  1  1  1
marc  20   8     1  1  1  1  1
mike  20   1    10  9  8  7  6
mike  20   2    10  9  8  7  6
mike  20   3    10  9  8  7  6
mike  20   4    10  3  3  3  3
mike  20   5     1  1  1  1  1
mike  20   6     1  2  3  4  5
mike  20   7     1  2  3  4  5
mike  20   8     1  2  3  4  5
bill  20   1    10 10 10 10 10
bill  20   2     9  9  9  9  9
bill  20   3     8  8  8  8  8
bill  20   4     0  0  0  0  0
bill  20   5     1  1  1  1  1
bill  20   6     1  1  1  1  1
bill  20   7     1  1  1  1  1
bill  20   8     1  1  1  1  1
bob   20   1    10  9  8  7  6
bob   20   2    10  9  8  7  6
bob   20   3    10  9  8  7  6
bob   20   4    10  3  3  3  3
bob   20   5     1  1  1  1  1
bob   20   6     1  2  3  4  5
bob   20   7     1  2  3  4  5
bob   20   8     1  2  3  4  5
;
```

These data are read by the following DATA step, and an integer program is built to solve the problem. The model is saved in the data set named MODEL. First, the objective function is built using the data saved in the RAW data set. Then, the constraints requiring a person to be working in each time slot are built. Next, the constraints allowing each person time for lunch are added. Then, the constraints restricting people to only two consecutive hours are added. Next, the constraints limiting the time that any one person works in a week are added. Finally, the constraints allowing a person to be assigned only to a time slot for which he is available are added. The code to build each of these constraints follows the formulation closely.

```
data model;
   array workweek{5} mon tue wed thu fri;
   array hours{4} hours1 hours2 hours3 hours4;
   retain hours1-hours4;

   set raw end=eof;

   length _row_ $ 8 _col_ $ 8 _type_ $ 8;
   keep _type_ _col_ _row_ _coef_;

   if      name='marc' then i=1;
   else if name='mike' then i=2;
   else if name='bill' then i=3;
   else if name='bob'  then i=4;

   hours{i}=hour;

/* build the objective function */

do k=1 to 5;
   _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);

   _row_='object';
   _coef_=workweek{k} * 1000;
   output;
   _row_='upper';
   if workweek{k}^=0 then _coef_=1;
   output;
   _row_='integer';
   _coef_=1;
   output;
end;

/* build the rest of the model */

if eof then do;
   _coef_=.;
   _col_=' ';
   _type_='upper';
   _row_='upper';
   output;
   _type_='max';
   _row_='object';
   output;
   _type_='int';
```

```
   _row_='integer';
   output;

   /* every hour 1 person working */

   do j=1 to 8;
      do k=1 to 5;
         _row_='work'||put(j,1.)||put(k,1.);
         _type_='eq';
         _col_='_RHS_';
         _coef_=1;
         output;
         _coef_=1;
         _type_=' ';
         do i=1 to 4;
            _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
            output;
         end;
      end;
   end;

/* each person has a lunch */

do i=1 to 4;
   do k=1 to 5;
      _row_='lunch'||put(i,1.)||put(k,1.);
      _type_='le';
      _col_='_RHS_';
      _coef_=1;
      output;
      _coef_=1;
      _type_=' ';
      _col_='x'||put(i,1.)||'4'||put(k,1.);
      output;
      _col_='x'||put(i,1.)||'5'||put(k,1.);
      output;
   end;
end;

/* work at most 2 slots in a row */

do i=1 to 4;
   do k=1 to 5;
      do l=1 to 6;
      _row_='seq'||put(i,1.)||put(k,1.)||put(l,1.);
      _type_='le';
      _col_='_RHS_';
      _coef_=2;
       output;
      _coef_=1;
      _type_=' ';
         do j=0 to 2;
            _col_='x'||put(i,1.)||put(l+j,1.)||put(k,1.);
            output;
```

```
         end;
      end;
   end;
end;

/* work at most n hours in a week */

do i=1 to 4;
   _row_='capacit'||put(i,1.);
   _type_='le';
   _col_='_RHS_';
   _coef_=hours{i};
   output;
   _coef_=1;
   _type_=' ';
   do j=1 to 8;
      do k=1 to 5;
         _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
         output;
      end;
   end;
end;
end;
run;
```

The model saved in the data set named MODEL is in the sparse format. The constraint that requires one person to work in time slot 1 on day 2 is named WORK12; it is $\sum_i x_{i12} = 1$.

The following model is saved in the MODEL data set (which has 1387 observations).

```
_TYPE_      _COL_    _ROW_       _COEF_

 eq         _RHS_    work12         1
            x112     work12         1
            x212     work12         1
            x312     work12         1
            x412     work12         1
```

The model is solved using the LP procedure. The option PRIMALOUT=SOLUTION causes PROC LP to save the primal solution in the data set named SOLUTION.

```
/* solve the linear program */

proc lp sparsedata noprint primalout=solution
   time=1000 maxit1=1000 maxit2=1000;
run;
```

The following DATA step takes the solution data set SOLUTION and generates a report data set named REPORT. It translates the variable names x_{ijk} so that a more meaningful report can be written. Then, the PROC TABULATE procedure is used to display a schedule showing how the eight time slots are covered for the week.

```
/* report the solution */
title 'Reported Solution';

data report;
   set solution;
   keep name slot mon tue wed thu fri;
   if substr(_var_,1,1)='x' then do;
     if _value_>0 then do;
        n=substr(_var_,2,1);
        slot=substr(_var_,3,1);
        d=substr(_var_,4,1);
        if      n='1' then name='marc';
        else if n='2' then name='mike';
        else if n='3' then name='bill';
        else               name='bob';
        if      d='1' then mon=1;
        else if d='2' then tue=1;
        else if d='3' then wed=1;
        else if d='4' then thu=1;
        else               fri=1;
        output;
     end;
  end;
run;

proc format;
   value xfmt 1='  xxx   ';
run;

proc tabulate data=report;
   class name slot;
   var mon--fri;
   table (slot * name), (mon tue wed thu fri)*sum=' '*f=xfmt.
          /misstext=' ';
run;
```

Output 5.13.1 from PROC TABULATE summarizes the schedule. Notice that the constraint requiring that a person be assigned to each possible time slot on each day is satisfied.

Output 5.13.1 A Scheduling Problem

Reported Solution

		mon	tue	wed	thu	fri
slot	name					
1	bill	xxx	xxx	xxx	xxx	xxx
2	bob	xxx				
	marc		xxx	xxx	xxx	xxx
3	bob		xxx			
	marc			xxx	xxx	xxx
	mike	xxx				
4	mike	xxx	xxx	xxx	xxx	xxx
5	bob	xxx	xxx	xxx	xxx	xxx
6	bob		xxx		xxx	
	marc	xxx				
	mike			xxx		xxx
7	bill	xxx				
	bob			xxx		
	mike		xxx		xxx	xxx
8	bill	xxx				
	bob					xxx
	mike		xxx	xxx	xxx	

Recall that PROC LP puts a character string in the macro variable _ORLP_ that describes the characteristics of the solution on termination. This string can be parsed using macro functions and the information obtained can be used in report writing. The variable can be written to the log with the command

```
%put &_orlp_;
```

which produces Figure 5.1.

Figure 5.1 _ORLP_ Macro Variable

```
STATUS=SUCCESSFUL PHASE=3 OBJECTIVE=211000 P_FEAS=YES D_FEAS=YES
        INT_ITER=0 INT_FEAS=1 ACTIVE=0 INT_BEST=211000 PHASE1_ITER=34
        PHASE2_ITER=51 PHASE3_ITER=0
```

From this you learn, for example, that at termination the solution is integer optimal and has an objective value of 211000.

Example 5.14: A Multicommodity Transshipment Problem with Fixed Charges

The following example illustrates a DATA step program for generating a linear program to solve a multicommodity network flow model that has fixed charges. Consider a network consisting of the following nodes: farm-a, farm-b, farm-c, Chicago, St. Louis, and New York. You can ship four commodities from each farm to Chicago or St. Louis and from Chicago or St. Louis to New York. The following table shows the unit shipping cost for each of the four commodities across each of the arcs. The table also shows the supply (positive numbers) at each of the from nodes and the demand (negative numbers) at each of the to nodes. The fixed charge is a fixed cost for shipping any nonzero amount across an arc. For example, if any amount of any of the four commodities is sent from farm-c to St. Louis, then a fixed charge of 75 units is added to the shipping cost.

Table 5.13 Farms to Cities Network Problem

From Node	To Node	Unit Shipping Cost 1	2	3	4	Supply and Demand 1	2	3	4	Fixed Charge
farm-a	Chicago	20	15	17	22	100	100	40	.	100
farm-b	Chicago	15	15	15	30	100	200	50	50	75
farm-c	Chicago	30	30	10	10	40	100	75	100	100
farm-a	StLouis	30	25	27	22	.	.	.	.	150
farm-c	StLouis	10	9	11	10	.	.	.	.	75
Chicago	NY	75	75	75	75	-150	-200	-50	-75	200
StLouis	NY	80	80	80	80	.	.	.	.	200

The following program is designed to take the data in the form given in the preceding table. It builds the node arc incidence matrix for a network given in this form and adds integer variables to capture the fixed charge using the type of constraints discussed in Example 5.8. The program solves the model using PROC LP, saves the solution in the PRIMALOUT= data set named SOLUTION, and displays the solution. The DATA step can be easily modified to handle larger problems with similar structure.

```
title 'Multi-commodity Transshipment Problem with Fixed-Charges';
%macro dooversd;
   _coef_=sd{_i_};
   if sd{_i_}>0 then do;                /* the node is a supply node */
      _row_=from||' commodity'||put(_i_,2.);
      if from^=' ' then output;
      end;
   else if sd{_i_}<0 then do;           /* the node is a demand node */
      _row_=to||' commodity'||put(_i_,2.);
      if to^=' ' then output;
   end;
   else if from^=' ' & to^=' ' then do; /* a transshipment node      */
      _coef_=0;
      _row_=from||' commodity'||put(_i_,2.); output;
      _row_=to  ||' commodity'||put(_i_,2.); output;
   end;
%mend dooversd;
```

```
%macro dooverc;
   _col_=arc||' commodity'||put(_i_,2.);
   if from^=' ' & to^=' ' then do;  /* add node arc incidence matrix */
      _type_='le'; _row_=from||' commodity'||put(_i_,2.);
      _coef_=1;  output;
      _row_=to  ||' commodity'||put(_i_,2.);
      _coef_=-1; output;
      _type_='  '; _row_='obj';
      _coef_=c{_i_};  output;
      /* add fixed charge variables   */
      _type_='le'; _row_=arc;
      _coef_=1;  output;
      _col_='_rhs_';
      _type_='  ';
      _coef_=0;  output;
      _col_=arc||'fx';
      _coef_=-M; output;
      _row_='int';
      _coef_=1;  output;
      _row_='obj';
      _coef_=fx; output;
      _row_='upper';
      _coef_=1;  output;
   end;
%mend dooverc;


data network;
retain M 1.0e6;
length _col_ $ 22 _row_ $ 22;
keep _type_ _col_ _row_ _coef_ ;
array sd sd1-sd4;
array c c1-c4;

input arc $10. from $ to $ c1 c2 c3 c4 sd1 sd2 sd3 sd4 fx;

/* for the first observation define some of the rows */

if _n_=1 then do;
   _type_='upperbd'; _row_='upper'; output;
   _type_='lowerbd'; _row_='lower'; output;
   _type_='min';     _row_='obj';   output;
   _type_='integer'; _row_='int';   output;
end;

_col_='_rhs_';  _type_='le';

do _i_ = 1 to dim(sd);
   %dooversd;
end;
do _i_ = 1 to dim(c);
   %dooverc;
end;
```

```
datalines;
a-Chicago   farm-a  Chicago    20 15 17 22     100 100  40   .  100
b-Chicago   farm-b  Chicago    15 15 15 30     100 200  50  50   75
c-Chicago   farm-c  Chicago    30 30 10 10      40 100  75 100  100
a-StLouis   farm-a  StLouis    30 25 27 22       .   .   .   .  150
c-StLouis   farm-c  StLouis    10  9 11 10       .   .   .   .   75
Chicago-NY  Chicago NY         75 75 75 75    -150 -200 -50 -75 200
StLous-NY   StLouis NY         80 80 80 80       .   .   .   .  200
;

/* solve the model */

proc lp sparsedata pout=solution noprint;
run;

 /* print the solution */

data;
   set solution;
   rename _var_=arc _value_=amount;
   if _value_^=0 & _type_='NON-NEG';
run;

proc print;
   id arc;
   var amount;
run;
```

The results from this example are shown in Output 5.14.1. The NOPRINT option in the PROC LP statement suppresses the Variable and Constraint Summary sections. This is useful when solving large models for which a report program is available. Here, the solution is saved in data set SOLUTION and reported using PROC PRINT. The solution shows the amount that is shipped over each arc.

Output 5.14.1 Multicommodity Transshipment Problem with Fixed Charges

Multi-commodity Transshipment Problem with Fixed-Charges

arc	amount
a-Chicago commodity 1	10
b-Chicago commodity 1	100
b-Chicago commodity 2	100
c-Chicago commodity 3	50
c-Chicago commodity 4	75
c-StLouis commodity 1	40
c-StLouis commodity 2	100
Chicago-NY commodity 1	110
Chicago-NY commodity 2	100
Chicago-NY commodity 3	50
Chicago-NY commodity 4	75
StLous-NY commodity 1	40
StLous-NY commodity 2	100

Example 5.15: Converting to an MPS-Format SAS Data Set

This example demonstrates the use of the MPSOUT= option to convert problem data in PROC LP input format into an MPS-format SAS data set for use with the OPTLP procedure.

Consider the oil blending problem introduced in the section "An Introductory Example" on page 171. Suppose you have saved the problem data in dense format by using the following DATA step:

```
data exdata;
   input _id_ $17. a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2 _type_ $ _rhs_;
datalines;
profit            -175 -165 -205  0  0  0 300 300 max     .
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq     0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq     0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq     0
recipe_1             0    0    0  0 .3 .7  -1   0  eq     0
recipe_2             0    0    0 .2  0 .8   0  -1  eq     0
available          110  165   80  .  .  .   .   . upperbd .
;
```

If you decide to solve the problem by using the OPTLP procedure, you will need to convert the data set exdata from dense format to MPS format. You can accomplish this by using the following statements:

```
proc lp data=exdata mpsout=mpsdata;
run;
```

The MPS-format SAS data set mpsdata is shown in Output 5.15.1.

Output 5.15.1 Data Set mpsdata

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		PROBLEM	.		.
2	ROWS			.		.
3	MAX	profit		.		.
4	E	naphtha_l_conv		.		.
5	E	naphtha_i_conv		.		.
6	E	heating_o_conv		.		.
7	E	recipe_1		.		.
8	E	recipe_2		.		.
9	COLUMNS			.		.
10		a_light	profit	-175.000	naphtha_l_conv	0.035
11		a_light	naphtha_i_conv	0.100	heating_o_conv	0.390
12		a_heavy	profit	-165.000	naphtha_l_conv	0.030
13		a_heavy	naphtha_i_conv	0.075	heating_o_conv	0.300
14		brega	profit	-205.000	naphtha_l_conv	0.045
15		brega	naphtha_i_conv	0.135	heating_o_conv	0.430
16		naphthal	naphtha_l_conv	-1.000	recipe_2	0.200
17		naphthai	naphtha_i_conv	-1.000	recipe_1	0.300
18		heatingo	heating_o_conv	-1.000	recipe_1	0.700
19		heatingo	recipe_2	0.800		.
20		jet_1	profit	300.000	recipe_1	-1.000
21		jet_2	profit	300.000	recipe_2	-1.000
22	BOUNDS			.		.
23	UP	.BOUNDS.	a_light	110.000		.
24	UP	.BOUNDS.	a_heavy	165.000		.
25	UP	.BOUNDS.	brega	80.000		.
26	ENDATA			.		.

Now that the problem data are in MPS format, you can solve the problem by using the OPTLP procedure. For more information, see Chapter 12, "The OPTLP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

Example 5.16: Migration to OPTMODEL: Assignment

The following example shows how to solve Example 5.12 using PROC OPTMODEL. The OBJECT, DEMAND, and RESOURCE data sets are the same as in that example. A new data set, GRADE, is added to aid in separating the data from the model.

```
title 'An Assignment Problem';

data grade(drop=i);
   do i = 1 to 6;
      grade = 'grade'||put(i,1.);
      output;
   end;
run;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SOLUTION:

```
proc optmodel;
   /* declare index sets */
   set CUSTOMERS;
   set <str> GRADES;
   set MACHINES;

   /* declare parameters */
   num return {CUSTOMERS, GRADES, MACHINES} init 0;
   num demand {CUSTOMERS, GRADES};
   num cost {GRADES, MACHINES} init 0;
   num avail {MACHINES};

   /* read the set of grades */
   read data grade into GRADES=[grade];

   /* read the set of customers and their demands */
   read data demand
      into CUSTOMERS=[customer]
      {j in GRADES} <demand[customer,j]=col(j)>;

   /* read the set of machines, costs, and availability */
   read data resource nomiss
      into MACHINES=[machine]
      {j in GRADES} <cost[j,machine]=col(j)>
      avail;

   /* read objective data */
   read data object nomiss
      into [machine customer]
      {j in GRADES} <return[customer,j,machine]=col(j)>;

   /* declare the model */
   var AmountProduced {CUSTOMERS, GRADES, MACHINES} >= 0;
   max TotalReturn = sum {i in CUSTOMERS, j in GRADES, k in MACHINES}
      return[i,j,k] * AmountProduced[i,j,k];
```

```
   con req_demand {i in CUSTOMERS, j in GRADES}:
      sum {k in MACHINES} AmountProduced[i,j,k] = demand[i,j];
   con req_avail {k in MACHINES}:
      sum {i in CUSTOMERS, j in GRADES}
         cost[j,k] * AmountProduced[i,j,k] <= avail[k];

   /* call the solver and save the results */
   solve;
   create data solution
      from [customer grade machine] = {i in CUSTOMERS, j in GRADES,
         k in MACHINES: AmountProduced[i,j,k].sol ne 0}
      amount=AmountProduced;

   /* print optimal solution */
   print AmountProduced;
quit;
```

The statements use both numeric (NUM) and character (STR) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The OPTMODEL parameters can be either single-dimensional (AVAIL) or multiple-dimensional (COST, DEMAND, RETURN). The RETURN and COST parameters are given initial values of 0, and the NOMISS option in the READ DATA statement tells OPTMODEL to read only the nonmissing values from the input data sets. The model declaration is nearly identical to the mathematical formulation. The logical condition **`AmountProduced[i,j,k].sol ne 0`** in the CREATE DATA statement makes sure that only the nonzero parts of the solution appear in the SOLUTION data set. In Example 5.12, the creation of this data set required postprocessing of the PROC LP output data set.

The main point is that the PROC OPTMODEL statements are much easier to read and maintain than the corresponding DATA step statements required for PROC LP.

The SOLUTION data set can be processed by PROC TABULATE as follows to create a compact representation of the solution:

```
proc tabulate data=solution;
   class customer grade machine;
   var   amount;
   table (machine*customer), (grade*amount=''*sum='');
run;
```

The output is the same as Output 5.12.2. The log is displayed in Output 5.16.1.

Output 5.16.1 OPTMODEL Log

```
NOTE: There were 6 observations read from the data set WORK.GRADE.
NOTE: There were 5 observations read from the data set WORK.DEMAND.
NOTE: There were 4 observations read from the data set WORK.RESOURCE.
NOTE: There were 20 observations read from the data set WORK.OBJECT.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 120 variables (0 free, 0 fixed).
NOTE: The problem has 34 linear constraints (4 LE, 30 EQ, 0 GE, 0 range).
NOTE: The problem has 220 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 43 variables and 7 constraints.
NOTE: The LP presolver removed 66 constraint coefficients.
NOTE: The presolved problem has 77 variables, 27 constraints, and 154
      constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                            Objective
      Phase Iteration         Value         Time
       D 1           1    0.000000E+00          0
       D 2           2    2.753237E+06          0
       D 2          55    8.714553E+05          0
       D 2          59    8.714260E+05          0
NOTE: Optimal.
NOTE: Objective = 871426.03763.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.SOLUTION has 26 observations and 4 variables.
```

Example 5.17: Migration to OPTMODEL: Multicommodity Transshipment

The following example shows how to solve Example 5.14 using PROC OPTMODEL. Three data sets contain the input data used in that example.

```
title 'Multicommodity Transshipment Problem with Fixed Charges';

data commodity_data;
   do c = 1 to 4;
      output;
   end;
run;

data arc_data;
   input from $ to $ c1 c2 c3 c4 fx;
   datalines;
farm-a  Chicago 20 15 17 22 100
farm-b  Chicago 15 15 15 30  75
```

```
farm-c  Chicago 30 30 10 10 100
farm-a  StLouis 30 25 27 22 150
farm-c  StLouis 10 9 11 10   75
Chicago NY      75 75 75 75 200
StLouis NY      80 80 80 80 200
;
run;

data supply_data;
   input node $ sd1 sd2 sd3 sd4;
   datalines;
farm-a  100  100  40   .
farm-b  100  200  50  50
farm-c   40  100  75 100
NY     -150 -200 -50 -75
;
run;
```

The following PROC OPTMODEL statements read the data sets, print the input parameters, build the mixed-integer linear programming model, solve the model, and output the optimal solution to a SAS data set called SOLUTION:

```
proc optmodel;
   set COMMODITIES;
   read data commodity_data into COMMODITIES=[c];

   set <str,str> ARCS;
   num unit_cost {ARCS, COMMODITIES};
   num fixed_charge {ARCS};
   read data arc_data into ARCS=[from to] {c in COMMODITIES}
      <unit_cost[from,to,c]=col('c'||c)> fixed_charge=fx;
   print unit_cost fixed_charge;

   set <str> NODES = union {<i,j> in ARCS} {i,j};
   num supply {NODES, COMMODITIES} init 0;
   read data supply_data nomiss into [node] {c in COMMODITIES}
      <supply[node,c]=col('sd'||c)>;
   print supply;

   var AmountShipped {ARCS, c in COMMODITIES} >= 0 <= sum {i in NODES}
      max(supply[i,c],0);

   /* UseArc[i,j] = 1 if arc (i,j) is used, 0 otherwise */
   var UseArc {ARCS} binary;

   /* TotalCost = variable costs + fixed charges */
   min TotalCost = sum {<i,j> in ARCS, c in COMMODITIES}
      unit_cost[i,j,c] * AmountShipped[i,j,c]
      + sum {<i,j> in ARCS} fixed_charge[i,j] * UseArc[i,j];

   con flow_balance {i in NODES, c in COMMODITIES}:
      sum {<(i),j> in ARCS} AmountShipped[i,j,c] -
      sum {<j,(i)> in ARCS} AmountShipped[j,i,c] <= supply[i,c];
```

```
   /* if AmountShipped[i,j,c] > 0 then UseArc[i,j] = 1 */
   con fixed_charge_def {<i,j> in ARCS, c in COMMODITIES}:
      AmountShipped[i,j,c] <= AmountShipped[i,j,c].ub * UseArc[i,j];

   solve;

   print AmountShipped;

   create data solution from [from to commodity]={<i,j> in ARCS,
   c in COMMODITIES: AmountShipped[i,j,c].sol ne 0} amount=AmountShipped;
quit;
```

Although Example 5.14 used **`M = 1.0e6`** in the FIXED_CHARGE_DEF constraint that links the continuous variable to the binary variable, it is numerically preferable to use a smaller, data-dependent value. Here, the upper bound on **`AmountShipped[i,j,c]`** is used instead. This upper bound is calculated in the first VAR statement as the sum of all positive supplies for commodity c. The logical condition **`AmountShipped[i,j,k].sol ne 0`** in the CREATE DATA statement makes sure that only the nonzero parts of the solution appear in the SOLUTION data set.

The optimal solution is the same as in Output 5.14.1. The log is displayed in Output 5.17.1.

Output 5.17.1 OPTMODEL Log

```
NOTE: There were 4 observations read from the data set WORK.COMMODITY_DATA.
NOTE: There were 7 observations read from the data set WORK.ARC_DATA.
NOTE: There were 4 observations read from the data set WORK.SUPPLY_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 35 variables (0 free, 0 fixed).
NOTE: The problem has 7 binary and 0 integer variables.
NOTE: The problem has 52 linear constraints (52 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 112 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 6 variables and 14 constraints.
NOTE: The MILP presolver removed 26 constraint coefficients.
NOTE: The MILP presolver modified 22 constraint coefficients.
NOTE: The presolved problem has 29 variables, 38 constraints, and 86 constraint
      coefficients.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       2  43180.0000000  26450.0000000   63.25%       0
             0       1       2  43180.0000000  42610.0000000    1.34%       0
             0       1       3  42825.0000000  42824.9900000    0.00%       0
NOTE: The MILP solver added 8 cuts with 28 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 42825.
NOTE: The data set WORK.SOLUTION has 14 observations and 4 variables.
```

References

Bartels, R. (1971), "A Stabilization of the Simplex Method," *Numerical Mathematics*, 16, 414–434.

Bland, R. G. (1977), "New Finite Pivoting Rules for the Simplex Method," *Mathematics of Operations Research*, 2, 103–107.

Breau, R. and Burdet, C. A. (1974), "Branch and Bound Experiments in Zero-One Programming," *Mathematical Programming Study*, 2, 1–50.

Crowder, H., Johnson, E. L., and Padberg, M. W. (1983), "Solving Large-Scale Zero-One Linear Programming Problems," *Operations Research*, 31, 803–834.

Dantzig, G. B. (1963), *Linear Programming and Extensions*, Princeton, NJ: Princeton University Press.

Garfinkel, R. S. and Nemhauser, G. L. (1972), *Integer Programming*, New York: John Wiley & Sons.

Greenberg, H. J. (1978), "Pivot Selection Tactics," in H. J. Greenberg, ed., *Design and Implementation of Optimization Software*, 143–174, Leiden, Netherlands: Sijthoff & Noordhoff.

Hadley, G. (1962), *Linear Programming*, Reading, MA: Addison-Wesley.

Harris, P. M. J. (1975), "Pivot Selection Methods of the Devex LP Code," *Mathematical Programming Study*, 4, 30–57.

Ignizio, J. P. (1976), *Goal Programming and Extensions*, Lexington, MA: D. C. Heath.

Murtagh, B. A. (1981), *Advanced Linear Programming: Computation and Practice*, New York: McGraw-Hill.

Nelson, M. (1992), *The Data Compression Book*, New York: M&T Books.

Reid, J. K. (1975), *A Sparsity-Exploiting Variant of the Bartels-Golub Decomposition for Linear Programming Bases*, Technical Report Harwell CSS 20, Atomic Energy Research Establishment, Harwell, UK.

Reid, J. K. (1976), *Fortran Subroutines for Handling Sparse Linear Programming Bases*, Technical Report Harwell AERE R 8269, Atomic Energy Research Establishment, Harwell, UK.

Savelsbergh, M. W. P. (1994), "Preprocessing and Probing Techniques for Mixed Integer Programming Problems," *ORSA Journal on Computing*, 6, 445–454.

Taha, H. A. (1975), *Integer Programming*, New York: Academic Press.

Chapter 6
The NETFLOW Procedure

Contents

Overview: NETFLOW Procedure **303**
- Introduction 303
- Network Models 304
- Side Constraints 305
- Advantages of Network Models over LP Models 310
- Mathematical Description of NPSC 311
- Flow Conservation Constraints 312
- Nonarc Variables 312
- Warm Starts 313

Getting Started: NETFLOW Procedure **314**
- Introductory Example 315

Syntax: NETFLOW Procedure **321**
- Functional Summary 321
- Interactivity 326
- PROC NETFLOW Statement 328
- CAPACITY Statement 340
- COEF Statement 341
- COLUMN Statement 341
- CONOPT Statement 341
- COST Statement 342
- DEMAND Statement 342
- HEADNODE Statement 342
- ID Statement 342
- LO Statement 343
- MULT Statement 343
- NAME Statement 343
- NODE Statement 344
- PIVOT Statement 344
- PRINT Statement 344
- QUIT Statement 350
- RESET Statement 350
- RHS Statement 370
- ROW Statement 371
- RUN Statement 371
- SAVE Statement 371
- SHOW Statement 373

SUPDEM Statement . . . 377
SUPPLY Statement . . . 377
TAILNODE Statement . . . 377
TYPE Statement . . . 378
VAR Statement . . . 379
Details: NETFLOW Procedure . . . **380**
Input Data Sets . . . 380
Output Data Sets . . . 388
Converting Any PROC NETFLOW Format to an MPS-Format SAS Data Set . . . 391
Case Sensitivity . . . 392
Loop Arcs . . . 392
Multiple Arcs . . . 393
Pricing Strategies . . . 393
Dual Variables, Reduced Costs, and Status . . . 396
The Working Basis Matrix . . . 397
Flow and Value Bounds . . . 399
Tightening Bounds and Side Constraints . . . 399
Reasons for Infeasibility . . . 399
Missing S Supply and Missing D Demand Values . . . 401
Balancing Total Supply and Total Demand . . . 405
Warm Starts . . . 406
How to Make the Data Read of PROC NETFLOW More Efficient . . . 410
Macro Variable _ORNETFL . . . 415
Memory Limit . . . 416
The Interior Point Algorithm: NETFLOW Procedure . . . **417**
Introduction . . . 417
Network Models: Interior Point Algorithm . . . 418
Linear Programming Models: Interior Point Algorithm . . . 428
Generalized Networks: NETFLOW Procedure . . . **449**
What Is a Generalized Network? . . . 449
How to Specify Data for Arc Multipliers . . . 451
Using the New EXCESS= Option in Pure Networks: NETFLOW Procedure . . . **454**
Handling Excess Supply or Demand . . . 455
Handling Missing Supply and Demand Simultaneously . . . 456
Maximum Flow Problems . . . 457
Handling Supply and Demand Ranges . . . 460
Using the New EXCESS= Option in Generalized Networks: NETFLOW Procedure . . . **461**
How Generalized Networks Differ from Pure Networks . . . 461
The EXCESS=SUPPLY Option . . . 462
The EXCESS=DEMAND Option . . . 464
Examples: NETFLOW Procedure . . . **466**
Example 6.1: Shortest Path Problem . . . 466
Example 6.2: Minimum Cost Flow Problem . . . 468
Example 6.3: Using a Warm Start . . . 472

Example 6.4: Production, Inventory, Distribution Problem 473
Example 6.5: Using an Unconstrained Solution Warm Start 482
Example 6.6: Adding Side Constraints, Using a Warm Start 488
Example 6.7: Using a Constrained Solution Warm Start 496
Example 6.8: Nonarc Variables in the Side Constraints 503
Example 6.9: Pure Networks: Using the EXCESS= Option 512
Example 6.10: Maximum Flow Problem 517
Example 6.11: Generalized Networks: Using the EXCESS= Option 520
Example 6.12: Generalized Networks: Maximum Flow Problem 522
Example 6.13: Machine Loading Problem 524
Example 6.14: Generalized Networks: Distribution Problem 527
Example 6.15: Converting to an MPS-Format SAS Data Set 530
Example 6.16: Migration to OPTMODEL: Generalized Networks 532
Example 6.17: Migration to OPTMODEL: Maximum Flow 535
Example 6.18: Migration to OPTMODEL: Production, Inventory, Distribution 537
Example 6.19: Migration to OPTMODEL: Shortest Path 539
References **541**

Overview: NETFLOW Procedure

Introduction

Constrained network models can be used to describe a wide variety of real-world applications ranging from production, inventory, and distribution problems to financial applications. These problems can be solved with the NETFLOW procedure.

These models are conceptually easy since they are based on network diagrams that represent the problem pictorially. PROC NETFLOW accepts the network specification in a format that is particularly suited to networks. This not only simplifies problem description but also aids in the interpretation of the solution.

Certain algebraic features of networks are exploited by a specialized version of the simplex method so that solution times are reduced. Another optimization algorithm, the interior point algorithm, has been implemented in PROC NETFLOW and can be used as an alternative to the simplex algorithm to solve network problems.

Should PROC NETFLOW detect there are no arcs and nodes in the model's data, (that is, there is no network component), it assumes it is dealing with a linear programming (LP) problem. The interior point algorithm is automatically selected to perform the optimization.

You can also solve LP problems by using the OPTLP procedure. The OPTLP procedure requires a linear program to be specified by using a SAS data set that adheres to the MPS format, a widely accepted format in the optimization community. You can use the MPSOUT= option in the NETFLOW procedure to convert typical PROC NETFLOW format data sets into MPS-format SAS data sets.

Network Models

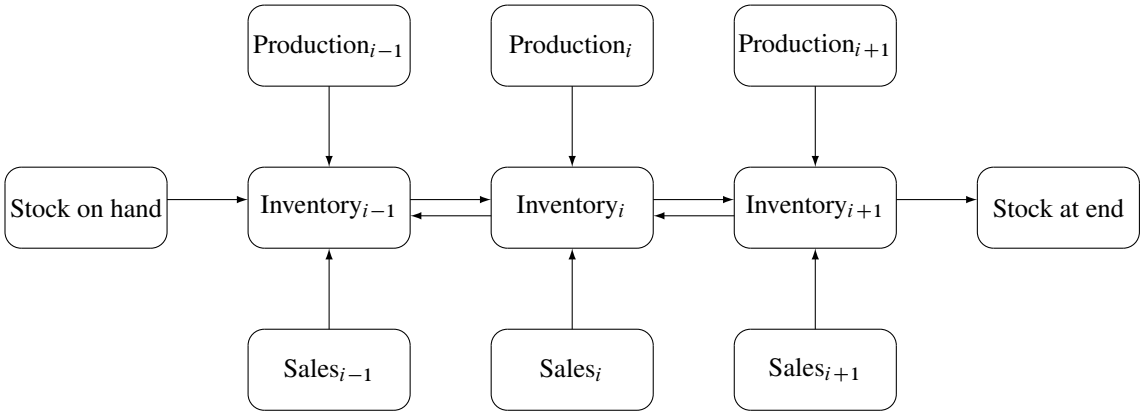
A network consists of a collection of nodes joined by a collection of arcs. The arcs connect nodes and convey flow of one or more commodities that are supplied at supply nodes and demanded at demand nodes in the network. Each arc has a cost per unit of flow, a flow capacity, and a lower flow bound associated with it. An important concept in network modeling is *conservation of flow*.

Conservation of flow means that the total flow in arcs directed toward a node, plus the supply at the node, minus the demand at the node, equals the total flow in arcs directed away from the node.

A network and its associated data can be described in SAS data sets. PROC NETFLOW uses this description and finds the flow through each arc in the network that minimizes the total cost of flow, meets the demand at demand nodes using the supply at supply nodes so that the flow through each arc is on or between the arc's lower flow bound and its capacity, and satisfies the conservation of flow.

One class of network models is the production-inventory-distribution problem. The diagram in Figure 6.1 illustrates this problem. The subscripts on the Production, Inventory, and Sales nodes indicate the time period. Notice that if you replicate sections of the model, the notion of time can be included.

Figure 6.1 Production-Inventory-Distribution Problem



In this type of model, the nodes can represent a wide variety of facilities. Several examples are suppliers, spot markets, importers, farmers, manufacturers, factories, parts of a plant, production lines, waste disposal facilities, workstations, warehouses, coolstores, depots, wholesalers, export markets, ports, rail junctions, airports, road intersections, cities, regions, shops, customers, and consumers. The diversity of this selection demonstrates the richness of potential applications of this model.

Depending upon the interpretation of the nodes, the objectives of the modeling exercise can vary widely. Some common types of objectives are

- to reduce collection or purchase costs of raw materials
- to reduce inventory holding or backorder costs. Warehouses and other storage facilities sometimes have capacities, and there can be limits on the amount of goods that can be placed on backorder.

- to decide where facilities should be located and what the capacity of these should be. Network models have been used to help decide where factories, hospitals, ambulance and fire stations, oil and water wells, and schools should be sited.
- to determine the assignment of resources (machines, production capability, workforce) to tasks, schedules, classes, or files
- to determine the optimal distribution of goods or services. This usually means minimizing transportation costs, and reducing time in transit or distances covered.
- to find the shortest path from one location to another
- to ensure that demands (for example, production requirements, market demands, contractual obligations) are met
- to maximize profits from the sale of products or the charge for services
- to maximize production by identifying bottlenecks

Some specific applications are

- car distribution models. These help determine which models and numbers of cars should be manufactured in which factories and where to distribute cars from these factories to zones in the United States in order to meet customer demand at least cost.
- models in the timber industry. These help determine when to plant and mill forests, schedule production of pulp, paper and wood products, and distribute products for sale or export.
- military applications. The nodes can be theatres, bases, ammunition dumps, logistical suppliers, or radar installations. Some models are used to find the best ways to mobilize personnel and supplies and to evacuate the wounded in the least amount of time.
- communications applications. The nodes can be telephone exchanges, transmission lines, satellite links, and consumers. In a model of an electrical grid, the nodes can be transformers, powerstations, watersheds, reservoirs, dams, and consumers. Of concern might be the effect of high loads or outages.

Side Constraints

Often all the details of a problem cannot be specified in a network model alone. In many of these cases, these details can be represented by the addition of side constraints to the model. Side constraints are a linear function of arc variables (variables containing flow through an arc) and nonarc variables (variables that are not part of the network). This enhancement to the basic network model allows for very general problems. In fact, any linear program can be represented with network models having these types of side constraints. The examples that follow help to clarify the notion of side constraints.

PROC NETFLOW enables you to specify side constraints. The data for a side constraint consist of coefficients of arcs and coefficients of nonarc variables, a constraint type (that is, $\leq$, =, or $\geq$) and a right-hand-side value (rhs). A nonarc variable has a name, an objective function coefficient analogous to an arc cost, an upper bound analogous to an arc capacity, and a lower bound analogous to an arc lower flow bound. PROC

NETFLOW finds the flow through the network and the values of any nonarc variables that minimize the total cost of the solution. Flow conservation is met, flow through each arc is on or between the arc's lower flow bound and capacity, the value of each nonarc variable is on or between the nonarc's lower and upper bounds, and the side constraints are satisfied. Note that, since many linear programs have large embedded networks, PROC NETFLOW is an attractive alternative to the LP procedure in many cases.

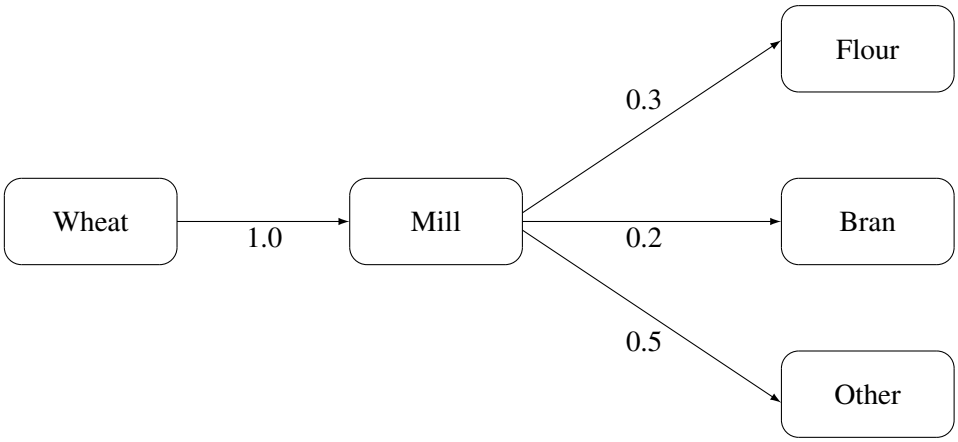
In order for arcs to be specified in side constraints, they must be named. By default, PROC NETFLOW names arcs using the names of the nodes at the head and tail of the arc. An arc is named with its tail node name followed by an underscore and its head node name. For example, an arc from node *from* to node *to* is called *from_to*.

Proportionality Constraints

Side constraints in network models fall into several categories that have special structure. They are frequently used when the flow through an arc must be proportional to the flow through another arc. Such constraints are called *proportionality constraints* and are useful in models where production is subject to refining or modification into different materials. The amount of each output, or any waste, evaporation, or reduction can be specified as a proportion of input.

Typically the arcs near the supply nodes carry raw materials and the arcs near the demand nodes carry refined products. For example, in a model of the milling industry, the flow through some arcs may represent quantities of wheat. After the wheat is processed, the flow through other arcs might be flour. For others it might be bran. The side constraints model the relationship between the amount of flour or bran produced as a proportion of the amount of wheat milled. Some of the wheat can end up as neither flour, bran, nor any useful product, so this waste is drained away via arcs to a waste node.

Figure 6.2 Proportionality Constraints



Consider the network fragment in Figure 6.2. The arc Wheat_Mill conveys the wheat milled. The cost of flow on this arc is the milling cost. The capacity of this arc is the capacity of the mill. The lower flow bound on this arc is the minimum quantity that must be milled for the mill to operate economically. The constraints

$$0.3\,\text{Wheat_Mill} - \text{Mill_Flour} = 0.0$$

$$0.2\,\text{Wheat_Mill} - \text{Mill_Bran} = 0.0$$

force every unit of wheat that is milled to produce 0.3 units of flour and 0.2 units of bran. Note that it is not necessary to specify the constraint

$$0.5\,\mathsf{Wheat_Mill} - \mathsf{Mill_Other} = 0.0$$

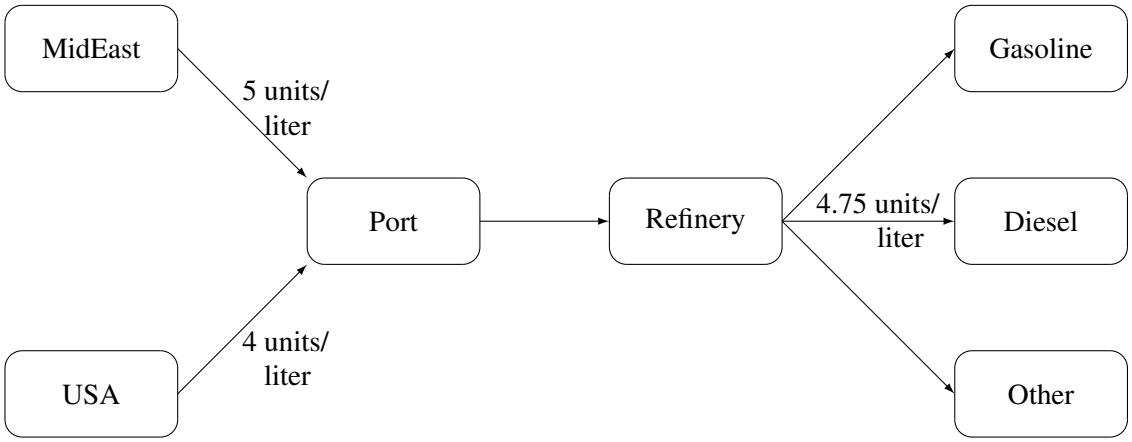
since flow conservation implies that any flow that does not traverse through Mill_Flour or Mill_Bran must be conveyed through Mill_Other. And, computationally, it is better if this constraint is not specified, since there is one less side constraint and fewer problems with numerical precision. Notice that the sum of the proportions must equal 1.0 exactly; otherwise, flow conservation is violated.

Blending Constraints

Blending or quality constraints can also influence the recipes or proportions of ingredients that are mixed. For example, different raw materials can have different properties. In an application of the oil industry, the amount of products that are obtained could be different for each type of crude oil. Furthermore, fuel might have a minimum octane requirement or limited sulphur or lead content, so that a blending of crudes is needed to produce the product.

The network fragment in Figure 6.3 shows an example of this.

Figure 6.3 Blending Constraints



The arcs MidEast_Port and USA_Port convey crude oil from the two sources. The arc Port_Refinery represents refining while the arcs Refinery_Gasoline and Refinery_Diesel carry the gas and diesel produced. The proportionality constraints

$$0.4\,\mathsf{Port_Refinery} - \mathsf{Refinery_Gasoline} = 0.0$$

$$0.2\,\mathsf{Port_Refinery} - \mathsf{Refinery_Diesel} = 0.0$$

capture the restrictions for producing gasoline and diesel from crude. Suppose that, if only crude from the Middle East is used, the resulting diesel would contain 5 units of sulphur per liter. If only crude from the

USA is used, the resulting diesel would contain 4 units of sulphur per liter. Diesel can have at most 4.75 units of sulphur per liter. Some crude from the USA must be used if Middle East crude is used in order to meet the 4.75 sulphur per liter limit. The side constraint to model this requirement is

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 4.75\,\text{Port_Refinery} \leq 0.0$$

Since Port_Refinery = MidEast_Port + USA_Port, flow conservation allows this constraint to be simplified to

$$1\,\text{MidEast_Port} - 3\,\text{USA_Port} \leq 0.0$$

If, for example, 120 units of crude from the Middle East is used, then at least 40 units of crude from the USA must be used. The preceding constraint is simplified because you assume that the sulphur concentration of diesel is proportional to the sulphur concentration of the crude mix. If this is not the case, the relation

$$0.2\,\text{Port_Refinery} = \text{Refinery_Diesel}$$

is used to obtain

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 4.75\,(1.0/0.2\ \ \text{Refinery_Diesel}) \leq 0.0$$

which equals

$$5\,\text{MidEast_Port} + 4\,\text{USA_Port} - 23.75\,\text{Refinery_Diesel} \leq 0.0$$

An example similar to this Oil Industry problem is solved in the section "Introductory Example" on page 315.

Multicommodity Problems

Side constraints are also used in models in which there are capacities on transportation or some other shared resource, or there are limits on overall production or demand in multicommodity, multidivisional or multiperiod problems. Each commodity, division or period can have a separate network coupled to one main system by the side constraints. Side constraints are used to combine the outputs of subdivisions of a problem (either commodities, outputs in distinct time periods, or different process streams) to meet overall demands or to limit overall production or expenditures. This method is more desirable than doing separate *local* optimizations for individual commodity, process, or time networks and then trying to establish relationships between each when determining an overall policy if the *global* constraint is not satisfied. Of course, to make models more realistic, side constraints may be necessary in the local problems.

Figure 6.4 Multicommodity Problem

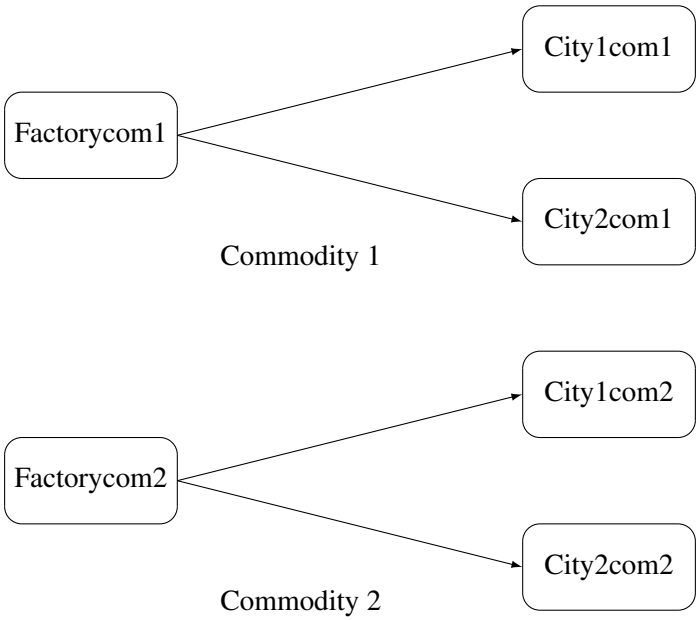


Figure 6.4 shows two network fragments. They represent identical production and distribution sites of two different commodities. Suffix *com1* represents commodity 1 and suffix *com2* represents commodity 2. The nodes Factorycom1 and Factorycom2 model the same factory, and nodes City1com1 and City1com2 model the same location, city 1. Similarly, City2com1 and City2com2 are the same location, city 2. Suppose that commodity 1 occupies 2 cubic meters, commodity 2 occupies 3 cubic meters, the truck dispatched to city 1 has a capacity of 200 cubic meters, and the truck dispatched to city 2 has a capacity of 250 cubic meters. How much of each commodity can be loaded onto each truck? The side constraints for this case are

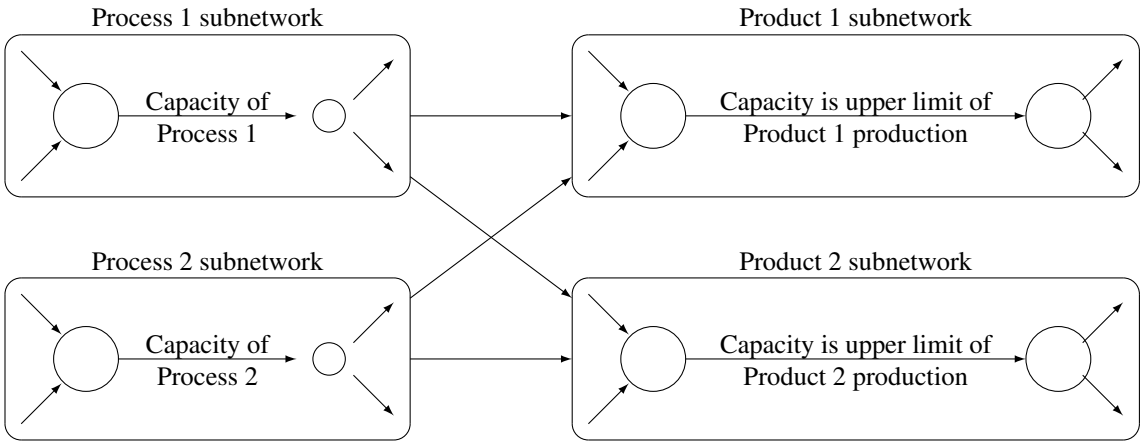
$$2\,\text{Factorycom1_City1com1} + 3\,\text{Factorycom2_City1com2} \leq 200$$

$$2\,\text{Factorycom1_City2com1} + 3\,\text{Factorycom2_City2com2} \leq 250$$

Large Modeling Strategy

In many cases, the flow through an arc might actually represent the flow or movement of a commodity from place to place or from time period to time period. However, sometimes an arc is included in the network as a method of capturing some aspect of the problem that you would not normally think of as part of a network model. For example, in a multiprocess, multiproduct model (Figure 6.5), there might be subnetworks for each process and each product. The subnetworks can be joined together by a set of arcs that have flows that represent the amount of product *j* produced by process *i*. To model an upper limit constraint on the total amount of product *j* that can be produced, direct all arcs carrying product *j* to a single node and from there through a single arc. The capacity of this arc is the upper limit of product *j* production. It is preferable to model this structure in the network rather than to include it in the side constraints because the efficiency of the optimizer is affected less by a reasonable increase in the size of the network.

Figure 6.5 Multiprocess, Multiproduct Example



It is often a good strategy when starting a project to use a small network formulation and then use that model as a framework upon which to add detail. For example, in the multiprocess, multiproduct model, you might start with the network depicted in Figure 6.5. Then, for example, the process subnetwork can be enhanced to include the distribution of products. Other phases of the operation could be included by adding more subnetworks. Initially, these subnetworks can be single nodes, but in subsequent studies they can be expanded to include greater detail.

The NETFLOW procedure accepts the side constraints in the same dense and sparse formats that the LP procedure provides. Although PROC LP can solve network problems, the NETFLOW procedure generally solves network flow problems more efficiently than PROC LP.

Advantages of Network Models over LP Models

Many linear programming problems have large embedded network structures. Such problems often result when modeling manufacturing processes, transportation or distribution networks, or resource allocation, or when deciding where to locate facilities. Often, some commodity is to be moved from place to place, so the more natural formulation in many applications is that of a constrained network rather than a linear program.

Using a network diagram to visualize a problem makes it possible to capture the important relationships in an easily understood picture form. The network diagram aids the communication between model builder and model user, making it easier to comprehend how the model is structured, how it can be changed, and how results can be interpreted.

If a network structure is embedded in a linear program, the problem is a network programming problem with side constraints (NPSC). When the network part of the problem is large compared to the nonnetwork part, especially if the number of side constraints is small, it is worthwhile to exploit this structure in the solution process. This is what PROC NETFLOW does. It uses a variant of the revised primal simplex algorithm that exploits the network structure to reduce solution time.

Mathematical Description of NPSC

If a network programming problem with side constraints has n nodes, a arcs, g nonarc variables, and k side constraints, then the formal statement of the problem solved by PROC NETFLOW is

$$\begin{array}{ll} \text{minimize} & c^T x + d^T z \\ \text{subject to} & Fx = b \\ & Hx + Qz \geq, =, \leq r \\ & l \leq x \leq u \\ & m \leq z \leq v \end{array}$$

where

- c is the $a \times 1$ arc variable objective function coefficient vector (the cost vector)
- x is the $a \times 1$ arc variable value vector (the flow vector)
- d is the $g \times 1$ nonarc variable objective function coefficient vector
- z is the $g \times 1$ nonarc variable value vector
- F is the $n \times a$ node-arc incidence matrix of the network, where

$$F_{i,j} = \begin{cases} -1, & \text{if arc } j \text{ is directed from node } i \\ 1, & \text{if arc } j \text{ is directed toward node } i \\ 0, & \text{otherwise} \end{cases}$$

- b is the $n \times 1$ node supply/demand vector, where

$$b_i = \begin{cases} s, & \text{if node } i \text{ has supply capability of } s \text{ units of flow} \\ -d, & \text{if node } i \text{ has demand of } d \text{ units of flow} \\ 0, & \text{if node } i \text{ is a trans-shipment node} \end{cases}$$

- H is the $k \times a$ side constraint coefficient matrix for arc variables, where $H_{i,j}$ is the coefficient of arc j in the ith side constraint
- Q is the $k \times g$ side constraint coefficient matrix for nonarc variables, where $Q_{i,j}$ is the coefficient of nonarc j in the ith side constraint
- r is the $k \times 1$ side constraint right-hand-side vector
- l is the $a \times 1$ arc lower flow bound vector
- u is the $a \times 1$ arc capacity vector
- m is the $g \times 1$ nonarc variable lower bound vector
- v is the $g \times 1$ nonarc variable upper bound vector

Flow Conservation Constraints

The constraints $Fx = b$ are referred to as the nodal flow conservation constraints. These constraints algebraically state that the sum of the flow through arcs directed toward a node plus that node's supply, if any, equals the sum of the flow through arcs directed away from that node plus that node's demand, if any. The flow conservation constraints are implicit in the network model and should not be specified explicitly in side constraint data when using PROC NETFLOW. The constrained problems most amenable to being solved by the NETFLOW procedure are those that, after the removal of the flow conservation constraints, have very few constraints. PROC NETFLOW is superior to linear programming optimizers when the network part of the problem is significantly larger than the nonnetwork part.

The NETFLOW procedure can also be used to solve an unconstrained network problem, that is, one in which H, Q, d, r, and z do not exist.

Nonarc Variables

If the constrained problem to be solved has no nonarc variables, then Q, d, and z do not exist. However, nonarc variables can be used to simplify side constraints. For example, if a sum of flows appears in many constraints, it may be worthwhile to equate this expression with a nonarc variable and use this in the other constraints. By assigning a nonarc variable a nonzero objective function, it is then possible to incur a cost for using resources above some lowest feasible limit. Similarly, a profit (a negative objective function coefficient value) can be made if all available resources are not used.

In some models, nonarc variables are used in constraints to absorb excess resources or supply needed resources. Then, either the excess resource can be used or the needed resource can be supplied to another component of the model.

For example, consider a multicommodity problem of making television sets that have either 19- or 25-inch screens. In their manufacture, 3 and 4 chips, respectively, are used. Production occurs at 2 factories during March and April. The supplier of chips can supply only 2600 chips to factory 1 and 3750 chips to factory 2 each month. The names of arcs are in the form Prod*n*_*s*_*m*, where n is the factory number, s is the screen size, and m is the month. For example, Prod1_25_Apr is the arc that conveys the number of 25-inch TVs produced in factory 1 during April. You might have to determine similar systematic naming schemes for your application.

As described, the constraints are

$$3\,\text{Prod1_19_Mar} + 4\,\text{Prod1_25_Mar} \leq 2600$$

$$3\,\text{Prod2_19_Mar} + 4\,\text{Prod2_25_Mar} \leq 3750$$

$$3\,\text{Prod1_19_Apr} + 4\,\text{Prod1_25_Apr} \leq 2600$$

$$3\,\text{Prod2_19_Apr} + 4\,\text{Prod2_25_Apr} \leq 3750$$

If there are chips that could be obtained for use in March but not used for production in March, why not keep these unused chips until April? Furthermore, if the March excess chips at factory 1 could be used either at factory 1 or factory 2 in April, the model becomes

$$3\,\text{Prod1_19_Mar} + 4\,\text{Prod1_25_Mar} + \text{F1_Unused_Mar} = 2600$$

$$3\,\text{Prod2_19_Mar} + 4\,\text{Prod2_25_Mar} + \text{F2_Unused_Mar} = 3750$$

$$3\,\text{Prod1_19_Apr} + 4\,\text{Prod1_25_Apr} - \text{F1_Kept_Since_Mar} = 2600$$

$$3\,\text{Prod2_19_Apr} + 4\,\text{Prod2_25_Apr} - \text{F2_Kept_Since_Mar} = 3750$$

$$\text{F1_Unused_Mar} + \text{F2_Unused_Mar} \text{ (continued)}$$

$$- \text{F1_Kept_Since_Mar} - \text{F2_Kept_Since_Mar} \geq 0.0$$

where F1_Kept_Since_Mar is the number of chips used during April at factory 1 that were obtained in March at either factory 1 or factory 2 and F2_Kept_Since_Mar is the number of chips used during April at factory 2 that were obtained in March. The last constraint ensures that the number of chips used during April that were obtained in March does not exceed the number of chips not used in March. There may be a cost to hold chips in inventory. This can be modeled having a positive objective function coefficient for the nonarc variables F1_Kept_Since_Mar and F2_Kept_Since_Mar. Moreover, nonarc variable upper bounds represent an upper limit on the number of chips that can be held in inventory between March and April.

See Example 6.4 through Example 6.8 for a series of examples that use this TV problem. The use of nonarc variables as described previously is illustrated.

Warm Starts

If you have a problem that has already been partially solved and is to be solved further to obtain a better, optimal solution, information describing the solution now available may be used as an initial solution. This is called *warm starting* the optimization, and the supplied solution data are called the *warm start*.

Some data can be changed between the time when a warm start is created and when it is used as a warm start for a subsequent PROC NETFLOW run. Elements in the arc variable cost vector, the nonarc variable objective function coefficient vector, and sometimes capacities, upper value bounds, and side constraint data can be changed between PROC NETFLOW calls. See the section "Warm Starts" on page 406. Also, see Example 6.4 through Example 6.8 (the TV problem) for a series of examples that show the use of warm starts.

Getting Started: NETFLOW Procedure

To solve network programming problems with side constraints using PROC NETFLOW, you save a representation of the network and the side constraints in three SAS data sets. These data sets are then passed to PROC NETFLOW for solution. There are various forms that a problem's data can take. You can use any one or a combination of several of these forms.

The NODEDATA= data set contains the names of the supply and demand nodes and the supply or demand associated with each. These are the elements in the column vector *b* in problem (NPSC).

The ARCDATA= data set contains information about the variables of the problem. Usually these are arcs, but there can also be data related to nonarc variables in the ARCDATA= data set.

An arc is identified by the names of its tail node (where it originates) and head node (where it is directed). Each observation can be used to identify an arc in the network and, optionally, the cost per flow unit across the arc, the arc's capacity, lower flow bound, and name. These data are associated with the matrix *F* and the vectors *c*, *l*, and *u* in problem (NPSC).

NOTE: Although *F* is a node-arc incidence matrix, it is specified in the ARCDATA= data set by arc definitions.

In addition, the ARCDATA= data set can be used to specify information about nonarc variables, including objective function coefficients, lower and upper value bounds, and names. These data are the elements of the vectors *d*, *m*, and *v* in problem (NPSC). Data for an arc or nonarc variable can be given in more than one observation.

Supply and demand data also can be specified in the ARCDATA= data set. In such a case, the NODEDATA= data set may not be needed.

The CONDATA= data set describes the side constraints and their right-hand sides. These data are elements of the matrices *H* and *Q* and the vector *r*. Constraint types are also specified in the CONDATA= data set. You can include in this data set upper bound values or capacities, lower flow or value bounds, and costs or objective function coefficients. It is possible to give all information about some or all nonarc variables in the CONDATA= data set.

An arc is identified in this data set by its name. If you specify an arc's name in the ARCDATA= data set, then this name is used to associate data in the CONDATA= data set with that arc. Each arc also has a default name that is the name of the tail and head node of the arc concatenated together and separated by an underscore character; tail_head, for example.

If you use the dense side constraint input format (described in the section "CONDATA= Data Set" on page 380) and want to use the default arc names, these arc names are names of SAS variables in the VAR list of the CONDATA= data set.

If you use the sparse side constraint input format (see the section "CONDATA= Data Set" on page 380) and want to use the default arc names, these arc names are values of the COLUMN list SAS variable of the CONDATA= data set.

The execution of PROC NETFLOW has three stages. In the preliminary (zeroth) stage, the data are read from the NODEDATA= data set, the ARCDATA= data set, and the CONDATA= data set. Error checking is performed, and an initial basic feasible solution is found. If an unconstrained solution warm start is being used, then an initial basic feasible solution is obtained by reading additional data containing that information in the NODEDATA= data set and the ARCDATA= data set. In this case, only constraint data and nonarc variable data are read from the CONDATA= data set.

In the first stage, an optimal solution to the network flow problem neglecting any side constraints is found. The primal and dual solutions for this relaxed problem can be saved in the ARCOUT= data set and the NODEOUT= data set, respectively. These data sets are named in the PROC NETFLOW, RESET, and SAVE statements.

In the second stage, an optimal solution to the network flow problem with side constraints is found. The primal and dual solutions for this side constrained problem are saved in the CONOUT= data set and the DUALOUT= data set, respectively. These data sets are also named in the PROC NETFLOW, RESET, and SAVE statements.

If a constrained solution warm start is being used, PROC NETFLOW does not perform the zeroth and first stages. This warm start can be obtained by reading basis data containing additional information in the NODEDATA= data set (also called the DUALIN= data set) and the ARCDATA= data set.

If warm starts are to be used in future optimizations, the FUTURE1 and FUTURE2 options must be used in addition to specifying names for the data sets that contain the primal and dual solutions in stages one and two. Then, most of the information necessary for restarting problems is available in the output data sets containing the primal and dual solutions of both the relaxed and side constrained network programs.

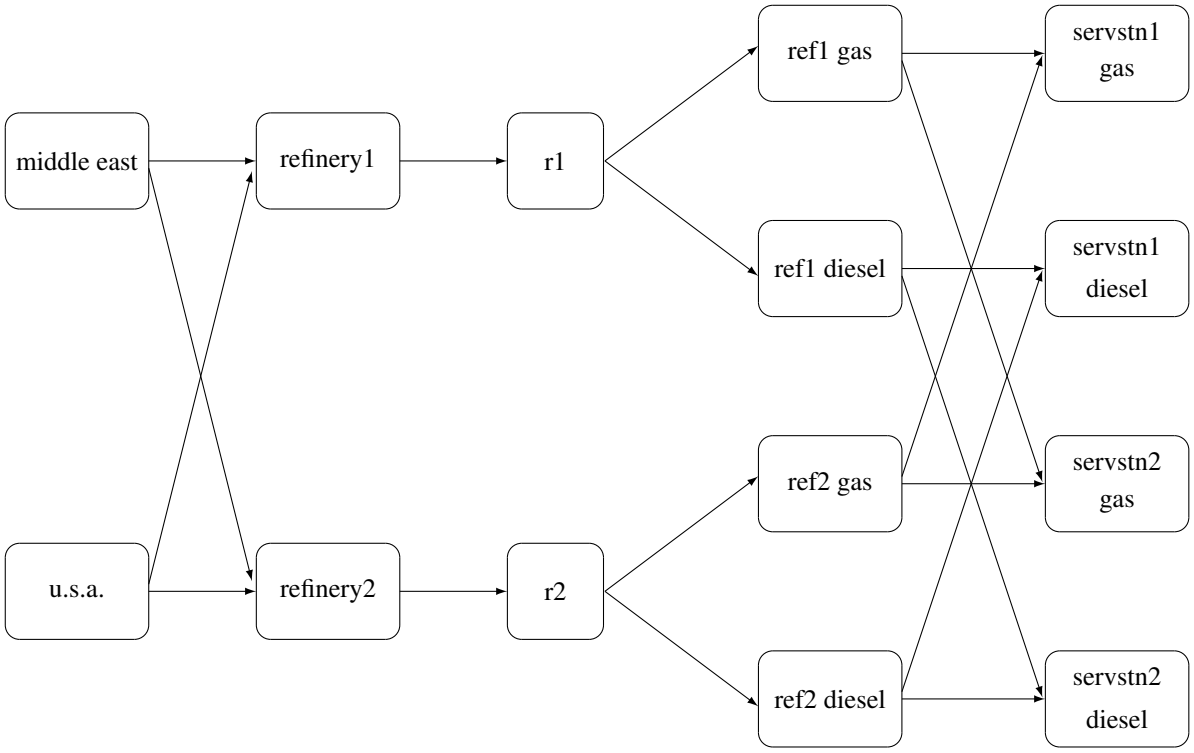
Introductory Example

Consider the following trans-shipment problem for an oil company. Crude oil is shipped to refineries where it is processed into gasoline and diesel fuel. The gasoline and diesel fuel are then distributed to service stations. At each stage, there are shipping, processing, and distribution costs. Also, there are lower flow bounds and capacities.

In addition, there are two sets of side constraints. The first set is that two times the crude from the Middle East cannot exceed the throughput of a refinery plus 15 units. (The phrase “plus 15 units” that finishes the last sentence is used to enable some side constraints in this example to have a nonzero rhs.) The second set of constraints are necessary to model the situation that one unit of crude mix processed at a refinery yields three-fourths of a unit of gasoline and one-fourth of a unit of diesel fuel.

Because there are two products that are not independent in the way in which they flow through the network, a network programming problem with side constraints is an appropriate model for this example (see Figure 6.6). The side constraints are used to model the limitations on the amount of Middle Eastern crude that can be processed by each refinery and the conversion proportions of crude to gasoline and diesel fuel.

Figure 6.6 Oil Industry Example



To solve this problem with PROC NETFLOW, save a representation of the model in three SAS data sets. In the NODEDATA= data set, you name the supply and demand nodes and give the associated supplies and demands. To distinguish demand nodes from supply nodes, specify demands as negative quantities. For the oil example, the NODEDATA= data set can be saved as follows:

```
title  'Oil Industry Example';
title3 'Setting Up Nodedata = Noded For Proc Netflow';
data noded;
  input   _node_&$15. _sd_;
  datalines;
middle east       100
u.s.a.             80
servstn1 gas      -95
servstn1 diesel   -30
servstn2 gas      -40
servstn2 diesel   -15
;
```

The ARCDATA= data set contains the rest of the information about the network. Each observation in the data set identifies an arc in the network and gives the cost per flow unit across the arc, the capacities of the arc, the lower bound on flow across the arc, and the name of the arc.

```
title3 'Setting Up Arcdata = Arcd1 For Proc Netflow';
data arcd1;
  input _from_&$11. _to_&$15. _cost_ _capac_ _lo_ _name_ $;
  datalines;
middle east    refinery 1        63     95   20    m_e_ref1
middle east    refinery 2        81     80   10    m_e_ref2
u.s.a.         refinery 1        55      .    .    .
u.s.a.         refinery 2        49      .    .    .
refinery 1     r1               200    175   50    thruput1
refinery 2     r2               220    100   35    thruput2
r1             ref1 gas           .    140    .    r1_gas
r1             ref1 diesel        .     75    .    .
r2             ref2 gas           .    100    .    r2_gas
r2             ref2 diesel        .     75    .    .
ref1 gas       servstn1 gas      15     70    .    .
ref1 gas       servstn2 gas      22     60    .    .
ref1 diesel    servstn1 diesel   18      .    .    .
ref1 diesel    servstn2 diesel   17      .    .    .
ref2 gas       servstn1 gas      17     35    5    .
ref2 gas       servstn2 gas      31      .    .    .
ref2 diesel    servstn1 diesel   36      .    .    .
ref2 diesel    servstn2 diesel   23      .    .    .
;
```

Finally, the CONDATA= data set contains the side constraints for the model.

```
title3 'Setting Up Condata = Cond1 For Proc Netflow';
data cond1;
  input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
        _type_ $ _rhs_;
  datalines;
-2  .  1 .  . . >= -15
. -2  . .  1 . GE -15
.  . -3 4  . . EQ   0
.  .  . . -3 4  =   0
;
```

Note that the SAS variable names in the CONDATA= data set are the names of arcs given in the ARCDATA= data set. These are the arcs that have nonzero constraint coefficients in side constraints. For example, the proportionality constraint that specifies that one unit of crude at each refinery yields three-fourths of a unit of gasoline and one-fourth of a unit of diesel fuel is given for REFINERY 1 in the third observation and for REFINERY 2 in the last observation. The third observation requires that each unit of flow on arc THRUPUT1 equals three-fourths of a unit of flow on arc R1_GAS. Because all crude processed at REFINERY 1 flows through THRUPUT1 and all gasoline produced at REFINERY 1 flows through R1_GAS, the constraint models the situation. It proceeds similarly for REFINERY 2 in the last observation.

To find the minimum cost flow through the network that satisfies the supplies, demands, and side constraints, invoke PROC NETFLOW as follows:

```
proc netflow
  nodedata=noded        /* the supply and demand data */
  arcdata=arcd1         /* the arc descriptions       */
  condata=cond1         /* the side constraints       */
  conout=solution;      /* the solution data set      */
  run;
```

The following messages, which appear on the SAS log, summarize the model as read by PROC NETFLOW and note the progress toward a solution:

```
NOTE: Number of nodes= 14 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 180 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of iterations performed (neglecting any constraints)= 14 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 50600 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 8 .
NOTE: Number of iterations, optimizing with constraints= 4 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= 50875 .
NOTE: The data set WORK.SOLUTION has 18 observations and 14 variables.
```

Unlike PROC LP, which displays the solution and other information as output, PROC NETFLOW saves the optimum in output SAS data sets that you specify. For this example, the solution is saved in the SOLUTION data set. It can be displayed with the PRINT procedure as

```
proc print data=solution;
  var _from_ _to_ _cost_ _capac_ _lo_ _name_
      _supply_ _demand_ _flow_ _fcost_ _rcost_;
  sum _fcost_;
  title3 'Constrained Optimum';
  run;
```

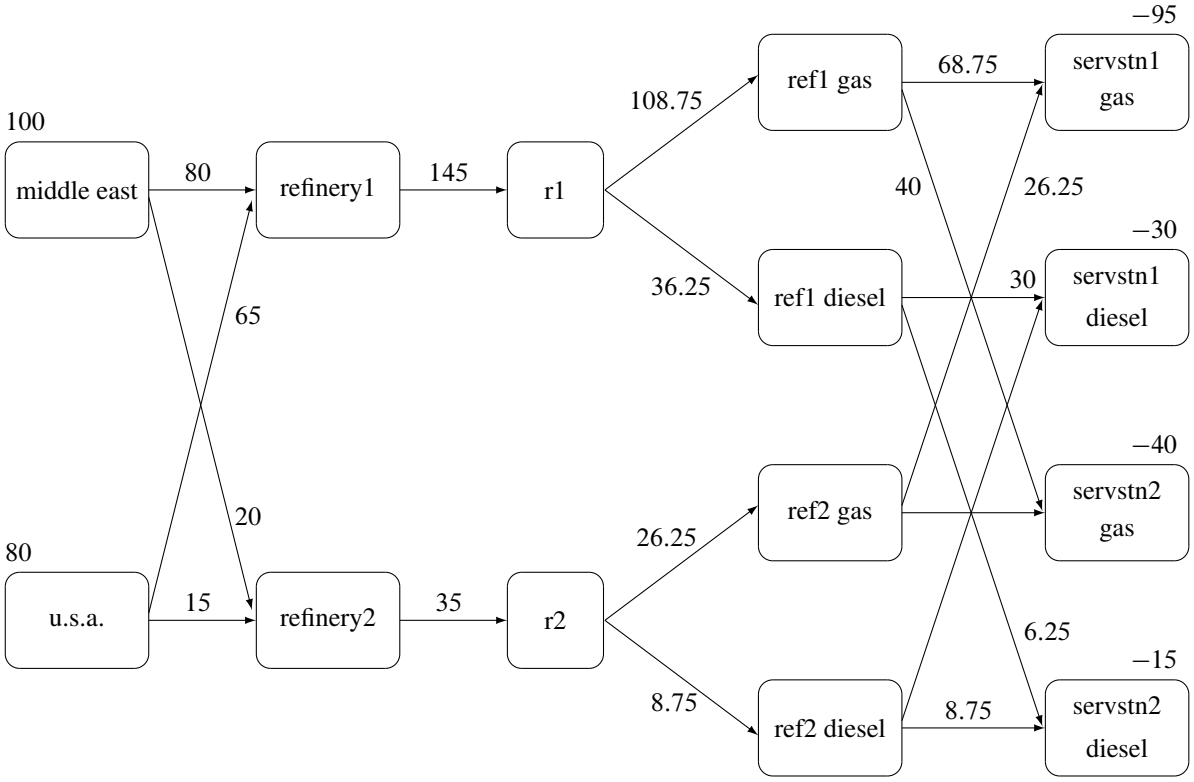
Figure 6.7 CONOUT=SOLUTION

Constrained Optimum

Obs	_from_	_to_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	refinery 1	r1	200	175	50	thruput1	.	.	145.00	29000.00	.
2	refinery 2	r2	220	100	35	thruput2	.	.	35.00	7700.00	29
3	r1	ref1 diesel	0	75	0		.	.	36.25	0.00	.
4	r1	ref1 gas	0	140	0	r1_gas	.	.	108.75	0.00	.
5	r2	ref2 diesel	0	75	0		.	.	8.75	0.00	.
6	r2	ref2 gas	0	100	0	r2_gas	.	.	26.25	0.00	.
7	middle east	refinery 1	63	95	20	m_e_ref1	100	.	80.00	5040.00	.
8	u.s.a.	refinery 1	55	99999999	0		80	.	65.00	3575.00	.
9	middle east	refinery 2	81	80	10	m_e_ref2	100	.	20.00	1620.00	.
10	u.s.a.	refinery 2	49	99999999	0		80	.	15.00	735.00	.
11	ref1 diesel	servstn1 diesel	18	99999999	0		.	30	30.00	540.00	.
12	ref2 diesel	servstn1 diesel	36	99999999	0		.	30	0.00	0.00	12
13	ref1 gas	servstn1 gas	15	70	0		.	95	68.75	1031.25	.
14	ref2 gas	servstn1 gas	17	35	5		.	95	26.25	446.25	.
15	ref1 diesel	servstn2 diesel	17	99999999	0		.	15	6.25	106.25	.
16	ref2 diesel	servstn2 diesel	23	99999999	0		.	15	8.75	201.25	.
17	ref1 gas	servstn2 gas	22	60	0		.	40	40.00	880.00	.
18	ref2 gas	servstn2 gas	31	99999999	0		.	40	0.00	0.00	7
										50875.00	

Notice that, in CONOUT=SOLUTION (Figure 6.7), the optimal flow through each arc in the network is given in the variable named _FLOW_, and the cost of flow through each arc is given in the variable _FCOST_.

Figure 6.8 Oil Industry Solution



Syntax: NETFLOW Procedure

Below are statements used in PROC NETFLOW, listed in alphabetical order as they appear in the text that follows.

PROC NETFLOW *options* ;
CAPACITY *variable* ;
COEF *variables* ;
COLUMN *variable* ;
CONOPT *;* ;
COST *variable* ;
DEMAND *variable* ;
HEADNODE *variable* ;
ID *variables* ;
LO *variable* ;
NAME *variable* ;
NODE *variable* ;
PIVOT *;* ;
PRINT *options* ;
QUIT *;* ;
RESET *options* ;
RHS *variables* ;
ROW *variables* ;
RUN *;* ;
SAVE *options* ;
SHOW *options* ;
SUPDEM *variable* ;
SUPPLY *variable* ;
TAILNODE *variable* ;
TYPE *variable* ;
VAR *variables* ;

Functional Summary

The following table outlines the options available for the NETFLOW procedure classified by function.

Table 6.1 Functional Summary

Description	Statement	Option
Input Data Set Options:		
Arcs input data set	PROC NETFLOW	ARCDATA=
Nodes input data set	PROC NETFLOW	NODEDATA=
Constraint input data set	PROC NETFLOW	CONDATA=
Output Data Set Options:		
Unconstrained primal solution data set	PROC NETFLOW	ARCOUT=

Description	Statement	Option
Unconstrained dual solution data set	PROC NETFLOW	NODEOUT=
Constrained primal solution data set	PROC NETFLOW	CONOUT=
Constrained dual solution data set	PROC NETFLOW	DUALOUT=
Convert sparse or dense format input data set into MPS format output data set	PROC NETFLOW	MPSOUT=
Data Set Read Options:		
CONDATA has sparse data format	PROC NETFLOW	SPARSECONDATA
Default constraint type	PROC NETFLOW	DEFCONTYPE=
Special COLUMN variable value	PROC NETFLOW	TYPEOBS=
Special COLUMN variable value	PROC NETFLOW	RHSOBS=
Used to interpret arc and nonarc variable names	PROC NETFLOW	NAMECTRL=
No new nonarc variables	PROC NETFLOW	SAME_NONARC_DATA
No nonarc data in ARCDATA	PROC NETFLOW	ARCS_ONLY_ARCDATA
Data for an arc found once in ARCDATA	PROC NETFLOW	ARC_SINGLE_OBS
Data for a constraint found once in CONDATA	PROC NETFLOW	CON_SINGLE_OBS
Data for a coefficient found once in CONDATA	PROC NETFLOW	NON_REPLIC=
Data are grouped, exploited during data read	PROC NETFLOW	GROUPED=
Problem Size Specification Options:		
Approximate number of nodes	PROC NETFLOW	NNODES=
Approximate number of arcs	PROC NETFLOW	NARCS=
Approximate number of nonarc variables	PROC NETFLOW	NNAS=
Approximate number of coefficients	PROC NETFLOW	NCOEFS=
Approximate number of constraints	PROC NETFLOW	NCONS=
Network Options:		
Default arc cost	PROC NETFLOW	DEFCOST=
Default arc capacity	PROC NETFLOW	DEFCAPACITY=
Default arc lower flow bound	PROC NETFLOW	DEFMINFLOW=
Network's only supply node	PROC NETFLOW	SOURCE=
SOURCE's supply capability	PROC NETFLOW	SUPPLY=
Network's only demand node	PROC NETFLOW	SINK=
SINK's demand	PROC NETFLOW	DEMAND=
Convey excess supply/demand through network	PROC NETFLOW	THRUNET
Find maximal flow between SOURCE and SINK	PROC NETFLOW	MAXFLOW
Cost of bypass arc for MAXFLOW problem	PROC NETFLOW	BYPASSDIVIDE=
Find shortest path from SOURCE to SINK	PROC NETFLOW	SHORTPATH
Specify generalized networks	PROC NETFLOW	GENNET
Specify excess demand or supply	PROC NETFLOW	EXCESS=
Memory Control Options:		
Issue memory usage messages to SAS log	PROC NETFLOW	MEMREP
Number of bytes to use for main memory	PROC NETFLOW	BYTES=
Proportion of memory for arrays	PROC NETFLOW	COREFACTOR=

Description	Statement	Option
Memory allocated for LU factors	PROC NETFLOW	DWIA=
Linked list for updated column	PROC NETFLOW	SPARSEP2
Use 2-dimensional array for basis matrix	PROC NETFLOW	INVD_2D
Maximum bytes for a single array	PROC NETFLOW	MAXARRAYBYTES=
Simplex Options:		
Use big-M instead of two-phase method, stage 1	RESET	BIGM1
Use Big-M instead of two-phase method, stage 2	RESET	BIGM2
Anti-cycling option	RESET	CYCLEMULT1=
Interchange first nonkey with leaving key arc	RESET	INTFIRST
Controls working basis matrix inversions	RESET	INVFREQ=
Maximum number of L row operations allowed before refactorization	RESET	MAXL=
Maximum number of LU factor column updates	RESET	MAXLUUPDATES=
Anti-cycling option	RESET	MINBLOCK1=
Use first eligible leaving variable, stage 1	RESET	LRATIO1
Use first eligible leaving variable, stage 2	RESET	LRATIO2
Negates INTFIRST	RESET	NOINTFIRST
Negates LRATIO1	RESET	NOLRATIO1
Negates LRATIO2	RESET	NOLRATIO2
Negates PERTURB1	RESET	NOPERTURB1
Anti-cycling option	RESET	PERTURB1
Controls working basis matrix refactorization	RESET	REFACTFREQ=
Use two-phase instead of big-M method, stage 1	RESET	TWOPHASE1
Use two-phase instead of big-M method, stage 2	RESET	TWOPHASE2
Pivot element selection parameter	RESET	U=
Zero tolerance, stage 1	RESET	ZERO1=
Zero tolerance, stage 2	RESET	ZERO2=
Zero tolerance, real number comparisons	RESET	ZEROTOL=
Pricing Options:		
Frequency of dual value calculation	RESET	DUALFREQ=
Pricing strategy, stage 1	RESET	PRICETYPE1=
Pricing strategy, stage 2	RESET	PRICETYPE2=
Used when P1SCAN=PARTIAL	RESET	P1NPARTIAL=
Controls search for entering candidate, stage 1	RESET	P1SCAN=
Used when P2SCAN=PARTIAL	RESET	P2NPARTIAL=
Controls search for entering candidate, stage 2	RESET	P2SCAN=
Initial queue size, stage 1	RESET	QSIZE1=
Initial queue size, stage 2	RESET	QSIZE2=
Used when Q1FILLSCAN=PARTIAL	RESET	Q1FILLNPARTIAL=
Controls scan when filling queue, stage 1	RESET	Q1FILLSCAN=
Used when Q2FILLSCAN=PARTIAL	RESET	Q2FILLNPARTIAL=
Controls scan when filling queue, stage 2	RESET	Q2FILLSCAN=
Queue size reduction factor, stage 1	RESET	REDUCEQSIZE1=

Description	Statement	Option
Queue size reduction factor, stage 2	RESET	REDUCEQSIZE2=
Frequency of refreshing queue, stage 1	RESET	REFRESHQ1=
Frequency of refreshing queue, stage 2	RESET	REFRESHQ2=
Optimization Termination Options:		
Pause after stage 1; do not start stage 2	RESET	ENDPAUSE1
Pause when feasible, stage 1	RESET	FEASIBLEPAUSE1
Pause when feasible, stage 2	RESET	FEASIBLEPAUSE2
Maximum number of iterations, stage 1	RESET	MAXIT1=
Maximum number of iterations, stage 2	RESET	MAXIT2=
Negates ENDPAUSE1	RESET	NOENDPAUSE1
Negates FEASIBLEPAUSE1	RESET	NOFEASIBLEPAUSE1
Negates FEASIBLEPAUSE2	RESET	NOFEASIBLEPAUSE2
Pause every PAUSE1 iterations, stage 1	RESET	PAUSE1=
Pause every PAUSE2 iterations, stage 2	RESET	PAUSE2=
Interior Point Algorithm Options:		
Use interior point algorithm	PROC NETFLOW	INTPOINT
Factorization method	RESET	FACT_METHOD=
Allowed amount of dual infeasibility	RESET	TOLDINF=
Allowed amount of primal infeasibility	RESET	TOLPINF=
Allowed total amount of dual infeasibility	RESET	TOLTOTDINF=
Allowed total amount of primal infeasibility	RESET	TOLTOTPINF=
Cut-off tolerance for Cholesky factorization	RESET	CHOLTINYTOL=
Density threshold for Cholesky processing	RESET	DENSETHR=
Step-length multiplier	RESET	PDSTEPMULT=
Preprocessing type	RESET	PRSLTYPE=
Print optimization progress on SAS log	RESET	PRINTLEVEL2=
Interior Point Stopping Criteria Options:		
Maximum number of interior point iterations	RESET	MAXITERB=
Primal-dual (duality) gap tolerance	RESET	PDGAPTOL=
Stop because of complementarity	RESET	STOP_C=
Stop because of duality gap	RESET	STOP_DG=
Stop because of $infeas_b$	RESET	STOP_IB=
Stop because of $infeas_c$	RESET	STOP_IC=
Stop because of $infeas_d$	RESET	STOP_ID=
Stop because of complementarity	RESET	AND_STOP_C=
Stop because of duality gap	RESET	AND_STOP_DG=
Stop because of $infeas_b$	RESET	AND_STOP_IB=
Stop because of $infeas_c$	RESET	AND_STOP_IC=
Stop because of $infeas_d$	RESET	AND_STOP_ID=
Stop because of complementarity	RESET	KEEPGOING_C=
Stop because of duality gap	RESET	KEEPGOING_DG=
Stop because of $infeas_b$	RESET	KEEPGOING_IB=

Description	Statement	Option
Stop because of $infeas_c$	RESET	KEEPGOING_IC=
Stop because of $infeas_d$	RESET	KEEPGOING_ID=
Stop because of complementarity	RESET	AND_KEEPGOING_C=
Stop because of duality gap	RESET	AND_KEEPGOING_DG=
Stop because of $infeas_b$	RESET	AND_KEEPGOING_IB=
Stop because of $infeas_c$	RESET	AND_KEEPGOING_IC=
Stop because of $infeas_d$	RESET	AND_KEEPGOING_ID=
PRINT Statement Options:		
Display everything	PRINT	PROBLEM
Display arc information	PRINT	ARCS
Display nonarc variable information	PRINT	NONARCS
Display variable information	PRINT	VARIABLES
Display constraint information	PRINT	CONSTRAINTS
Display information for some arcs	PRINT	SOME_ARCS
Display information for some nonarc variables	PRINT	SOME_NONARCS
Display information for some variables	PRINT	SOME_VARIABLES
Display information for some constraints	PRINT	SOME_CONS
Display information for some constraints associated with some arcs	PRINT	CON_ARCS
Display information for some constraints associated with some nonarc variables	PRINT	CON_NONARCS
Display information for some constraints associated with some variables	PRINT	CON_VARIABLES
PRINT Statement Qualifiers:		
Produce a short report	PRINT	/ SHORT
Produce a long report	PRINT	/ LONG
Display arcs/variables with zero flow/value	PRINT	/ ZERO
Display arcs/variables with nonzero flow/value	PRINT	/ NONZERO
Display basic arcs/variables	PRINT	/ BASIC
Display nonbasic arcs/variables	PRINT	/ NONBASIC
SHOW Statement Options:		
Show problem, optimization status	SHOW	STATUS
Show network model parameters	SHOW	NETSTMT
Show data sets that have been or will be created	SHOW	DATASETS
Show options that pause optimization	SHOW	PAUSE
Show simplex algorithm options	SHOW	SIMPLEX
Show pricing strategy options	SHOW	PRICING
Show miscellaneous options	SHOW	MISC
SHOW Statement Qualifiers:		
Display information only on relevant options	SHOW	/ RELEVANT
Display options for current stage only	SHOW	/ STAGE

Description	Statement	Option
Miscellaneous Options:		
Infinity value	PROC NETFLOW	INFINITY=
Scale constraint row, nonarc variable column coefficients, or both	PROC NETFLOW	SCALE=
Maximization instead of minimization	PROC NETFLOW	MAXIMIZE
Use warm start solution	PROC NETFLOW	WARM
All-artificial starting solution	PROC NETFLOW	ALLART
Output complete basis information to ARCOUT= and NODEOUT= data sets	RESET	FUTURE1
Output complete basis information to CONOUT= and DUALOUT= data sets	RESET	FUTURE2
Turn off infeasibility or optimality flags	RESET	MOREOPT
Negates FUTURE1	RESET	NOFUTURE1
Negates FUTURE2	RESET	NOFUTURE2
Negates SCRATCH	RESET	NOSCRATCH
Negates ZTOL1	RESET	NOZTOL1
Negates ZTOL2	RESET	NOZTOL2
Write optimization time to SAS log	RESET	OPTIM_TIMER
No stage 1 optimization; do stage 2 optimization	RESET	SCRATCH
Suppress similar SAS log messages	RESET	VERBOSE=
Use zero tolerance, stage 1	RESET	ZTOL1
Use zero tolerance, stage 2	RESET	ZTOL2

Interactivity

PROC NETFLOW can be used interactively. You begin by giving the PROC NETFLOW statement, and you must specify the ARCDATA= data set. The CONDATA= data set must also be specified if the problem has side constraints. If necessary, specify the NODEDATA= data set.

The variable lists should be given next. If you have variables in the input data sets that have special names (for example, a variable in the ARCDATA= data set named _TAIL_ that has tail nodes of arcs as values), it may not be necessary to have many or any variable lists.

The CONOPT, PIVOT, PRINT, QUIT, SAVE, SHOW, RESET, and RUN statements follow and can be listed in any order. The CONOPT and QUIT statements can be used only once. The others can be used as many times as needed.

Use the RESET or SAVE statement to change the names of the output data sets. With RESET, you can also indicate the reasons why optimization should stop (for example, you can indicate the maximum number of stage 1 or stage 2 iterations that can be performed). PROC NETFLOW then has a chance to either execute the next statement, or, if the next statement is one that PROC NETFLOW does not recognize (the next PROC or DATA step in the SAS session), do any allowed optimization and finish. If no new statement has been

submitted, you are prompted for one. Some options of the RESET statement enable you to control aspects of the primal simplex algorithm. Specifying certain values for these options can reduce the time it takes to solve a problem. Note that any of the RESET options can be specified in the PROC NETFLOW statement.

The RUN statement starts or resumes optimization. The PIVOT statement makes PROC NETFLOW perform one simplex iteration. The QUIT statement immediately stops PROC NETFLOW. The CONOPT statement forces PROC NETFLOW to consider constraints when it next performs optimization. The SAVE statement has options that enable you to name output data sets; information about the current solution is put in these output data sets. Use the SHOW statement if you want to examine the values of options of other statements. Information about the amount of optimization that has been done and the STATUS of the current solution can also be displayed using the SHOW statement.

The PRINT statement instructs PROC NETFLOW to display parts of the problem. PRINT ARCS produces information on all arcs. PRINT SOME_ARCS limits this output to a subset of arcs. There are similar PRINT statements for nonarc variables and constraints:

```
print nonarcs;
print some_nonarcs;
print constraints;
print some_cons;
```

PRINT CON_ARCS enables you to limit constraint information that is obtained to members of a set of arcs that have nonzero constraint coefficients in a set of constraints. PRINT CON_NONARCS is the corresponding statement for nonarc variables.

For example, an interactive PROC NETFLOW run might look something like this:

```
proc netflow arcdata=data set
      other options;
   variable list specifications; /* if necessary */
   reset options;
   print options;    /* look at problem                  */
   run;              /* do some optimization             */
   /* suppose that optimization stopped for  */
   /* some reason or you manually stopped it */
   print options;    /* look at the current solution     */
   save options;     /* keep current solution            */
   show options;     /* look at settings                 */
   reset options;    /* change some settings, those that */
                     /* caused optimization to stop      */
   run;              /* do more optimization             */
   print options;    /* look at the optimal solution     */
   save options;     /* keep optimal solution            */
```

If you are interested only in finding the optimal solution, have used SAS variables that have special names in the input data sets, and want to use default settings for everything, then the following statement is all you need:

PROC NETFLOW ARCDATA= *data set* ;

PROC NETFLOW Statement

PROC NETFLOW *options* **;**

This statement invokes the procedure. The following options and the options listed with the RESET statement can appear in the PROC NETFLOW statement.

Data Set Options

This section briefly describes all the input and output data sets used by PROC NETFLOW. The ARCDATA= data set, NODEDATA= data set, and CONDATA= data set can contain SAS variables that have special names, for instance _CAPAC_, _COST_, and _HEAD_. PROC NETFLOW looks for such variables if you do not give explicit variable list specifications. If a SAS variable with a special name is found and that SAS variable is not in another variable list specification, PROC NETFLOW determines that values of the SAS variable are to be interpreted in a special way. By using SAS variables that have special names, you may not need to have any variable list specifications.

ARCDATA=*SAS-data-set*

names the data set that contains arc and, optionally, nonarc variable information and nodal supply/demand data. The ARCDATA= data set must be specified in all PROC NETFLOW statements.

ARCOUT=*SAS-data-set*

AOUT=*SAS-data-set*

names the output data set that receives all arc and nonarc variable data, including flows or values, and other information concerning the unconstrained optimal solution. The supply and demand information can also be found in the ARCOUT= data set. Once optimization that considers side constraints starts, you are not able to obtain an ARCOUT= data set. Instead, use the CONOUT= data set to get the current solution. See the section "ARCOUT= and CONOUT= Data Sets" on page 389 for more information.

CONDATA=*SAS-data-set*

names the data set that contains the side constraint data. The data set can also contain other data such as arc costs, capacities, lower flow bounds, nonarc variable upper and lower bounds, and objective function coefficients. PROC NETFLOW needs a CONDATA= data set to solve a constrained problem or a linear programming problem. See the section "CONDATA= Data Set" on page 380 for more information.

CONOUT=*SAS-data-set*

COUT=*SAS-data-set*

names the output data set that receives an optimal primal solution to the problem obtained by performing optimization that considers the side constraints. See the section "ARCOUT= and CONOUT= Data Sets" on page 389 for more information.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

names the output data set that receives an optimal dual solution to the problem obtained by performing optimization that considers the side constraints. See the section "NODEOUT= and DUALOUT= Data Sets" on page 390 for more information.

NODEDATA=*SAS-data-set*

DUALIN=*SAS-data-set*

names the data set that contains the node supply and demand specifications. You do not need observations in the NODEDATA= data set for trans-shipment nodes. (Trans-shipment nodes neither supply nor demand flow.) All nodes are assumed to be trans-shipment nodes unless supply or demand data indicate otherwise. It is acceptable for some arcs to be directed toward supply nodes or away from demand nodes.

The use of the NODEDATA= data set is optional in the PROC NETFLOW statement provided that, if the NODEDATA= data set is not used, supply and demand details are specified by other means. Other means include using the MAXFLOW or SHORTPATH option, SUPPLY or DEMAND list variables (or both) in the ARCDATA= data set, and the SOURCE=, SUPPLY=, SINK=, or DEMAND= option in the PROC NETFLOW statement.

NODEOUT=*SAS-data-set*

names the output data set that receives all information about nodes (supply and demand and nodal dual variable values) and other information concerning the optimal solution found by the optimizer when neglecting side constraints. Once optimization that considers side constraints starts, you are not able to obtain a NODEOUT= data set. Instead, use the DUALOUT= data set to get the current solution dual information. See the section "NODEOUT= and DUALOUT= Data Sets" on page 390 for a more complete description.

MPSOUT=*SAS-data-set*

names the SAS data set that contains converted sparse or dense format input data in MPS format. Invoking this option directs the NETFLOW procedure to halt before attempting optimization. For more information about the MPSOUT= option, see the section "Converting Any PROC NETFLOW Format to an MPS-Format SAS Data Set" on page 391. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*),.

General Options

The following is a list of options you can use with PROC NETFLOW. The options are listed in alphabetical order.

ALLART

indicates that PROC NETFLOW uses an all artificial initial solution (Kennington and Helgason 1980, p. 68) instead of the default *good path* method for determining an initial solution (Kennington and Helgason 1980, p. 245). The ALLART initial solution is generally not as good; more iterations are usually required before the optimal solution is obtained. However, because less time is used when setting up an ALLART start, it can offset the added expenditure of CPU time in later computations.

ARCS_ONLY_ARCDATA

indicates that data for only arcs are in the ARCDATA= data set. When PROC NETFLOW reads the data in ARCDATA= data set, memory would not be wasted to receive data for nonarc variables. The read might then be performed faster. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

ARC_SINGLE_OBS

indicates that for all arcs and nonarc variables, data for each arc or nonarc variable is found in only one observation of the ARCDATA= data set. When reading the data in the ARCDATA= data set, PROC NETFLOW knows that the data in an observation is for an arc or a nonarc variable that has not had data previously read that needs to be checked for consistency. The read might then be performed faster.

If you specify ARC_SINGLE_OBS, PROC NETFLOW automatically works as if GROUPED=ARCDATA is also specified. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

BYPASSDIVIDE=*b*

BYPASSDIV=*b*

BPD=*b*

should be used only when the MAXFLOW option has been specified; that is, PROC NETFLOW is solving a maximal flow problem. PROC NETFLOW prepares to solve maximal flow problems by setting up a bypass arc. This arc is directed from the SOURCE to the SINK and will eventually convey flow equal to INFINITY minus the maximal flow through the network. The cost of the bypass arc must be expensive enough to drive flow through the network, rather than through the bypass arc. However, the cost of the bypass arc must be less than the cost of artificial variables (otherwise these might have nonzero optimal value and a false infeasibility error will result). Also, the cost of the bypass arc must be greater than the eventual total cost of the maximal flow, which can be nonzero if some network arcs have nonzero costs. The cost of the bypass is set to the value of the INFINITY= option. Valid values for the BYPASSDIVIDE= option must be greater than or equal to 1.1.

If there are no nonzero costs of arcs in the MAXFLOW problem, the cost of the bypass arc is set to 1.0 (-1.0 if maximizing) if you do not specify the BYPASSDIVIDE= option. The reduced costs in the ARCOUT= data set and the CONOUT= data set will correctly reflect the value that would be added to the maximal flow if the capacity of the arc is increased by one unit. If there are nonzero costs, or if you specify the BYPASSDIVIDE= option, the reduced costs may be contaminated by the cost of the bypass arc and no economic interpretation can be given to reduced cost values. The default value for the BYPASSDIVIDE= option (in the presence of nonzero arc costs) is 100.0.

BYTES=*b*

indicates the size of the main working memory (in bytes) that PROC NETFLOW will allocate. The default value for the BYTES= option is near to the number of bytes of the largest contiguous memory that can be allocated for this purpose. The working memory is used to store all the arrays and buffers used by PROC NETFLOW. If this memory has a size smaller than what is required to store all arrays and buffers, PROC NETFLOW uses various schemes that page information between memory and disk.

PROC NETFLOW uses more memory than the main working memory. The additional memory requirements cannot be determined at the time when the main working memory is allocated. For example, every time an output data set is created, some additional memory is required. Do not specify a value for the BYTES= option equal to the size of available memory.

CON_SINGLE_OBS

improves how the CONDATA= data set is read. How it works depends on whether the CONDATA has a dense or sparse format.

If CONDATA has the dense format, specifying CON_SINGLE_OBS indicates that, for each constraint, data can be found in only one observation of CONDATA.

If CONDATA has a sparse format, and data for each arc and nonarc variable can be found in only one observation of CONDATA, then specify the CON_SINGLE_OBS option. If there are *n* SAS variables in the ROW and COEF list, then each arc or nonarc can have at most *n* constraint coefficients in the model. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

COREFACTOR=*c*

CF=*c*

enables you to specify the maximum proportion of memory to be used by the arrays frequently accessed by PROC NETFLOW. PROC NETFLOW strives to maintain all information required during optimization in core. If the amount of available memory is not great enough to store the arrays completely in core, either initially or as memory requirements grow, PROC NETFLOW can change the memory management scheme it uses. Large problems can still be solved. When necessary, PROC NETFLOW transfers data from random access memory (RAM) or core that can be accessed quickly but is of limited size to slower access large capacity disk memory. This is called *paging*.

Some of the arrays and buffers used during constrained optimization either vary in size, are not required as frequently as other arrays, or are not required throughout the simplex iteration. Let *a* be the amount of memory in bytes required to store frequently accessed arrays of nonvarying size. Specify the MEMREP option in the PROC NETFLOW statement to get the value for *a* and a report of memory usage. If the size of the main working memory BYTES=*b* multiplied by COREFACTOR=*c* is greater than *a*, PROC NETFLOW keeps the frequently accessed arrays of nonvarying size resident in core throughout the optimization. If the other arrays cannot fit into core, they are paged in and out of the remaining part of the main working memory.

If *b* multiplied by *c* is less than *a*, PROC NETFLOW uses a different memory scheme. The working memory is used to store only the arrays needed in the part of the algorithm being executed. If necessary, these arrays are read from disk into the main working area. Paging, if required, is done for all these arrays, and sometimes information is written back to disk at the end of that part of the algorithm. This memory scheme is not as fast as the other memory schemes. However, problems can be solved with memory that is too small to store every array.

PROC NETFLOW is capable of solving very large problems in a modest amount of available memory. However, as more time is spent doing input/output operations, the speed of PROC NETFLOW decreases. It is important to choose the value of the COREFACTOR= option carefully. If the value is too small, the memory scheme that needs to be used might not be as efficient as another that could have been used had a larger value been specified. If the value is too large, too much of the main working memory is occupied by the frequently accessed, nonvarying sized arrays, leaving too little for the other arrays. The amount of input/output operations for these other arrays can be so high that another memory scheme might have been used more beneficially.

The valid values of COREFACTOR=*c* are between 0.0 and 0.95, inclusive. The default value for *c* is 0.75 when there are over 200 side constraints, and 0.9 when there is only one side constraint. When the problem has between 2 and 200 constraints, the value of *c* lies between the two points (1, 0.9) and (201, 0.75).

DEFCAPACITY=*c*

DC=*c*

requests that the default arc capacity and the default nonarc variable value upper bound be *c*. If this option is not specified, then DEFCAPACITY= INFINITY.

DEFCONTYPE=*c*

DEFTYPE=*c*

DCT=*c*

specifies the default constraint type. This default constraint type is either *less than or equal to* or is the type indicated by DEFCONTYPE=*c*. Valid values for this option are

LE, le, <=	for *less than or equal to*
EQ, eq, =	for *equal to*
GE, ge, >=	for *greater than or equal to*

The values do not need to be enclosed in quotes.

DEFCOST=*c*

requests that the default arc cost and the default nonarc variable objective function coefficient be *c*. If this option is not specified, then DEFCOST=0.0.

DEFMINFLOW=*m*

DMF=*m*

requests that the default lower flow bound through arcs and the default lower value bound of nonarc variables be *m*. If a value is not specified, then DEFMINFLOW=0.0.

DEMAND=*d*

specifies the demand at the SINK node specified by the SINK= option. The DEMAND= option should be used only if the SINK= option is given in the PROC NETFLOW statement and neither the SHORTPATH option nor the MAXFLOW option is specified. If you are solving a minimum cost network problem and the SINK= option is used to identify the sink node, but the DEMAND= option is not specified, then the demand at the sink node is made equal to the network's total supply.

DWIA=*i*

controls the initial amount of memory to be allocated to store the **LU** factors of the working basis matrix. DWIA stands for D_W *initial allocation* and *i* is the number of nonzeros and matrix row operations in the **LU** factors that can be stored in this memory. Due to fill-in in the **U** factor and the growth in the number of row operations, it is often necessary to move information about elements of a particular row or column to another location in the memory allocated for the **LU** factors. This process leaves some memory temporarily unoccupied. Therefore, DWIA=*i* must be greater than the memory required to store only the **LU** factors.

Occasionally, it is necessary to compress the **U** factor so that it again occupies contiguous memory. Specifying too large a value for DWIA means that more memory is required by PROC NETFLOW. This might cause more expensive memory mechanisms to be used than if a smaller but adequate value had been specified for DWIA=. Specifying too small a value for the DWIA= option can make time-consuming compressions more numerous. The default value for the DWIA= option is eight times the number of side constraints.

EXCESS=*option*

enables you to specify how to handle excess supply or demand in a network, if it exists.

For pure networks EXCESS=ARCS and EXCESS=SLACKS are valid options. By default EXCESS=ARCS is used. Note that if you specify EXCESS=SLACKS, then the interior point solver is

used and you need to specify the output data set using the CONOUT= data set. For more details see the section "Using the New EXCESS= Option in Pure Networks: NETFLOW Procedure" on page 454.

For generalized networks you can either specify EXCESS=DEMAND or EXCESS=SUPPLY to indicate that the network has excess demand or excess supply, respectively. For more details see the section "Using the New EXCESS= Option in Generalized Networks: NETFLOW Procedure" on page 461.

GENNET

This option is necessary if you need to solve a generalized network flow problem and there are no arc multipliers specified in the ARCDATA= data set.

GROUPED=*grouped*

PROC NETFLOW can take a much shorter time to read data if the data have been grouped prior to the PROC NETFLOW call. This enables PROC NETFLOW to conclude that, for instance, a new NAME list variable value seen in an ARCDATA= data set grouped by the values of the NAME list variable before PROC NETFLOW was called is new. PROC NETFLOW does not need to check that the NAME has been read in a previous observation. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

- GROUPED=ARCDATA indicates that the ARCDATA= data set has been grouped by values of the NAME list variable. If _NAME_ is the name of the NAME list variable, you could use PROC SORT DATA=ARCDATA; BY _NAME_; prior to calling PROC NETFLOW. Technically, you do not have to sort the data, only ensure that all similar values of the NAME list variable are grouped together. If you specify the ARCS_ONLY_ARCDATA option, PROC NETFLOW automatically works as if GROUPED=ARCDATA is also specified.
- GROUPED=CONDATA indicates that the CONDATA= data set has been grouped.

  If the CONDATA= data set has a dense format, GROUPED=CONDATA indicates that the CONDATA= data set has been grouped by values of the ROW list variable. If _ROW_ is the name of the ROW list variable, you could use PROC SORT DATA=CONDATA; BY _ROW_; prior to calling PROC NETFLOW. Technically, you do not have to sort the data, only ensure that all similar values of the ROW list variable are grouped together. If you specify the CON_SINGLE_OBS option, or if there is no ROW list variable, PROC NETFLOW automatically works as if GROUPED=CONDATA has been specified.

  If the CONDATA= data set has the sparse format, GROUPED=CONDATA indicates that the CONDATA= data set has been grouped by values of the COLUMN list variable. If _COL_ is the name of the COLUMN list variable, you could use PROC SORT DATA=CONDATA; BY _COL_; prior to calling PROC NETFLOW. Technically, you do not have to sort the data, only ensure that all similar values of the COLUMN list variable are grouped together.
- GROUPED=BOTH indicates that both GROUPED=ARCDATA and GROUPED=CONDATA are TRUE.
- GROUPED=NONE indicates that the data sets have not been grouped, that is, neither GROUPED=ARCDATA nor GROUPED=CONDATA is TRUE. This is the default, but it is much better if GROUPED=ARCDATA, or GROUPED=CONDATA, or GROUPED=BOTH.

A data set like

```
... _XXXXX_ ....
      bbb
      bbb
      aaa
      ccc
      ccc
```

is a candidate for the GROUPED= option. Similar values are grouped together. When PROC NETFLOW is reading the ith observation, either the value of the _XXXXX_ variable is the same as the $(i-1)$st (that is, the previous observation's) _XXXXX_ value, or it is a new _XXXXX_ value not seen in any previous observation. This also means that if the ith _XXXXX_ value is different from the $(i-1)$st _XXXXX_ value, the value of the $(i-1)$st _XXXXX_ variable will not be seen in any observations $i, i+1, \ldots$.

INFINITY=*i*

INF=*i*

is the largest number used by PROC NETFLOW in computations. A number too small can adversely affect the solution process. You should avoid specifying an enormous value for the INFINITY= option because numerical roundoff errors can result. If a value is not specified, then INFINITY=99999999. The INFINITY= option cannot be assigned a value less than 9999.

INTPOINT

indicates that the interior point algorithm is to be used. The INTPOINT option must be specified if you want the interior point algorithm to be used for solving network problems, otherwise the simplex algorithm is used instead. For linear programming problems (problems with no network component), PROC NETFLOW must use the interior point algorithm, so you need not specify the INTPOINT option.

INVD_2D

controls the way in which the inverse of the working basis matrix is stored. How this matrix is stored affects computations as well as how the working basis or its inverse is updated. The working basis matrix is defined in the section "Details: NETFLOW Procedure" on page 380. If INVD_2D is specified, the working basis matrix inverse is stored as a matrix. Typically, this memory scheme is best when there are few side constraints or when the working basis is dense.

If INVD_2D is not specified, lower (**L**) and upper (**U**) factors of the working basis matrix are used. **U** is an upper triangular matrix and **L** is a lower triangular matrix corresponding to a sequence of elementary matrix row operations. The sparsity-exploiting variant of the Bartels-Golub decomposition is used to update the **LU** factors. This scheme works well when the side constraint coefficient matrix is sparse or when many side constraints are nonbinding.

MAXARRAYBYTES=*m*

specifies the maximum number of bytes an individual array can occupy. This option is of most use when solving large problems and the amount of available memory is insufficient to store all arrays at once. Specifying the MAXARRAYBYTES= option ensures that arrays that need a large amount of memory do not consume too much memory at the expense of other arrays.

There is one array that contains information about nodes and the network basis spanning tree description. This tree description enables computations involving the network part of the basis to be performed

very quickly and is the reason why PROC NETFLOW is more suited to solving constrained network problems than PROC LP. It is beneficial that this array be stored in core when possible, otherwise this array must be paged, slowing down the computations. Try not to specify a MAXARRAYBYTES=*m* value smaller than the amount of memory needed to store the main node array. You are told what this memory amount is on the SAS log if you specify the MEMREP option in the PROC NETFLOW statement.

MAXFLOW

MF

specifies that PROC NETFLOW solve a maximum flow problem. In this case, the PROC NETFLOW procedure finds the maximum flow from the node specified by the SOURCE= option to the node specified by the SINK= option. PROC NETFLOW automatically assigns an INFINITY= option supply to the SOURCE= option node and the SINK= option is assigned the INFINITY= option demand. In this way, the MAXFLOW option sets up a maximum flow problem as an equivalent minimum cost problem.

You can use the MAXFLOW option when solving any flow problem (not necessarily a maximum flow problem) when the network has one supply node (with infinite supply) and one demand node (with infinite demand). The MAXFLOW option can be used in conjunction with all other options (except SHORTPATH, SUPPLY=, and DEMAND=) and capabilities of PROC NETFLOW.

MAXIMIZE

MAX

specifies that PROC NETFLOW find the maximum cost flow through the network. If both the MAXIMIZE and the SHORTPATH options are specified, the solution obtained is the longest path between the SOURCE= and SINK= nodes. Similarly, MAXIMIZE and MAXFLOW together cause PROC NETFLOW to find the minimum flow between these two nodes; this is zero if there are no nonzero lower flow bounds.

MEMREP

indicates that information on the memory usage and paging schemes (if necessary) is reported by PROC NETFLOW on the SAS log. As optimization proceeds, you are informed of any changes in the memory requirements and schemes used by PROC NETFLOW.

NAMECTRL=*i*

is used to interpret arc and nonarc variable names in the CONDATA= data set.

In the ARCDATA= data set, an arc is identified by its tail and head node. In the CONDATA= data set, arcs are identified by names. You can give a name to an arc by having a NAME list specification that indicates a SAS variable in the ARCDATA= data set that has names of arcs as values.

PROC NETFLOW requires arcs that have information about them in the CONDATA= data set to have names, but arcs that do not have information about them in the CONDATA= data set can also have names. Unlike a nonarc variable whose name uniquely identifies it, an arc can have several different names. An arc has a default name in the form *tail_head*, that is, the name of the arc's tail node followed by an underscore and the name of the arc's head node.

In the CONDATA= data set, if the dense data format is used, (described in the section "CONDATA= Data Set" on page 380) a name of an arc or a nonarc variable is the *name* of a SAS variable listed in the VAR list specification. If the sparse data format of the CONDATA= data set is used, a name of an arc or a nonarc variable is a *value* of the SAS variable listed in the COLUMN list specification.

The NAMECTRL= option is used when a name of an arc or nonarc variable in the CONDATA= data set (either a VAR list SAS variable name or value of the COLUMN list SAS variable) is in the form *tail_head* and there exists an arc with these end nodes. If *tail_head* has not already been tagged as belonging to an arc or nonarc variable in the ARCDATA= data set, PROC NETFLOW needs to know whether *tail_head* is the name of the arc or the name of a nonarc variable.

If you specify NAMECTRL=1, a name that is not defined in the ARCDATA= data set is assumed to be the name of a nonarc variable. NAMECTRL=2 treats *tail_head* as the name of the arc with these endnodes, provided no other name is used to associate data in the CONDATA= data set with this arc. If the arc does have other names that appear in the CONDATA= data set, *tail_head* is assumed to be the name of a nonarc variable. If you specify NAMECTRL=3, *tail_head* is assumed to be a name of the arc with these end nodes, whether the arc has other names or not. The default value of NAMECTRL is 3. Note that if you use the dense side constraint input format, the default arc name *tail_head* is not recognized (regardless of the NAMECTRL value) unless the head node and tail node names contain no lowercase letters.

If the dense format is used for the CONDATA= data set, the SAS System converts SAS variable names in a SAS program to uppercase. The VAR list variable names are uppercased. Because of this, PROC NETFLOW automatically uppercases names of arcs and nonarc variables (the values of the NAME list variable) in the ARCDATA= data set. The names of arcs and nonarc variables (the values of the NAME list variable) appear uppercased in the ARCOUT= data set and the CONOUT= data set, and in the PRINT statement output.

Also, if the dense format is used for the CONDATA= data set, be careful with default arc names (names in the form tailnode_headnode). Node names (values in the TAILNODE and HEADNODE list variables) in the ARCDATA= data set are not uppercased by PROC NETFLOW. Consider the following code:

```
data arcdata;
   input _from_ $ _to_ $ _name $ ;
   datalines;
from to1 .
from to2 arc2
TAIL TO3 .
;
data densecon;
   input from_to1 from_to2 arc2 tail_to3;
   datalines;
2 3 3 5
;
proc netflow
   arcdata=arcdata condata=densecon;
   run;
```

The SAS System does not uppercase character string values. PROC NETFLOW never uppercases node names, so the arcs in observations 1, 2, and 3 in the preceding ARCDATA= data set have the default names “from_to1”, “from_to2”, and “TAIL_TO3”, respectively. When the dense format of the CONDATA= data set is used, PROC NETFLOW does uppercase values of the NAME list variable, so the name of the arc in the second observation of the ARCDATA= data set is “ARC2”. Thus, the second arc has two names: its default “from_to2” and the other that was specified “ARC2”.

As the SAS System does uppercase program code, you must think of the input statement

```
input from_to1 from_to2 arc2 tail_to3;
```

as really being

```
INPUT FROM_TO1 FROM_TO2 ARC2 TAIL_TO3;
```

The SAS variables named "FROM_TO1" and "FROM_TO2" are *not* associated with any of the arcs in the preceding ARCDATA= data set. The values "FROM_TO1" and "FROM_TO2" are different from all of the arc names "from_to1", "from_to2", "TAIL_TO3", and "ARC2". "FROM_TO1" and "FROM_TO2" could end up being the names of two nonarc variables. It is sometimes useful to specify PRINT NONARCS; before commencing optimization to ensure that the model is correct (has the right set of nonarc variables).

The SAS variable named "ARC2" is the name of the second arc in the ARCDATA= data set, even though the name specified in the ARCDATA= data set looks like "arc2". The SAS variable named "TAIL_TO3" is the default name of the third arc in the ARCDATA= data set.

NARCS=*n*
specifies the approximate number of arcs. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

NCOEFS=*n*
specifies the approximate number of constraint coefficients. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

NCONS=*n*
specifies the approximate number of constraints. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

NNAS=*n*
specifies the approximate number of nonarc variables. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

NNODES=*n*
specifies the approximate number of nodes. See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

NON_REPLIC=*non_replic*
prevents PROC NETFLOW from doing unnecessary checks of data previously read.

- NON_REPLIC=COEFS indicates that each constraint coefficient is specified *once* in the CONDATA= data set.
- NON_REPLIC=NONE indicates that constraint coefficients can be specified more than once in the CONDATA= data set. NON_REPLIC=NONE is the default.

See the section "How to Make the Data Read of PROC NETFLOW More Efficient" on page 410.

RHSOBS=*charstr*

specifies the keyword that identifies a right-hand-side observation when using the sparse format for data in the CONDATA= data set. The keyword is expected as a value of the SAS variable in the CONDATA= data set named in the COLUMN list specification. The default value of the RHSOBS= option is _RHS_ or _rhs_. If *charstr* is not a valid SAS variable name, enclose it in single quotes.

SAME_NONARC_DATA

SND

If all nonarc variable data are given in the ARCDATA= data set, or if the problem has no nonarc variables, the unconstrained warm start can be read more quickly if the option SAME_NONARC_DATA is specified. SAME_NONARC_DATA indicates that any nonconstraint nonarc variable data in the CONDATA= data set is to be ignored. Only side constraint data in the CONDATA= data set are read.

If you use an unconstrained warm start and SAME_NONARC_DATA is not specified, any nonarc variable objective function coefficient, upper bound, or lower bound can be changed. Any nonarc variable data in the CONDATA= data set overrides (without warning messages) corresponding data in the ARCDATA= data set. You can possibly introduce new nonarc variables to the problem, that is, nonarc variables that were not in the problem when the warm start was generated.

SAME_NONARC_DATA should be specified if nonarc variable data in the CONDATA= data set are to be deliberately ignored. Consider

```
proc netflow options arcdata=arc0 nodedata=node0
          condata=con0
             /* this data set has nonarc variable        */
             /* objective function coefficient data      */
          future1 arcout=arc1 nodeout=node1;
   run;

data arc2;
   reset arc1;   /* this data set has nonarc variable obs */
   if _cost_<50.0 then _cost_=_cost_*1.25;
                 /* some objective coefficients of nonarc */
                 /* variable might be changed             */
proc netflow options
          warm arcdata=arc2 nodedata=node1
          condata=con0 same_nonarc_data
             /* This data set has old nonarc variable    */
             /* obj, fn. coefficients. same_nonarc_data  */
             /* indicates that the "new" coefs in the    */
             /* arcdata=arc2 are to be used.             */
   run;
```

SCALE=*scale*

indicates that the side constraints are to be scaled. Scaling is useful when some coefficients of a constraint or nonarc variable are either much larger or much smaller than other coefficients. Scaling might make all coefficients have values that have a smaller range, and this can make computations more stable numerically. Try the SCALE= option if PROC NETFLOW is unable to solve a problem because of numerical instability. Specify

- SCALE=ROW, SCALE=CON, or SCALE=CONSTRAINT if the largest absolute value of coefficients in each constraint is about 1.0

- SCALE=COL, SCALE=COLUMN, or SCALE=NONARC if nonarc variable columns are scaled so that the absolute value of the largest constraint coefficient of a nonarc variable is near to 1
- SCALE=BOTH if the largest absolute value of coefficients in each constraint, and the absolute value of the largest constraint coefficient of a nonarc variable is near to 1. This is the default.
- SCALE=NONE if no scaling should be done

SHORTPATH

SP

specifies that PROC NETFLOW solve a shortest path problem. The NETFLOW procedure finds the shortest path between the nodes specified in the SOURCE= option and the SINK= option. The costs of arcs are their *lengths*. PROC NETFLOW automatically assigns a supply of one flow unit to the SOURCE= node, and the SINK= node is assigned to have a one flow unit demand. In this way, the SHORTPATH option sets up a shortest path problem as an equivalent minimum cost problem.

If a network has one supply node (with supply of one unit) and one demand node (with demand of one unit), you could specify the SHORTPATH option, with the SOURCE= and SINK= nodes, even if the problem is not a shortest path problem. You then should not provide any supply or demand data in the NODEDATA= data set or the ARCDATA= data set.

SINK=*sinkname*

SINKNODE=*sinkname*

identifies the demand node. The SINK= option is useful when you specify the MAXFLOW option or the SHORTPATH option and need to specify toward which node the shortest path or maximum flow is directed. The SINK= option also can be used when a minimum cost problem has only one demand node. Rather than having this information in the ARCDATA= data set or the NODEDATA= data set, use the SINK= option with an accompanying DEMAND= specification for this node. The SINK= option must be the name of a head node of at least one arc; thus, it must have a character value. If the value of the SINK= option is not a valid SAS character variable name, it must be enclosed in single quotes and can contain embedded blanks.

SOURCE=*sourcename*

SOURCENODE=*sourcename*

identifies a supply node. The SOURCE= option is useful when you specify the MAXFLOW or the SHORTPATH option and need to specify from which node the shortest path or maximum flow originates. The SOURCE= option also can be used when a minimum cost problem has only one supply node. Rather than having this information in the ARCDATA= data set or the NODEDATA= data set, use the SOURCE= option with an accompanying SUPPLY= amount of supply at this node. The SOURCE= option must be the name of a tail node of at least one arc; thus, it must have a character value. If the value of the SOURCE= option is not a valid SAS character variable name, it must be enclosed in single quotes and can contain embedded blanks.

SPARSECONDATA

SCDATA

indicates that the CONDATA= data set has data in the sparse data format. Otherwise, it is assumed that the data are in the dense format.

NOTE: If the SPARSECONDATA option is not specified, and you are running SAS software Version 6 or you have specified options validvarname=v6;, all NAME list variable values in the ARCDATA= data set are uppercased. See the section "Case Sensitivity" on page 392.

SPARSEP2

SP2

indicates that the new column of the working basis matrix that replaces another column be held in a linked list. If the SPARSEP2 option is not specified, a one-dimensional array is used to store this column's information, that can contain elements that are 0.0 and use more memory than the linked list. The linked list mechanism requires more work if the column has numerous nonzero elements in many iterations. Otherwise, it is superior. Sometimes, specifying SPARSEP2 is beneficial when the side constrained coefficient matrix is very sparse or when some paging is necessary.

SUPPLY=*s*

specifies the supply at the source node specified by the SOURCE= option. The SUPPLY= option should be used only if the SOURCE= option is given in the PROC NETFLOW statement and neither the SHORTPATH option nor the MAXFLOW option is specified. If you are solving a minimum cost network problem and the SOURCE= option is used to identify the source node and the SUPPLY= option is not specified, then by default the supply at the source node is made equal to the network's total demand.

THRUNET

tells PROC NETFLOW to force through the network any excess supply (the amount by which total supply exceeds total demand) or any excess demand (the amount by which total demand exceeds total supply) as is required. If a network problem has unequal total supply and total demand and the THRUNET option is not specified, PROC NETFLOW drains away the excess supply or excess demand in an optimal manner. The consequences of specifying or not specifying THRUNET are discussed in the section "Balancing Total Supply and Total Demand" on page 405.

TYPEOBS=*charstr*

specifies the keyword that identifies a type observation when using the sparse format for data in the CONDATA= data set. The keyword is expected as a value of the SAS variable in the CONDATA= data set named in the COLUMN list specification. The default value of the TYPEOBS= option is _TYPE_ or _type_. If *charstr* is not a valid SAS variable name, enclose it in single quotes.

WARM

indicates that the NODEDATA= data set or the DUALIN= data set and the ARCDATA= data set contain extra information of a warm start to be used by PROC NETFLOW. See the section "Warm Starts" on page 406.

CAPACITY Statement

CAPACITY *variable* **;**

CAPAC *variable* **;**

UPPERBD *variable* **;**

The CAPACITY statement identifies the SAS variable in the ARCDATA= data set that contains the maximum feasible flow or capacity of the network arcs. If an observation contains nonarc variable information, the CAPACITY list variable is the upper value bound for the nonarc variable named in the NAME list variable in that observation. The CAPACITY list variable must have numeric values. It is not necessary to have a CAPACITY statement if the name of the SAS variable is _CAPAC_, _UPPER_, _UPPERBD, or _HI_.

COEF Statement

COEF *variables* **;**

The COEF list is used with the sparse input format of the CONDATA= data set. The COEF list can contain more than one SAS variable, each of which must have numeric values. If the COEF statement is not specified, the CONDATA= data set is searched and SAS variables with names beginning with _COE are used. The number of SAS variables in the COEF list must be no greater than the number of SAS variables in the ROW list.

The values of the COEF list variables in an observation can be interpreted differently than these variables' values in other observations. The values can be coefficients in the side constraints, costs and objective function coefficients, bound data, constraint type data, or rhs data. If the COLUMN list variable has a value that is a name of an arc or nonarc variable, the *i*th COEF list variable is associated with the constraint or special row name named in the *i*th ROW list variable. Otherwise, the COEF list variables indicate type values, rhs values, or missing values.

COLUMN Statement

COLUMN *variable* **;**

The COLUMN list is used with the sparse input format of side constraints. This list consists of one SAS variable in the CONDATA= data set that has as values the names of arc variables, nonarc variables, or missing values. Some, if not all of these values, also can be values of the NAME list variables of the ARCDATA= data set. The COLUMN list variable can have other special values (refer to the TYPEOBS= and RHSOBS= options). If the COLUMN list is not specified after the PROC NETFLOW statement, the CONDATA= data set is searched and a SAS variable named _COLUMN_ is used. The COLUMN list variable must have character values.

CONOPT Statement

CONOPT ;

The CONOPT statement has no options. It is equivalent to specifying RESET SCRATCH;. The CONOPT statement should be used before stage 2 optimization commences. It indicates that the optimization performed next should consider the side constraints.

Usually, the optimal unconstrained network solution is used as a starting solution for constrained optimization. Finding the unconstrained optimum usually reduces the amount of stage 2 optimization. Furthermore, the unconstrained optimum is almost always "closer" to the constrained optimum than the initial basic solution determined before any optimization is performed. However, as the optimum is approached during stage 1 optimization, the flow change candidates become scarcer and a solution good enough to start stage 2 optimization may already be at hand. You should then specify the CONOPT statement.

COST Statement

COST *variable* **;**

OBJFN *variable* **;**

The COST statement identifies the SAS variable in the ARCDATA= data set that contains the per unit flow cost through an arc. If an observation contains nonarc variable information, the value of the COST list variable is the objective function coefficient of the nonarc variable named in the NAME list variable in that observation. The COST list variable must have numeric values. It is not necessary to specify a COST statement if the name of the SAS variable is _COST_ or _LENGTH_.

DEMAND Statement

DEMAND *variable* **;**

The DEMAND statement identifies the SAS variable in the ARCDATA= data set that contains the demand at the node named in the corresponding HEADNODE list variable. The DEMAND list variable must have numeric values. It is not necessary to have a DEMAND statement if the name of this SAS variable is _DEMAND_.

HEADNODE Statement

HEADNODE *variable* **;**

HEAD *variable* **;**

TONODE *variable* **;**

TO *variable* **;**

The HEADNODE statement specifies the SAS variable that must be present in the ARCDATA= data set that contains the names of nodes toward which arcs are directed. It is not necessary to have a HEADNODE statement if the name of the SAS variable is _HEAD_ or _TO_. The HEADNODE variable must have character values.

ID Statement

ID *variables* **;**

The ID statement specifies SAS variables containing values for pre- and post-optimal processing and analysis. These variables are not processed by PROC NETFLOW but are read by the procedure and written in the ARCOUT= and CONOUT= data sets and the output of PRINT statements. For example, imagine a network used to model a distribution system. The SAS variables listed on the ID statement can contain information on type of vehicle, transportation mode, condition of road, time to complete journey, name of driver, or other ancillary information useful for report writing or describing facets of the operation that do not have bearing on the optimization. The ID variables can be character, numeric, or both.

If no ID list is specified, the procedure forms an ID list of all SAS variables not included in any other implicit or explicit list specification. If the ID list is specified, any SAS variables in the ARCDATA= data set not in any list are dropped and do not appear in the ARCOUT= or CONOUT= data sets, or in the PRINT statement output.

LO Statement

LO *variable* **;**

LOWERBD *variable* **;**

MINFLOW *variable* **;**

The LO statement identifies the SAS variable in the ARCDATA= data set that contains the minimum feasible flow or lower flow bound for arcs in the network. If an observation contains nonarc variable information, the LO list variable has the value of the lower bound for the nonarc variable named in the NAME list variable. The LO list variables must have numeric values. It is not necessary to have a LO statement if the name of this SAS variable is _LOWER_, _LO_, _LOWERBD, or _MINFLOW.

MULT Statement

MULT *variables* **;**

MULTIPLIER *variables* **;**

The MULT statement identifies the SAS variable in the ARCDATA= data set associated with the values of arc multipliers in the network. These values must be positive real numbers. It is not necessary to have a MULT statement if the name of this SAS variable is _MULT_.

NAME Statement

NAME *variable* **;**

ARCNAME *variable* **;**

VARNAME *variable* **;**

Each arc and nonarc variable that has data in the CONDATA= data set must have a unique name. This variable is identified in the ARCDATA= data set. The NAME list variable must have character values (see the NAMECTRL= option in the PROC NETFLOW statement for more information). It is not necessary to have a NAME statement if the name of this SAS variable is _NAME_.

NODE Statement

NODE *variable* **;**

The NODE list variable, which must be present in the NODEDATA= data set, has names of nodes as values. These values must also be values of the TAILNODE list variable, the HEADNODE list variable, or both. If this list is not explicitly specified, the NODEDATA= data set is searched for a SAS variable with the name _NODE_. The NODE list variable must have character values.

PIVOT Statement

PIVOT ;

The PIVOT statement has no options. It indicates that one simplex iteration is to be performed. The PIVOT statement forces a simplex iteration to be performed in spite of the continued presence of any reasons or solution conditions that caused optimization to be halted. For example, if the number of iterations performed exceeds the value of the MAXIT1= or MAXIT2= option and you issue a PIVOT statement, the iteration is performed even though the MAXIT1= or MAXIT2= value has not yet been changed using a RESET statement.

PRINT Statement

PRINT *options / qualifiers* **;**

The options available with the PRINT statement of PROC NETFLOW are summarized by purpose in the following table.

Table 6.2 Functional Summary, PRINT Statement

Description	Statement	Option
PRINT Statement Options:		
Display everything	PRINT	PROBLEM
Display arc information	PRINT	ARCS
Display nonarc variable information	PRINT	NONARCS
Display variable information	PRINT	VARIABLES
Display constraint information	PRINT	CONSTRAINTS
Display information for some arcs	PRINT	SOME_ARCS
Display information for some nonarc variables	PRINT	SOME_NONARCS
Display information for some variables	PRINT	SOME_VARIABLES
Display information for some constraints	PRINT	SOME_CONS
Display information for some constraints associated with some arcs	PRINT	CON_ARCS
Display information for some constraints associated with some nonarc variables	PRINT	CON_NONARCS

Description	Statement	Option
Display information for some constraints associated with some variables	PRINT	CON_VARIABLES
PRINT Statement Qualifiers:		
Produce a short report	PRINT	/ SHORT
Produce a long report	PRINT	/ LONG
Display arcs/variables with zero flow/value	PRINT	/ ZERO
Display arcs/variables with nonzero flow/value	PRINT	/ NONZERO
Display basic arcs/variables	PRINT	/ BASIC
Display nonbasic arcs/variables	PRINT	/ NONBASIC

The PRINT statement enables you to examine part or all of the problem. You can limit the amount of information displayed when a PRINT statement is processed by specifying PRINT statement options. The name of the PRINT option indicates what part of the problem is to be examined. If no options are specified, or PRINT PROBLEM is specified, information about the entire problem is produced.

The amount of displayed information can be limited further by following any PRINT statement options with a slash (/) and one or more of the qualifiers SHORT or LONG, ZERO or NONZERO, BASIC or NONBASIC.

Some of the PRINT statement options require you to specify a list of some type of entity, thereby enabling you to indicate what entities are of interest. The entities of interest are the ones you want to display. These entities might be tail node names, head node names, nonarc variable names, or constraint names. The entity list is made up of one or more of the following constructs. Each construct can add none, one, or more entities to the set of entities to be displayed.

- _ALL_
  Display all entities in the required list.
- entity
  Display the named entity that is interesting.
- entity1 - entity2 (one hyphen)
  entity1 -- entity2 (two hyphens)
  entity1 - CHARACTER - entity2 or
  entity1 - CHAR - entity2
  Both entity1 and entity2 have names made up of the same character string prefix followed by a numeric suffix. The suffixes of both entity1 and entity2 have the same number of numerals but can have different values. A specification of entity1 - entity2 indicates that all entities with the same prefix and suffixes with values on or between the suffixes of entity1 and entity2 are to be put in the set of entities to be displayed. The numeric suffix of both entity1 and entity2 can be followed by a character string. For example, _OBS07_ - _OBS13_ is a valid construct of the forms entity1 - entity2.
- part_of_entity_name:
  Display all entities in the required list that have names beginning with the character string preceding the colon.

The following options can appear in the PRINT statement:

ARCS
indicates that you want to have displayed information about all arcs.

SOME_ARCS (*taillist,headlist*)
is similar to the statement PRINT ARCS except that, instead of displaying information about all arcs, only arcs directed from nodes in a specified set of tail nodes to nodes in a specified set of head nodes are included. The nodes or node constructs belonging to the *taillist* list are separated by blanks. The nodes or node constructs belonging to the *headlist* list are also separated by blanks. The lists are separated by a comma.

NONARCS

VARIABLES
indicates that information is to be displayed about all nonarc variables.

SOME_NONARCS (*nonarclist*)
is similar to the PRINT NONARCS statement except that, instead of displaying information about all nonarc variables, only those belonging to a specified set of nonarc variables have information displayed. The nonarc variables or nonarc variable constructs belonging to the *nonarclist* list are separated by blanks.

CONSTRAINTS
indicates that you want to have displayed information about all constraint coefficients.

SOME_CONS (*conlist*)
displays information for coefficients in a specified set of constraints. The constraints or constraint constructs belonging to the *conlist* list are separated by blanks.

CON_ARCS (*taillist, headlist*)
is similar to the PRINT SOME_CONS (*conlist*) statement except that, instead of displaying information about all coefficients in specified constraints, information about only those coefficients that are associated with arcs directed from a set of specified tail nodes toward a set of specified head nodes is displayed. The constraints or constraint constructs belonging to the *conlist* list are separated by blanks; so too are the nodes or node constructs belonging to the *taillist* list and the nodes or node constructs belonging to the *headlist* list. The lists are separated by commas.

CON_NONARCS (*conlist, nonarclist*)

CON_VARIABLES (*conlist, variablelist*)
is similar to the PRINT SOME_CONS (*conlist*) statement except that, instead of displaying information about all coefficients in specified constraints, information about only those coefficients that are associated with nonarc variables in a specified set is displayed. The constraints or constraint constructs belonging to the *conlist* list are separated by blanks. The nonarc variables or nonarc variable constructs belonging to the *nonarclist* list are separated by blanks. The lists are separated by a comma.

PROBLEM
is equivalent to the statement PRINT ARCS NONARCS CONSTRAINTS.

Following a slash (/), the qualifiers SHORT or LONG, ZERO or NONZERO, BASIC or NONBASIC can appear in any PRINT statement. These qualifiers are described below.

- BASIC
  Only rows that are associated with arcs or nonarc variables that are basic are displayed. The _STATUS_ column values are KEY_ARC BASIC or NONKEY ARC BASIC for arcs, and NONKEY_BASIC for nonarc variables.
- LONG
  All table columns are displayed (the default when no qualifier is used).
- NONBASIC
  Only rows that are associated with arcs or nonarc variables that are nonbasic are displayed. The _STATUS_ column values are LOWERBD NONBASIC or UPPERBD NONBASIC.
- NONZERO
  Only rows that have nonzero _FLOW_ column values (nonzero arc flows, nonzero nonarc variable values) are displayed.
- SHORT
  The table columns are _N_, _FROM_, _TO_, _COST_, _CAPAC_, _LO_, _NAME_, and _FLOW_, or the names of the SAS variables specified in the corresponding variable lists (TAILNODE, HEADNODE, COST, CAPACITY, LO, and NAME lists). _COEF_ or the name of the SAS variable in the COEF list specification will head a column when the SHORT qualifier is used in PRINT CONSTRAINTS, SOME_CONS, CON_ARCS, or CON_NONARCS.
- ZERO
  Only rows that have zero _FLOW_ column values (zero arc flows, zero nonarc variable values) are displayed.

The default qualifiers are BASIC, NONBASIC, ZERO, NONZERO, and LONG.

Displaying Information On All Constraints

In the oil refinery problem, if you had entered

```
print constraints;
```

after the RUN statement, the output in Figure 6.9 would have been produced.

Displaying Information About Selected Arcs

In the oil refinery problem, if you had entered

```
print some_arcs(refin:,_all_)/short;
```

after the RUN statement, the output in Figure 6.10 would have been produced.

Figure 6.9 PRINT CONSTRAINTS

Constrained Optimum

The NETFLOW Procedure

_N_	_CON_	_type_	_rhs_	_name_	_from_	_to_	_cost_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_FLOW_	_COEF_
1	_OBS1_	GE	-15	m_e_ref1	middle east	refinery 1	63	95	20	100	.	80	-2
2	_OBS1_	GE	-15	thruput1	refinery 1	r1	200	175	50	.	.	145	1
3	_OBS2_	GE	-15	m_e_ref2	middle east	refinery 2	81	80	10	100	.	20	-2
4	_OBS2_	GE	-15	thruput2	refinery 2	r2	220	100	35	.	.	35	1
5	_OBS3_	EQ	0	thruput1	refinery 1	r1	200	175	50	.	.	145	-3
6	_OBS3_	EQ	0	r1_gas	r1	ref1 gas	0	140	0	.	.	108.75	4
7	_OBS4_	EQ	0	thruput2	refinery 2	r2	220	100	35	.	.	35	-3
8	_OBS4_	EQ	0	r2_gas	r2	ref2 gas	0	100	0	.	.	26.25	4

_N_	_FCOST_	_RCOST_	_STATUS_	
1	5040	.	KEY_ARC	BASIC
2	29000	.	KEY_ARC	BASIC
3	1620	.	NONKEY ARC	BASIC
4	7700	29	LOWERBD	NONBASIC
5	29000	.	KEY_ARC	BASIC
6	0	.	KEY_ARC	BASIC
7	7700	29	LOWERBD	NONBASIC
8	0	.	KEY_ARC	BASIC

Figure 6.10 PRINT SOME_ARCS

Constrained Optimum

The NETFLOW Procedure

_N_	_from_	_to_	_cost_	_capac_	_lo_	_name_	_FLOW_
1	refinery 1	r1	200	175	50	thruput1	145
2	refinery 2	r2	220	100	35	thruput2	35

Displaying Information About Selected Constraints

In the oil refinery problem, if you had entered

```
print some_cons(_obs3_-_obs4_)/nonzero short;
```

after the RUN statement, the output in Figure 6.11 would have been produced.

Figure 6.11 PRINT SOME_CONS

Constrained Optimum

The NETFLOW Procedure

_N_	_CON_	_type_	_rhs_	_name_	_from_	_to_	_cost_	_capac_	_lo_	_FLOW_	_COEF_
1	_OBS3_	EQ	0	thruput1	refinery 1	r1	200	175	50	145	-3
2	_OBS3_	EQ	0	r1_gas	r1	ref1 gas	0	140	0	108.75	4
3	_OBS4_	EQ	0	thruput2	refinery 2	r2	220	100	35	35	-3
4	_OBS4_	EQ	0	r2_gas	r2	ref2 gas	0	100	0	26.25	4

If you had entered

```
print con_arcs(_all_,r1 r2,_all_)/short;
```

after the RUN statement, the output in Figure 6.12 would have been produced. Constraint information about arcs directed from selected tail nodes is displayed.

Figure 6.12 PRINT CON_ARCS

Constrained Optimum

The NETFLOW Procedure

_N_	_CON_	_type_	_rhs_	_name_	_from_	_to_	_cost_	_capac_	_lo_	_FLOW_	_COEF_
1	_OBS3_	EQ	0	r1_gas	r1	ref1 gas	0	140	0	108.75	4
2	_OBS4_	EQ	0	r2_gas	r2	ref2 gas	0	100	0	26.25	4

Cautions

This subsection has two parts; the first part is applicable if you are running Version 7 or later of the SAS System, and the second part is applicable if you are running Version 6. You can get later versions to "act" like Version 6 by specifying

```
options validvarname=v6;
```

For Version 7 onward, PROC NETFLOW strictly respects case sensitivity. The PRINT statements of PROC NETFLOW that require lists of entities will work properly *only* if the entities have the same case as in the input data sets. Entities that contain blanks must be enclosed in single or double quotes. For example,

```
print some_arcs (_all_,"Ref1 Gas");
```

In this example, a head node of an arc in the model is "Ref1 Gas" (without the quotes). If you omit the quotes, PROC NETFLOW issues a message on the SAS log:

```
WARNING: The node Ref1 in the list of head nodes in the PRINT
         SOME_ARCS or PRINT CON_ARCS is not a node in the
         problem. This statement will be ignored.
```

If you had specified

```
print some_arcs (_all_,"ref1 Gas");
```

(note the small r), you would have been warned

```
WARNING: The node ref1 Gas in the list of head nodes in the PRINT
         SOME_ARCS or PRINT CON_ARCS is not a node in the
         problem. This statement will be ignored.
```

If you are running Version 6, or if you are running a later version and you have specified

```
options validvarname=v6;
```

when information is parsed to procedures, the SAS System converts the text that makes up statements into uppercase. The PRINT statements of PROC NETFLOW that require lists of entities will work properly *only* if the entities are uppercase in the input data sets. If you do not want this uppercasing to occur, you must enclose the entity in single or double quotes.

```
print some_arcs('lowercase tail','lowercase head');
print some_cons('factory07'-'factory12');
print some_cons('_factory07_'-'_factory12_');
print some_nonarcs("CO2 content":);
```

Entities that contain blanks must be enclosed in single or double quotes.

QUIT Statement

QUIT ;

The QUIT statement indicates that PROC NETFLOW is to be terminated immediately. The solution is not saved in the current output data sets. The QUIT statement has no options.

RESET Statement

RESET *options* **;**

SET *options* **;**

The RESET statement is used to change options after PROC NETFLOW has started execution. Any of the following options can appear in the PROC NETFLOW statement.

Another name for the RESET statement is SET. You can use RESET when you are resetting options and SET when you are setting options for the first time. The following options fall roughly into five categories:

- output data set specifications
- options that indicate conditions under which optimization is to be halted temporarily, giving you an opportunity to use PROC NETFLOW interactively

- options that control aspects of the operation of the network primal simplex optimization
- options that control the pricing strategies of the network simplex optimizer
- miscellaneous options

If you want to examine the setting of any options, use the SHOW statement. If you are interested in looking at only those options that fall into a particular category, the SHOW statement has options that enable you to do this.

The execution of PROC NETFLOW has three stages. In stage zero the problem data are read from the NODEDATA=, ARCDATA=, and CONDATA= data sets. If a warm start is not available, an initial basic feasible solution is found. Some options of the PROC NETFLOW statement control what occurs in stage zero. By the time the first RESET statement is processed, stage zero has already been completed.

In the first stage, an optimal solution to the network flow problem neglecting any side constraints is found. The primal and dual solutions for this relaxed problem can be saved in the ARCOUT= data set and the NODEOUT= data set, respectively.

In the second stage, the side constraints are examined and some initializations occur. Some preliminary work is also needed to commence optimization that considers the constraints. An optimal solution to the network flow problem with side constraints is found. The primal and dual solutions for this side-constrained problem are saved in the CONOUT= data set and the DUALOUT= data set, respectively.

Many options in the RESET statement have the same name except that they have as a suffix the numeral 1 or 2. Such options have much the same purpose, but option1 controls what occurs during the first stage when optimizing the network neglecting any side constraints and option2 controls what occurs in the second stage when PROC NETFLOW is performing constrained optimization.

Some options can be turned off by the option prefixed by the word *NO*. For example, FEASIBLEPAUSE1 may have been specified in a RESET statement and in a later RESET statement, you can specify NOFEASIBLEPAUSE1. In a later RESET statement, you can respecify FEASIBLEPAUSE1 and, in this way, toggle this option.

The options available with the RESET statement are summarized by purpose in the following table.

Table 6.3 Functional Summary, RESET Statement

Description	Statement	Option
Output Data Set Options:		
Unconstrained primal solution data set	RESET	ARCOUT=
Unconstrained dual solution data set	RESET	NODEOUT=
Constrained primal solution data set	RESET	CONOUT=
Constrained dual solution data set	RESET	DUALOUT=
Simplex Options:		
Use big-M instead of two-phase method, stage 1	RESET	BIGM1
Use Big-M instead of two-phase method, stage 2	RESET	BIGM2
Anti-cycling option	RESET	CYCLEMULT1=
Interchange first nonkey with leaving key arc	RESET	INTFIRST

Description	Statement	Option
Controls working basis matrix inversions	RESET	INVFREQ=
Maximum number of L row operations allowed before refactorization	RESET	MAXL=
Maximum number of LU factor column updates	RESET	MAXLUUPDATES=
Anti-cycling option	RESET	MINBLOCK1=
Use first eligible leaving variable, stage 1	RESET	LRATIO1
Use first eligible leaving variable, stage 2	RESET	LRATIO2
Negates INTFIRST	RESET	NOINTFIRST
Negates LRATIO1	RESET	NOLRATIO1
Negates LRATIO2	RESET	NOLRATIO2
Negates PERTURB1	RESET	NOPERTURB1
Anti-cycling option	RESET	PERTURB1
Controls working basis matrix refactorization	RESET	REFACTFREQ=
Use two-phase instead of big-M method, stage 1	RESET	TWOPHASE1
Use two-phase instead of big-M method, stage 2	RESET	TWOPHASE2
Pivot element selection parameter	RESET	U=
Zero tolerance, stage 1	RESET	ZERO1=
Zero tolerance, stage 2	RESET	ZERO2=
Zero tolerance, real number comparisons	RESET	ZEROTOL=
Pricing Options:		
Frequency of dual value calculation	RESET	DUALFREQ=
Pricing strategy, stage 1	RESET	PRICETYPE1=
Pricing strategy, stage 2	RESET	PRICETYPE2=
Used when P1SCAN=PARTIAL	RESET	P1NPARTIAL=
Controls search for entering candidate, stage 1	RESET	P1SCAN=
Used when P2SCAN=PARTIAL	RESET	P2NPARTIAL=
Controls search for entering candidate, stage 2	RESET	P2SCAN=
Initial queue size, stage 1	RESET	QSIZE1=
Initial queue size, stage 2	RESET	QSIZE2=
Used when Q1FILLSCAN=PARTIAL	RESET	Q1FILLNPARTIAL=
Controls scan when filling queue, stage 1	RESET	Q1FILLSCAN=
Used when Q2FILLSCAN=PARTIAL	RESET	Q2FILLNPARTIAL=
Controls scan when filling queue, stage 2	RESET	Q2FILLSCAN=
Queue size reduction factor, stage 1	RESET	REDUCEQSIZE1=
Queue size reduction factor, stage 2	RESET	REDUCEQSIZE2=
Frequency of refreshing queue, stage 1	RESET	REFRESHQ1=
Frequency of refreshing queue, stage 2	RESET	REFRESHQ2=
Optimization Termination Options:		
Pause after stage 1; do not start stage 2	RESET	ENDPAUSE1
Pause when feasible, stage 1	RESET	FEASIBLEPAUSE1
Pause when feasible, stage 2	RESET	FEASIBLEPAUSE2
Maximum number of iterations, stage 1	RESET	MAXIT1=
Maximum number of iterations, stage 2	RESET	MAXIT2=

Description	Statement	Option
Negates ENDPAUSE1	RESET	NOENDPAUSE1
Negates FEASIBLEPAUSE1	RESET	NOFEASIBLEPAUSE1
Negates FEASIBLEPAUSE2	RESET	NOFEASIBLEPAUSE2
Pause every PAUSE1 iterations, stage 1	RESET	PAUSE1=
Pause every PAUSE2 iterations, stage 2	RESET	PAUSE2=
Interior Point Algorithm Options:		
Factorization method	RESET	FACT_METHOD=
Allowed amount of dual infeasibility	RESET	TOLDINF=
Allowed amount of primal infeasibility	RESET	TOLPINF=
Allowed total amount of dual infeasibility	RESET	TOLTOTDINF=
Allowed total amount of primal infeasibility	RESET	TOLTOTPINF=
Cut-off tolerance for Cholesky factorization	RESET	CHOLTINYTOL=
Density threshold for Cholesky processing	RESET	DENSETHR=
Step-length multiplier	RESET	PDSTEPMULT=
Preprocessing type	RESET	PRSLTYPE=
Print optimization progress on SAS log	RESET	PRINTLEVEL2=
Interior Point Stopping Criteria Options:		
Maximum number of interior point iterations	RESET	MAXITERB=
Primal-dual (duality) gap tolerance	RESET	PDGAPTOL=
Stop because of complementarity	RESET	STOP_C=
Stop because of duality gap	RESET	STOP_DG=
Stop because of $infeas_b$	RESET	STOP_IB=
Stop because of $infeas_c$	RESET	STOP_IC=
Stop because of $infeas_d$	RESET	STOP_ID=
Stop because of complementarity	RESET	AND_STOP_C=
Stop because of duality gap	RESET	AND_STOP_DG=
Stop because of $infeas_b$	RESET	AND_STOP_IB=
Stop because of $infeas_c$	RESET	AND_STOP_IC=
Stop because of $infeas_d$	RESET	AND_STOP_ID=
Stop because of complementarity	RESET	KEEPGOING_C=
Stop because of duality gap	RESET	KEEPGOING_DG=
Stop because of $infeas_b$	RESET	KEEPGOING_IB=
Stop because of $infeas_c$	RESET	KEEPGOING_IC=
Stop because of $infeas_d$	RESET	KEEPGOING_ID=
Stop because of complementarity	RESET	AND_KEEPGOING_C=
Stop because of duality gap	RESET	AND_KEEPGOING_DG=
Stop because of $infeas_b$	RESET	AND_KEEPGOING_IB=
Stop because of $infeas_c$	RESET	AND_KEEPGOING_IC=
Stop because of $infeas_d$	RESET	AND_KEEPGOING_ID=
Miscellaneous Options:		
Output complete basis information to ARCOUT= and NODEOUT= data sets	RESET	FUTURE1

Description	Statement	Option
Output complete basis information to CONOUT= and DUALOUT= data sets	RESET	FUTURE2
Turn off infeasibility or optimality flags	RESET	MOREOPT
Negates FUTURE1	RESET	NOFUTURE1
Negates FUTURE2	RESET	NOFUTURE2
Negates SCRATCH	RESET	NOSCRATCH
Negates ZTOL1	RESET	NOZTOL1
Negates ZTOL2	RESET	NOZTOL2
Write optimization time to SAS log	RESET	OPTIM_TIMER
No stage 1 optimization; do stage 2 optimization	RESET	SCRATCH
Suppress similar SAS log messages	RESET	VERBOSE=
Use zero tolerance, stage 1	RESET	ZTOL1
Use zero tolerance, stage 2	RESET	ZTOL2

Output Data Set Specifications

In a RESET statement, you can specify an ARCOUT= data set, a NODEOUT= data set, a CONOUT= data set, or a DUALOUT= data set. You are advised to specify these output data sets early because if you make a syntax error when using PROC NETFLOW interactively or, for some other reason, PROC NETFLOW encounters or does something unexpected, these data sets will contain information about the solution that was reached. If you had specified the FUTURE1 or FUTURE2 option in a RESET statement, PROC NETFLOW may be able to resume optimization in a subsequent run.

You can turn off these current output data set specifications by specifying ARCOUT=NULL, NODEOUT=NULL, CONOUT=NULL, or DUALOUT=NULL.

If PROC NETFLOW is outputting observations to an output data set and you want this to stop, press the keys used to stop SAS procedures. PROC NETFLOW waits, if necessary, and then executes the next statement.

ARCOUT=*SAS-data-set*

AOUT=*SAS-data-set*

names the output data set that receives all information concerning arc and nonarc variables, including flows and other information concerning the current solution and the supply and demand information. The current solution is the latest solution found by the optimizer when the optimization neglecting side constraints is halted or the unconstrained optimum is reached.

You can specify an ARCOUT= data set in any RESET statement before the unconstrained optimum is found (even at commencement). Once the unconstrained optimum has been reached, use the SAVE statement to produce observations in an ARCOUT= data set. Once optimization that considers constraints starts, you will be unable to obtain an ARCOUT= data set. Instead, use a CONOUT= data set to get the current solution. See the section "ARCOUT= and CONOUT= Data Sets" on page 389 for more information.

CONOUT=*SAS-data-set*

COUT=*SAS-data-set*

names the output data set that contains the primal solution obtained after optimization considering side constraints reaches the optimal solution. You can specify a CONOUT= data set in any RESET statement before the constrained optimum is found (even at commencement or while optimizing neglecting constraints). Once the constrained optimum has been reached, or during stage 2 optimization, use the SAVE statement to produce observations in a CONOUT= data set. See the section "ARCOUT= and CONOUT= Data Sets" on page 389 for more information.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

names the output data set that contains the dual solution obtained after doing optimization that considers side constraints reaches the optimal solution. You can specify a DUALOUT= data set in any RESET statement before the constrained optimum is found (even at commencement or while optimizing neglecting constraints). Once the constrained optimum has been reached, or during stage 2 optimization, use the SAVE statement to produce observations in a DUALOUT= data set. See the section "NODEOUT= and DUALOUT= Data Sets" on page 390 for more information.

NODEOUT=*SAS-data-set*

NOUT=*SAS-data-set*

names the output data set that receives all information about nodes (supply/demand and nodal dual variable values) and other information concerning the unconstrained optimal solution.

You can specify a NODEOUT= data set in any RESET statement before the unconstrained optimum is found (even at commencement). Once the unconstrained optimum has been reached, or during stage 1 optimization, use the SAVE statement to produce observations in a NODEOUT= data set. Once optimization that considers constraints starts, you will not be able to obtain a NODEOUT= data set. Instead use a DUALOUT= data set to get the current solution. See the section "NODEOUT= and DUALOUT= Data Sets" on page 390 for more information.

Options to Halt Optimization

The following options indicate conditions when optimization is to be halted. You then have a chance to use PROC NETFLOW interactively. If the NETFLOW procedure is optimizing and you want optimization to halt immediately, press the CTRL-BREAK key combination used to stop SAS procedures. Doing this is equivalent to PROC NETFLOW finding that some prespecified condition of the current solution under which optimization should stop has occurred.

If optimization does halt, you may need to change the conditions for when optimization should stop again. For example, if the number of iterations exceeded MAXIT2, use the RESET statement to specify a larger value for the MAXIT2= option before the next RUN statement. Otherwise, PROC NETFLOW will immediately find that the number of iterations still exceeds MAXIT2 and halt without doing any additional optimization.

ENDPAUSE1

indicates that PROC NETFLOW will pause after the unconstrained optimal solution has been obtained and information about this solution has been output to the current ARCOUT= data set, NODEOUT= data set, or both. The procedure then executes the next statement, or waits if no subsequent statement has been specified.

FEASIBLEPAUSE1

FP1

indicates that unconstrained optimization should stop once a feasible solution is reached. PROC NETFLOW checks for feasibility every 10 iterations. A solution is feasible if there are no artificial arcs having nonzero flow assigned to be conveyed through them. The presence of artificial arcs with nonzero flows means that the current solution does not satisfy all the nodal flow conservation constraints implicit in network problems.

MAXIT1=*m*

specifies the maximum number of primal simplex iterations PROC NETFLOW is to perform in stage 1. The default value for the MAXIT1= option is 1000. If MAXIT1=*m* iterations are performed and you want to continue unconstrained optimization, reset MAXIT1= to a number larger than the number of iterations already performed and issue another RUN statement.

NOENDPAUSE1

NOEP1

negates the ENDPAUSE1 option.

NOFEASIBLEPAUSE1

NOFP1

negates the FEASIBLEPAUSE1 option.

PAUSE1=*p*

indicates that PROC NETFLOW will halt unconstrained optimization and pause when the remainder of the number of stage 1 iterations divided by the value of the PAUSE1= option is zero. If present, the next statement is executed; if not, the procedure waits for the next statement to be specified. The default value for PAUSE1= is 999999.

FEASIBLEPAUSE2

FP2

NOFEASIBLEPAUSE2

NOFP2

PAUSE2=*p*

MAXIT2=*m*

are the stage 2 constrained optimization counterparts of the options described previously and having as a suffix the numeral 1.

Options Controlling the Network Simplex Optimization

BIGM1

NOTWOPHASE1

TWOPHASE1

NOBIGM1

BIGM1 indicates that the "big-M" approach to optimization is used. Artificial variables are treated like real arcs, slacks, surpluses and nonarc variables. Artificials have very expensive costs. BIGM1 is the default.

TWOPHASE1 indicates that the two-phase approach is used instead of the big-M approach. At first, artificial variables are the only variables to have nonzero objective function coefficients. An artificial variable's objective function coefficient is temporarily set to 1 and PROC NETFLOW minimizes. When all artificial variables have zero value, PROC NETFLOW has found a feasible solution, and phase 2 commences. Arcs and nonarc variables have their real costs and objective function coefficients.

Before all artificial variables are driven to have zero value, you can toggle between the big-M and the two-phase approaches by specifying BIGM1 or TWOPHASE1 in a RESET statement. The option NOTWOPHASE1 is synonymous with BIGM1, and NOBIGM1 is synonymous with TWOPHASE1.

CYCLEMULT1=*c*

MINBLOCK1=*m*

NOPERTURB1

PERTURB1

In an effort to reduce the number of iterations performed when the problem is highly degenerate, PROC NETFLOW has in stage 1 optimization adopted an algorithm outlined in Ryan and Osborne (1988).

If the number of consecutive degenerate pivots (those with no progress toward the optimum) performed equals the value of the CYCLEMULT1= option times the number of nodes, the arcs that were "blocking" (can leave the basis) are added to a list. In subsequent iterations, of the arcs that now can leave the basis, the one chosen to leave is an arc on the list of arcs that could have left in the previous iteration. In other words, preference is given to arcs that "block" many iterations. After several iterations, the list is cleared.

If the number of blocking arcs is less than the value of the MINBLOCK1= option, a list is not kept. Otherwise, if PERTURB1 is specified, the arc flows are perturbed by a random quantity, so that arcs on the list that block subsequent iterations are chosen to leave the basis randomly. Although perturbation often pays off, it is computationally expensive. Periodically, PROC NETFLOW has to clear out the lists and un-perturb the solution. You can specify NOPERTURB1 to prevent perturbation.

Defaults are CYCLEMULT1=0.15, MINBLOCK1=2, and NOPERTURB1.

LRATIO1

specifies the type of ratio test to use in determining which arc leaves the basis in stage 1. In some iterations, more than one arc is eligible to leave the basis. Of those arcs that can leave the basis, the leaving arc is the first encountered by the algorithm if the LRATIO1 option is specified. Specifying the LRATIO1 option can decrease the chance of cycling but can increase solution times. The alternative to the LRATIO1 option is the NOLRATIO1 option, which is the default.

LRATIO2

specifies the type of ratio test to use in determining what leaves the basis in stage 2. In some iterations, more than one arc, constraint slack, surplus, or nonarc variable is eligible to leave the basis. If the LRATIO2 option is specified, the leaving arc, constraint slack, surplus, or nonarc variable is the one that is eligible to leave the basis first encountered by the algorithm. Specifying the LRATIO2 option can decrease the chance of cycling but can increase solution times. The alternative to the LRATIO2 option is the NOLRATIO2 option, which is the default.

NOLRATIO1

specifies the type of ratio test to use in determining which arc leaves the basis in stage 1. If the NOLRATIO1 option is specified, of those arcs that can leave the basis, the leaving arc has the minimum (maximum) cost if the leaving arc is to be nonbasic with flow capacity equal to its capacity (lower flow bound). If more than one possible leaving arc has the minimum (maximum) cost, the first such arc encountered is chosen. Specifying the NOLRATIO1 option can decrease solution times, but can increase the chance of cycling. The alternative to the NOLRATIO1 option is the LRATIO1 option. The NOLRATIO1 option is the default.

NOLRATIO2

specifies the type of ratio test to use in determining which arc leaves the basis in stage 2. If the NOLRATIO2 option is specified, the leaving arc, constraint slack, surplus, or nonarc variable is the one eligible to leave the basis with the minimum (maximum) cost or objective function coefficient if the leaving arc, constraint slack or nonarc variable is to be nonbasic with flow or value equal to its capacity or upper value bound (lower flow or value bound), respectively. If several possible leaving arcs, constraint slacks, surpluses, or nonarc variables have the minimum (maximum) cost or objective function coefficient, then the first encountered is chosen. Specifying the NOLRATIO2 option can decrease solution times, but can increase the chance of cycling. The alternative to the NOLRATIO2 option is the LRATIO2 option. The NOLRATIO2 option is the default.

Options Applicable to Constrained Optimization

The INVFREQ= option is relevant only if INVD_2D is specified in the PROC NETFLOW statement; that is, the inverse of the working basis matrix is being stored and processed as a two-dimensional array. The REFACTFREQ=, U=, MAXLUUPDATES=, and MAXL= options are relevant if the INVD_2D option is not specified in the PROC NETFLOW statement; that is, if the working basis matrix is **LU** factored.

BIGM2

NOTWOPHASE2

TWOPHASE2

NOBIGM2

are the stage 2 constrained optimization counterparts of the options BIGM1, NOTWOPHASE1, TWOPHASE1, and NOBIGM1.

The TWOPHASE2 option is often better than the BIGM2 option when the problem has many side constraints.

INVFREQ=*n*

recalculates the working basis matrix inverse whenever *n* iterations have been performed where *n* is the value of the INVFREQ= option. Although a relatively expensive task, it is prudent to do as roundoff errors accumulate, especially affecting the elements of this matrix inverse. The default is INVFREQ=50. The INVFREQ= option should be used only if the INVD_2D option is specified in the PROC NETFLOW statement.

INTFIRST

In some iterations, it is found that what must leave the basis is an arc that is part of the spanning tree representation of the network part of the basis (called a *key* arc). It is necessary to interchange another basic arc not part of the tree (called a *nonkey* arc) with the tree arc that leaves to permit the basis update to be performed efficiently. Specifying the INTFIRST option indicates that of the nonkey arcs eligible to be swapped with the leaving key arc, the one chosen to do so is the first encountered by the algorithm. If the INTFIRST option is not specified, all such arcs are examined and the one with the best cost is chosen.

The terms *key* and *nonkey* are used because the algorithm used by PROC NETFLOW for network optimization considering side constraints (GUB-based, Primal Partitioning, or Factorization) is a variant of an algorithm originally developed to solve linear programming problems with generalized upper bounding constraints. The terms *key* and *nonkey* were coined then. The STATUS SAS variable in the ARCOUT= and CONOUT= data sets and the STATUS column in tables produced when PRINT statements are processed indicate whether basic arcs are key or nonkey. Basic nonarc variables are always nonkey.

MAXL=*m*

If the working basis matrix is **LU** factored, **U** is an upper triangular matrix and **L** is a lower triangular matrix corresponding to a sequence of elementary matrix row operations required to change the working basis matrix into **U**. **L** and **U** enable substitution techniques to be used to solve the linear systems of the simplex algorithm. Among other things, the **LU** processing strives to keep the number of **L** elementary matrix row operation matrices small. A buildup in the number of these could indicate that fill-in is becoming excessive and the computations involving **L** and **U** will be hampered. Refactorization should be performed to restore **U** sparsity and reduce **L** information. When the number of **L** matrix row operations exceeds the value of the MAXL= option, a refactorization is done rather than one or more updates. The default value for MAXL= is 10 times the number of side constraints. The MAXL= option should not be used if INVD_2D is specified in the PROC NETFLOW statement.

MAXLUUPDATES=*m*

MLUU=*m*

In some iterations, PROC NETFLOW must either perform a series of single column updates or a complete refactorization of the working basis matrix. More than one column of the working basis matrix must change before the next simplex iteration can begin. The single column updates can often be done faster than a complete refactorization, especially if few updates are necessary, the working basis matrix is sparse, or a refactorization has been performed recently. If the number of columns that must change is less than the value specified in the MAXLUUPDATES= option, the updates are attempted; otherwise, a refactorization is done. Refactorization also occurs if the sum of the number of columns that must be changed and the number of **LU** updates done since the last refactorization exceeds the value of the REFACTFREQ= option. The MAXLUUPDATES= option should not be used if the INVD_2D option is specified in the PROC NETFLOW statement.

In some iterations, a series of single column updates are not able to complete the changes required for a working basis matrix because, ideally, all columns should change at once. If the update cannot be completed, PROC NETFLOW performs a refactorization. The default value is 5.

NOINTFIRST
indicates that of the arcs eligible to be swapped with the leaving arc, the one chosen to do so has the best cost. See the INTFIRST option.

REFACTFREQ=*r*

RFF=*r*
specifies the maximum number of **L** and **U** updates between refactorization of the working basis matrix to reinitialize **LU** factors. In most iterations, one or several Bartels-Golub updates can be performed. An update is performed more quickly than a complete refactorization. However, after a series of updates, the sparsity of the **U** factor is degraded. A refactorization is necessary to regain sparsity and to make subsequent computations and updates more efficient. The default value is 50. The REFACTFREQ= option should not be used if INVD_2D is specified in the PROC NETFLOW statement.

U=*u*
controls the choice of pivot during **LU** decomposition or Bartels-Golub update. When searching for a pivot, any element less than the value of the U= option times the largest element in its matrix row is excluded, or matrix rows are interchanged to improve numerical stability. The U= option should have values on or between ZERO2 and 1.0. Decreasing the value of the U= option biases the algorithm toward maintaining sparsity at the expense of numerical stability and vice-versa. Reid (1975) suggests that the value of 0.01 is acceptable and this is the default for the U= option. This option should not be used if INVD_2D is specified in the PROC NETFLOW statement.

Pricing Strategy Options

There are three main types of pricing strategies:

- PRICETYPE*x*=NOQ
- PRICETYPE*x*=BLAND
- PRICETYPE*x*=Q

PRICETYPEx= option]PRICETYPE*x*= optionQSIZEx= option]QSIZE*x*= option

The one that usually performs better than the others is PRICETYPE*x*=Q, so this is the default.

Because the pricing strategy takes a lot of computational time, you should experiment with the following options to find the optimum specification. These options influence how the pricing step of the simplex iteration is performed. See the section "Pricing Strategies" on page 393 for further information.

PRICETYPE*x*=BLAND or PTYPE*x*=BLAND

PRICETYPE*x*=NOQ or PTYPE*x*=NOQ

- P*x*SCAN=BEST
- P*x*SCAN=FIRST
- P*x*SCAN=PARTIAL and P*x*NPARTIAL=*p*

PRICETYPE*x*=Q or PTYPE*x*=Q
QSIZE*x*=*q* or Q*x*=*q*
REFRESHQ*x*=*r*
REDUCEQSIZE*x*=*r*
REDUCEQ*x*=*r*

- P*x*SCAN=BEST
- P*x*SCAN=FIRST
- P*x*SCAN=PARTIAL and P*x*NPARTIAL=*p*
- Q*x*FILLSCAN=BEST
- Q*x*FILLSCAN=FIRST
- Q*x*FILLSCAN=PARTIAL and Q*x*FILLNPARTIAL=*q*

For stage 2 optimization, you can specify P2SCAN=ANY, which is used in conjunction with the DUALFREQ= option.

Miscellaneous Options

FUTURE1

signals that PROC NETFLOW must output extra observations to the NODEOUT= and ARCOUT= data sets. These observations contain information about the solution found by doing optimization neglecting any side constraints. These two data sets then can be used as the NODEDATA= and ARCDATA= data sets, respectively, in subsequent PROC NETFLOW runs with the WARM option specified. See the section "Warm Starts" on page 406.

FUTURE2

signals that PROC NETFLOW must output extra observations to the DUALOUT= and CONOUT= data sets. These observations contain information about the solution found by optimization that considers side constraints. These two data sets can then be used as the NODEDATA= data set (also called the DUALIN= data set) and the ARCDATA= data sets, respectively, in subsequent PROC NETFLOW runs with the WARM option specified. See the section "Warm Starts" on page 406.

MOREOPT

The MOREOPT option turns off all optimality and infeasibility flags that may have been raised. Unless this is done, PROC NETFLOW will not do any optimization when a RUN statement is specified.

If PROC NETFLOW determines that the problem is infeasible, it will not do any more optimization unless you specify MOREOPT in a RESET statement. At the same time, you can try resetting options (particularly zero tolerances) in the hope that the infeasibility was raised incorrectly.

Consider the following example:

```
proc netflow
   nodedata=noded          /* supply and demand data   */
   arcdata=arcd1           /* the arc descriptions     */
   condata=cond1           /* the side constraints     */
   conout=solution;        /* output the solution      */
   run;
/* Netflow states that the problem is infeasible.     */
```

```
/* You suspect that the zero tolerance is too large   */
   reset zero2=1.0e-10 moreopt;
   run;
/* Netflow will attempt more optimization.            */
/* After this, if it reports that the problem is      */
/* infeasible, the problem really might be infeasible */
```

If PROC NETFLOW finds an optimal solution, you might want to do additional optimization to confirm that an optimum has really been reached. Specify the MOREOPT option in a RESET statement. Reset options, but in this case tighten zero tolerances.

NOFUTURE1
negates the FUTURE1 option.

NOFUTURE2
negates the FUTURE2 option.

NOSCRATCH
negates the SCRATCH option.

NOZTOL1
indicates that the majority of tests for roundoff error should not be done. Specifying the NOZTOL1 option and obtaining the same optimal solution as when the NOZTOL1 option is not specified in the PROC NETFLOW statement (or the ZTOL1 option is specified), verifies that the zero tolerances were not too high. Roundoff error checks that are critical to the successful functioning of PROC NETFLOW and any related readjustments are always done.

NOZTOL2
indicates that the majority of tests for roundoff error are not to be done during an optimization that considers side constraints. The reasons for specifying the NOZTOL2 option are the same as those for specifying the NOZTOL1 option for stage 1 optimization (see the NOZTOL1 option).

OPTIM_TIMER
indicates that the procedure is to issue a message to the SAS log giving the CPU time spent doing optimization. This includes the time spent preprocessing, performing optimization, and postprocessing. Not counted in that time is the rest of the procedure execution, which includes reading the data and creating output SAS data sets.

The time spent optimizing can be small compared to the total CPU time used by the procedure. This is especially true when the problem is quite small (e.g., fewer than 10,000 variables).

SCRATCH
specifies that you do not want PROC NETFLOW to enter or continue stage 1 of the algorithm. Rather than specify RESET SCRATCH, you can use the CONOPT statement.

VERBOSE=*v*
limits the number of similar messages that are displayed on the SAS log.

For example, when reading the ARCDATA= data set, PROC NETFLOW might have cause to issue the following message many times:

```
ERROR: The HEAD list variable value in obs i in ARCDATA is
       missing and the TAIL list variable value of this obs
       is nonmissing. This is an incomplete arc specification.
```

If there are many observations that have this fault, messages that are similar are issued for only the first VERBOSE= such observations. After the ARCDATA= data set has been read, PROC NETFLOW will issue the message

```
NOTE: More messages similar to the ones immediately above
      could have been issued but were suppressed as
      VERBOSE=v.
```

If observations in the ARCDATA= data set have this error, PROC NETFLOW stops and you have to fix the data. Imagine that this error is only a warning and PROC NETFLOW proceeded to other operations such as reading the CONDATA= data set. If PROC NETFLOW finds there are numerous errors when reading that data set, the number of messages issued to the SAS log are also limited by the VERBOSE= option.

If you have a problem with a large number of side constraints and for some reason you stop stage 2 optimization early, PROC NETFLOW indicates that constraints are violated by the current solution. Specifying VERBOSE=*v* allows at most *v* violated constraints to be written to the log. If there are more, these are not displayed.

When PROC NETFLOW finishes and messages have been suppressed, the message

```
NOTE: To see all messages, specify VERBOSE=vmin.
```

is issued. The value of *vmin* is the smallest value that should be specified for the VERBOSE= option so that *all* messages are displayed if PROC NETFLOW is run again with the same data and everything else (except the VERBOSE= option) unchanged. No messages are suppressed.

The default value for the VERBOSE= option is 12.

ZERO1=*z*

Z1=*z*

specifies the zero tolerance level in stage 1. If the NOZTOL1 option is not specified, values within *z* of zero are set to 0.0, where *z* is the value of the ZERO1= option. Flows close to the lower flow bound or capacity of arcs are reassigned those exact values. Two values are deemed to be close if one is within *z* of the other. The default value for the ZERO1= option is 0.000001. Any value specified for the ZERO1= option that is < 0.0 or > 0.0001 is invalid.

ZERO2=*z*

Z2=*z*

specifies the zero tolerance level in stage 2. If the NOZTOL2 option is not specified, values within *z* of zero are set to 0.0, where *z* is the value of the ZERO2= option. Flows close to the lower flow bound or capacity of arcs are reassigned those exact values. If there are nonarc variables, values close to the lower or upper value bound of nonarc variables are reassigned those exact values. Two values are deemed to be close if one is within *z* of the other. The default value for the ZERO2= option is 0.000001. Any value specified for the ZERO2= option that is < 0.0 or > 0.0001 is invalid.

ZEROTOL=*z*

specifies the zero tolerance used when PROC NETFLOW must compare any real number with another real number, or zero. For example, if x and y are real numbers, then for x to be considered greater than y, x must be at least $y + z$. The ZEROTOL= option is used throughout any PROC NETFLOW run.

ZEROTOL=*z* controls the way PROC NETFLOW performs all double precision comparisons; that is, whether a double precision number is equal to, not equal to, greater than (or equal to), or less than (or equal to) zero or some other double precision number. A double precision number is deemed to be the same as another such value if the absolute difference between them is less than or equal to the value of the ZEROTOL= option.

The default value for the ZEROTOL= option is 1.0E−14. You can specify the ZEROTOL= option in the NETFLOW or RESET statement. Valid values for the ZEROTOL= option must be > 0.0 and < 0.0001. Do not specify a value too close to zero as this defeats the purpose of the ZEROTOL= option. Neither should the value be too large, as comparisons might be incorrectly performed.

The ZEROTOL= option is different from the ZERO1= and ZERO2= options in that ZERO1= and ZERO2= options work when determining whether optimality has been reached, whether an entry in the updated column in the ratio test of the simplex method is zero, whether a flow is the same as the arc's capacity or lower bound, or whether the value of a nonarc variable is at a bound. The ZEROTOL= option is used in all other general double precision number comparisons.

ZTOL1

indicates that all tests for roundoff error are performed during stage 1 optimization. Any alterations are carried out. The opposite of the ZTOL1 option is the NOZTOL1 option.

ZTOL2

indicates that all tests for roundoff error are performed during stage 2 optimization. Any alterations are carried out. The opposite of the ZTOL2 option is the NOZTOL2 option.

Interior Point Algorithm Options

FACT_METHOD=*f*

enables you to choose the type of algorithm used to factorize and solve the main linear systems at each iteration of the interior point algorithm.

FACT_METHOD=LEFT_LOOKING is new for SAS 9.1.2. It uses algorithms described in George, Liu, and Ng (2001). Left looking is one of the main methods used to perform Cholesky optimization and, along with some recently developed implementation approaches, can be faster and require less memory than other algorithms.

Specify FACT_METHOD=USE_OLD if you want the procedure to use the only factorization available prior to SAS 9.1.2.

TOLDINF=*t*

RTOLDINF=*t*

specifies the allowed amount of dual infeasibility. In the section "Interior Point Algorithmic Details" on page 429, the vector $infeas_d$ is defined. If all elements of this vector are $\leq t$, the solution is deemed feasible. $infeas_d$ is replaced by a zero vector, making computations faster. This option is the dual equivalent to the TOLPINF= option. Valid values for t are greater than 1.0E−12. The default is 1.0E−7.

TOLPINF=*t*

RTOLPINF=*t*

specifies the allowed amount of primal infeasibility. This option is the primal equivalent to the TOLDINF= option. In the section "Interior Point: Upper Bounds" on page 436, the vector $infeas_b$ is defined. In the section "Interior Point Algorithmic Details" on page 429, the vector $infeas_c$ is defined. If all elements in these vectors are $\leq t$, the solution is deemed feasible. $infeas_b$ and $infeas_c$ are replaced by zero vectors, making computations faster. Increasing the value of the TOLPINF= option too much can lead to instability, but a modest increase can give the algorithm added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

TOLTOTDINF=*t*

RTOLTOTDINF=*t*

specifies the allowed total amount of dual infeasibility. In the section "Interior Point Algorithmic Details" on page 429, the vector $infeas_d$ is defined. If $\sum_{i=1}^{n} infeas_{di} \leq t$, the solution is deemed feasible. $infeas_d$ is replaced by a zero vector, making computations faster. This option is the dual equivalent to the TOLTOTPINF= option. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

TOLTOTPINF=*t*

RTOLTOTPINF=*t*

specifies the allowed total amount of primal infeasibility. This option is the primal equivalent to the TOLTOTDINF= option. In the section "Interior Point: Upper Bounds" on page 436, the vector $infeas_b$ is defined. In the section "Interior Point Algorithmic Details" on page 429, the vector $infeas_c$ is defined. If $\sum_{i=1}^{n} infeas_{bi} \leq t$ and $\sum_{i=1}^{m} infeas_{ci} \leq t$, the solution is deemed feasible. $infeas_b$ and $infeas_c$ are replaced by zero vectors, making computations faster. Increasing the value of the TOLTOTPINF= option too much can lead to instability, but a modest increase can give the algorithm added flexibility and decrease the iteration count. Valid values for *t* are greater than 1.0E−12. The default is 1.0E−7.

CHOLTINYTOL=*c*

RCHOLTINYTOL=*c*

specifies the cut-off tolerance for Cholesky factorization of the $A\Theta A^{-1}$. If a diagonal value drops below *c*, the row is essentially treated as dependent and is ignored in the factorization. Valid values for *c* are between 1.0E−30 and 1.0E−6. The default value is 1.0E−8.

DENSETHR=*d*

RDENSETHR=*d*

specifies the density threshold for Cholesky processing. When the symbolic factorization encounters a column of *L* that has DENSETHR= proportion of nonzeros and the remaining part of *L* is at least 12×12, the remainder of *L* is treated as dense. In practice, the lower right part of the Cholesky triangular factor *L* is quite dense and it can be computationally more efficient to treat it as 100% dense. The default value for *d* is 0.7. A specification of $d \leq 0.0$ causes all dense processing; $d \geq 1.0$ causes all sparse processing.

PDSTEPMULT=*p*

RPDSTEPMULT=*p*

specifies the step-length multiplier. The maximum feasible step-length chosen by the Primal-Dual with Predictor-Corrector algorithm is multiplied by the value of the PDSTEPMULT= option. This number must be less than 1 to avoid moving beyond the barrier. An actual step length greater than 1 indicates numerical difficulties. Valid values for *p* are between 0.01 and 0.999999. The default value is 0.99995.

In the section "Interior Point Algorithmic Details" on page 429, the solution of the next iteration is obtained by moving along a direction from the current iteration's solution:

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where α is the maximum feasible step-length chosen by the interior point algorithm. If $\alpha \leq 1$, then α is reduced slightly by multiplying it by *p*. α is a value as large as possible but ≤ 1.0 and not so large that an x_i^{k+1} or s_i^{k+1} of some variable i is "too close" to zero.

PRSLTYPE=*p*

IPRSLTYPE=*p*

Preprocessing the linear programming problem often succeeds in allowing some variables and constraints to be temporarily eliminated from the LP that must be solved. This reduces the solution time and possibly also the chance that the optimizer will run into numerical difficulties. The task of preprocessing is inexpensive to do.

You control how much preprocessing to do by specifying PRSLTYPE=*p*, where *p* can be –1, 0, 1, 2, or 3.

–1 Do not perform preprocessing. For most problems, specifying PRSLTYPE= –1 is *not* recommended.

0 Given upper and lower bounds on each variable, the greatest and least contribution to the row activity of each variable is computed. If these are within the limits set by the upper and lower bounds on the row activity, then the row is redundant and can be discarded. Try to tighten the bounds on any of the variables it can. For example, if all coefficients in a constraint are positive and all variables have zero lower bounds, then the row's smallest contribution is zero. If the rhs value of this constraint is zero, then if the constraint type is = or $\leq$, all the variables in that constraint can be fixed to zero. These variables and the constraint can be removed. If the constraint type is $\geq$, the constraint is redundant. If the rhs is negative and the constraint is $\leq$, the problem is infeasible. If just one variable in a row is not fixed, use the row to impose an implicit upper or lower bound on the variable and then eliminate the row. The preprocessor also tries to tighten the bounds on constraint right-hand sides.

1 When there are exactly two unfixed variables with coefficients in an equality constraint, solve for one in terms of the other. The problem will have one less variable. The new matrix will have at least two fewer coefficients and one less constraint. In other constraints where both variables appear, two coefs are combined into one. PRSLTYPE=0 reductions are also done.

2 It may be possible to determine that an equality constraint is not constraining a variable. That is, if all variables are nonnegative, then $x - \sum_i y_i = 0$ does not constrain x, since it must be nonnegative if all the y_i's are nonnegative. In this case, eliminate x by subtracting this equation from all others containing x. This is useful when the only other entry for x is in the objective function. Perform this reduction if there is at most one other nonobjective coefficient. PRSLTYPE=0 reductions are also done.

3 All possible reductions are performed. PRSLTYPE=3 is the default.

Preprocessing is iterative. As variables are fixed and eliminated, and constraints are found to be redundant and they too are eliminated, and as variable bounds and constraint right-hand sides are tightened, the LP to be optimized is modified to reflect these changes. Another iteration of preprocessing of the modified LP may reveal more variables and constraints that can be eliminated.

PRINTLEVEL2=*p*

is used when you want to see PROC NETFLOW's progress to the optimum. PROC NETFLOW will produce a table on the SAS log. A row of the table is generated during each iteration and may consist of values of

- the affine step complementarity
- the complementarity of the solution for the next iteration
- the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436)
- the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429)
- the total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429)

As optimization progresses, the values in all columns should converge to zero. If you specify PRINTLEVEL2=2, all columns will appear in the table. If PRINTLEVEL2=1 is specified, only the affine step complementarity and the complementarity of the solution for the next iteration will appear. Some time is saved by not calculating the infeasibility values.

Interior Point Algorithm Options: Stopping Criteria

MAXITERB=*m*

IMAXITERB=*m*

specifies the maximum number of iterations of the interior point algorithm that can be performed. The default value for *m* is 100. One of the most remarkable aspects of the interior point algorithm is that for most problems, it usually needs to do a small number of iterations, *no matter the size of the problem*.

PDGAPTOL=*p*

RPDGAPTOL=*p*

specifies the primal-dual gap or duality gap tolerance. Duality gap is defined in the section "Interior Point Algorithmic Details" on page 429. If the relative gap $(duality\ gap/(c^T x))$ between the primal and dual objectives is smaller than the value of the PDGAPTOL= option and both the primal and dual problems are feasible, then PROC NETFLOW stops optimization with a solution that is deemed optimal. Valid values for *p* are between 1.0E−12 and 1.0E−1. The default is 1.0E−7.

STOP_C=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

STOP_DG=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

STOP_IB=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

STOP_IC=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is $\leq s$, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

STOP_ID=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is $\leq s$, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

AND_STOP_C=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, *and* the conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

AND_STOP_DG=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, *and* the conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

AND_STOP_IB=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $\leq s$, *and* the conditions related to other AND_STOP

parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

AND_STOP_IC=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is $\leq s$, *and* the conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

AND_STOP_ID=*s*
is used to determine whether optimization should stop. At the beginning of each iteration, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is $\leq s$, *and* the conditions related to other AND_STOP parameters are also satisfied, optimization will stop. This option is discussed in the section "Stopping Criteria" on page 433.

KEEPGOING_C=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $> s$, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

KEEPGOING_DG=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $> s$, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

KEEPGOING_IB=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is $> s$, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

KEEPGOING_IC=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is $> s$, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

KEEPGOING_ID=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is $> s$, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

AND_KEEPGOING_C=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if complementarity (the value of the Complem-ity column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, *and* the conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

AND_KEEPGOING_DG=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if the duality gap (the value of the Duality_gap column in the table produced when you specify PRINTLEVEL2=1 or PRINTLEVEL2=2) is > *s*, *and* the conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

AND_KEEPGOING_IB=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436; this value appears in the Tot_infeasb column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

AND_KEEPGOING_IC=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasc column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

AND_KEEPGOING_ID=*s*
is used to determine whether optimization should stop. If a stopping condition is met, if total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429; this value appears in the Tot_infeasd column in the table produced when you specify PRINTLEVEL2=2) is > *s*, *and* the conditions related to other AND_KEEPGOING parameters are also satisfied, optimization will continue. This option is discussed in the section "Stopping Criteria" on page 433.

RHS Statement

RHS *variable* **;**

The RHS variable list is used when the dense format of the CONDATA= data set is used. The values of the SAS variable specified in the RHS list are constraint right-hand-side values. If the RHS list is not specified, the CONDATA= data set is searched and a SAS variable with the name _RHS_ is used. If there is no RHS list and no SAS variable named _RHS_, all constraints are assumed to have zero right-hand-side values. The RHS list variable must have numeric values.

ROW Statement

ROW *variables* **;**

The ROW list is used when either the sparse or the dense format of side constraints is being used. SAS variables in the ROW list have values that are constraint or special row names. The SAS variables in the ROW list must have character values.

If the dense data format is used, there must be only one SAS variable in this list. In this case, if a ROW list is not specified, the CONDATA= data set is searched and the SAS variable with the name _ROW_ or _CON_ is used.

If the sparse data format is used and the ROW statement is not specified, the CONDATA= data set is searched and SAS variables with names beginning with _ROW or _CON are used. The number of SAS variables in the ROW list must not be less than the number of SAS variables in the COEF list. The *i*th ROW list variable is paired with the *i*th COEF list variable. If the number of ROW list variables is greater than the number of COEF list variables, the last ROW list variables have no COEF partner. These ROW list variables that have no corresponding COEF list variable are used in observations that have a TYPE list variable value. All ROW list variable values are tagged as having the type indicated. If there is no TYPE list variable, all ROW list variable values are constraint names.

RUN Statement

RUN ;

The RUN statement causes optimization to be started or resumed. The RUN statement has no options. If PROC NETFLOW is called and is not terminated because of an error or a QUIT statement, and you have not used a RUN statement, a RUN statement is assumed implicitly as the last statement of PROC NETFLOW. Therefore, PROC NETFLOW always performs optimization and saves the obtained (optimal) solution in the current output data sets.

SAVE Statement

SAVE *options* **;**

The options available with the SAVE statement of PROC NETFLOW are summarized by purpose in the following table.

Table 6.5 Functional Summary, SAVE Statement

Description	Statement	Option
Output Data Set Options:		
Unconstrained primal solution data set	SAVE	ARCOUT=
Unconstrained dual solution data set	SAVE	NODEOUT=
Constrained primal solution data set	SAVE	CONOUT=
Constrained dual solution data set	SAVE	DUALOUT=

The SAVE statement can be used to specify output data sets and create observations in these data sets. Use the SAVE statement if no optimization is to be performed before these output data sets are created.

The SAVE statement must be used to save solutions in data sets if there is no more optimization to do. If more optimization is to be performed, after which you want to save the solution, then do one of the following:

- Submit a RUN statement followed by a SAVE statement.
- Use the PROC NETFLOW or RESET statement to specify current output data sets. After optimization, output data sets are created and observations are automatically sent to the current output data sets.

Consider the following example:

```
proc netflow options; lists;
   reset maxit1=10 maxit2=25
         arcout=arcout0   nodeout=nodeout0
         conout=conout0   dualout=dualout0;
   run;
   /* Stage 1 optimization stops after iteration 10. */
   /* No output data sets are created yet.           */
   save arcout=arcout1 nodeout=nodeout1;
   /* arcout1 and nodeout1 are created.              */
   reset arcout=arcout2 maxit1=999999;
   run;
   /* The stage 1 optimum is reached.                */
   /* Arcout2 and nodeout0 are created.              */
   /* Arcout0 is not created as arcout=arcout2 over- */
   /* rides the arcout=arcout0 specified earlier.    */
   /* Stage 2 optimization stops after 25 iterations */
   /* as MAXIT2=25 was specified.                    */
   save conout=conout1;
   /* Conout1 is created.                            */
   reset maxit2=999999 dualout=null;
   run;
   /* The stage 2 optimum is reached.                */
   /* Conout0 is created.                            */
   /* No dualout is created as the last NETFLOW or   */
   /* reset statements dualout=data set specification*/
   /* was dualout=null.                              */
```

The data sets specified in the PROC NETFLOW and RESET statements are created when an optimal solution is found. The data sets specified in SAVE statements are created immediately.

The data sets in the preceding example are all distinct, but this need not be the case. The only exception to this is that the ARCOUT= data set and the NODEOUT= data set (or the CONOUT= data set and the DUALOUT= data set) that are being created at the same time must be distinct. Use the SHOW DATASETS statement to examine what data sets are current and when they were created.

The following options can appear in the SAVE statement:

ARCOUT=*SAS-data-set* (or **AOUT=***SAS-data-set*)

NODEOUT=*SAS-data-set* (or **NOUT=***SAS-data-set*)

CONOUT=*SAS-data-set* (or **COUT=***SAS-data-set*)

DUALOUT=*SAS-data-set* (or **DOUT=***SAS-data-set*)

SHOW Statement

SHOW *options / qualifiers* **;**

The options available with the SHOW statement of PROC NETFLOW are summarized by purpose in the following table.

Table 6.6 Functional Summary, SHOW Statement

Description	Statement	Option
SHOW Statement Options:		
Show problem, optimization status	SHOW	STATUS
Show network model parameters	SHOW	NETSTMT
Show data sets that have been or will be created	SHOW	DATASETS
Show options that pause optimization	SHOW	PAUSE
Show simplex algorithm options	SHOW	SIMPLEX
Show pricing strategy options	SHOW	PRICING
Show miscellaneous options	SHOW	MISC
SHOW Statement Qualifiers:		
Display information only on relevant options	SHOW	/ RELEVANT
Display options for current stage only	SHOW	/ STAGE

The SHOW statement enables you to examine the status of the problem and values of the RESET statement options. All output of the SHOW statement appears on the SAS log. The amount of information displayed when a SHOW statement is processed can be limited if some of the options of the SHOW statement are specified. These options indicate whether the problem status or a specific category of the RESET options is of interest. If no options are specified, the problem status and information on all RESET statement options in every category is displayed. The amount of displayed information can be limited further by following any SHOW statement options with a slash (/) and one or both qualifiers, RELEVANT and STAGE.

STATUS

produces one of the following optimization status reports, whichever is applicable. The warning messages are issued only if the network or entire problem is infeasible.

```
NOTE: Optimization Status.
      Optimization has not started yet.

NOTE: Optimization Status.
      Optimizing network (ignoring any side constraints).
      Number of iterations=17
      Of these, 3 were degenerate
```

```
WARNING: This optimization has detected that the network is
         infeasible.

NOTE: Optimization Status.
      Found network optimum (ignoring side constraints)
      Number of iterations=23
      Of these, 8 were degenerate

NOTE: Optimization Status.
      Optimizing side constrained network.
      Number of iterations=27
      Of these, 9 were degenerate
WARNING: This optimization has detected that the problem is
         infeasible.

NOTE: Optimization Status.
      Found side constrained network optimum
      Number of iterations=6
      Of these, 0 were degenerate
```

DATASETS

produces a report on output data sets.

```
NOTE: Current output SAS data sets
      No output data sets have been specified

NOTE: Current output SAS data sets
      ARCOUT=libname.memname
      NODEOUT=libname.memname
      CONOUT=libname.memname
      DUALOUT=libname.memname

NOTE: Other SAS data sets specified in previous ARCOUT=, NODEOUT=,
      CONOUT=, or DUALOUT=.
      libname.memname
              .
              .
              .

NOTE: Current output SAS data sets (SHOW DATASETS)
      libname.memname
              .
              .
              .
NOTE: SAS data sets specified as ARCOUT= NODEOUT= CONOUT= or
      DUALOUT= data sets in previous PROC NETFLOW, SET, RESET
      and SAVE statements.
      The number following the data set specification was the
      iteration number when observations were placed into the
      data set.
      libname.memname      iteration_number
              .                    .
              .                    .
```

. .

PAUSE

produces a report on the current settings of options used to make optimization pause.

```
NOTE: Options and parameters that stop optimization for reasons
      other than infeasibility or optimality (SHOW PAUSE)
      FEASIBLEPAUSE1=FALSE
      ENDPAUSE1=FALSE
      PAUSE1=999999
      MAXIT1=1000
      FEASIBLEPAUSE2=FALSE
      PAUSE2=999999
      MAXIT2=999999
```

SIMPLEX

produces the following:

```
NOTE: Options and parameters that control the primal simplex
      network algorithm (excluding those that affect the
      pricing strategies) (SHOW SIMPLEX)
      LRATIO1=FALSE
      BIGM1=NOTWOPHASE1=TRUE, TWOPHASE1=NOBIGM1=FALSE
      CYCLEMULT1=0.15
      PERTURB1=FALSE
      MINBLOCK1=2
      INTFIRST=TRUE
      LRATIO2=FALSE
      BIGM2=NOTWOPHASE2=TRUE, TWOPHASE2=NOBIGM2=FALSE
      REFACTFREQ=50
      U=0.1
      MAXLUUPDATES=6
      MAXL=40
```

PRICING

produces the following:

```
NOTE: Options and parameters that control the primal simplex
      network algorithm pricing strategies (SHOW PRICING)
      PRICETYPE1=Q
      P1SCAN=FIRST
      P1NPARTIAL=10
      Q1FILLSCAN=FIRST
      QSIZE1=24
      REFRESHQ1=0.75
      REDUCEQSIZE1=1
      Q1FILLNPARTIAL=10
      PRICETYPE2=Q
      P2SCAN=FIRST
      P2NPARTIAL=10
      DUALFREQ=4
```

```
Q2FILLSCAN=FIRST
QSIZE2=24
REFRESHQ2=0.75
REDUCEQSIZE2=1
Q2FILLNPARTIAL=10
```

MISC

produces the following:

```
NOTE: Miscellaneous options and parameters (SHOW MISC)
      VERBOSE=12
      ZTOL1=TRUE
      ZERO1=1E-6
      FUTURE1=FALSE
      ZTOL2=TRUE
      ZERO2=1E-6
      FUTURE2=FALSE
```

Following a slash (/), the qualifiers below can appear in any SHOW statement.

RELEVANT

indicates that you want information only on relevant options of the RESET statement. The following will *not* be displayed if / RELEVANT is specified:

- information on noncurrent data sets
- the options that control the reasons why stage 1 optimization should be halted and the options that control the simplex algorithm during stage 1 optimization, if the unconstrained optimum has been reached or constrained optimization has been performed
- if P1SCAN=BEST or P1SCAN=FIRST, the P1NPARTIAL= option is irrelevant
- if PRICETYPE1=BLAND or PRICETYPE1=NOQ, the options QSIZE1=, Q1FILLSCAN=, REFRESHQ1=, and REDUCEQSIZE1= are irrelevant
- if Q1FILLSCAN=BEST or Q1FILLSCAN=FIRST, the Q1FILLNPARTIAL= option is irrelevant
- the options that control the reasons stage 2 optimization should be halted, the options that control the simplex algorithm during stage 2 optimization, if the constrained optimum has been reached
- if P2SCAN=BEST or P2SCAN=FIRST, the P2NPARTIAL= option is irrelevant
- if PRICETYPE2=BLAND or PRICETYPE2=NOQ, the options QSIZE2=, Q2FILLSCAN=, REFRESHQ2=, and REDUCEQSIZE2= are irrelevant
- if Q2FILLSCAN=BEST or Q2FILLSCAN=FIRST, the Q2FILLNPARTIAL= option is irrelevant

STAGE

indicates that you want to examine only the options that affect the optimization that is performed if a RUN statement is executed next. Before any optimization has been done, only stage 2 options are displayed if the problem has side constraints and the SCRATCH option is used, or if the CONOPT statement is specified. Otherwise, stage 1 options are displayed. If still optimizing neglecting constraints, only stage 1 options will be displayed. If the unconstrained optimum has been reached and

optimization that considers constraints has not been performed, stage 1 options are displayed. If the problem has constraints, stage 2 options are displayed. If any optimization that considers constraints has been performed, only stage 2 options are displayed.

SUPDEM Statement

SUPDEM *variable* **;**

The SAS variable in this list, which must be present in the NODEDATA= data set, contains supply and demand information for the nodes in the NODE list. A positive SUPDEM list variable value s ($s > 0$) denotes that the node named in the NODE list variable can supply s units of flow. A negative SUPDEM list variable value $-d$ ($d > 0$) means that this node demands d units of flow. If a SAS variable is not explicitly specified, a SAS variable with the name _SUPDEM_ or _SD_ in the NODEDATA= data set is used as the SUPDEM variable. If a node is a transshipment node (neither a supply nor a demand node), an observation associated with this node need not be present in the NODEDATA= data set. If present, the SUPDEM list variable value must be zero or a missing value.

SUPPLY Statement

SUPPLY *variable* **;**

The SUPPLY statement identifies the SAS variable in the ARCDATA= data set that contains the supply at the node named in that observation's TAILNODE list variable. If a tail node does not supply flow, use zero or a missing value for the observation's SUPPLY list variable value. If a tail node has supply capability, a missing value indicates that the supply quantity is given in another observation. It is not necessary to have a SUPPLY statement if the name of this SAS variable is _SUPPLY_.

TAILNODE Statement

TAILNODE *variable* **;**

TAIL *variable* **;**

FROMNODE *variable* **;**

FROM *variable* **;**

The TAILNODE statement specifies the SAS variable that must be present in the ARCDATA= data set that has as values the names of tail nodes of arcs. The TAILNODE variable must have character values. It is not necessary to have a TAILNODE statement if the name of the SAS variable is _TAIL_ or _FROM_. If the TAILNODE list variable value is missing, it is assumed that the observation of ARCDATA= data set contains information concerning a nonarc variable.

TYPE Statement

TYPE *variable* **;**

CONTYPE *variable* **;**

The TYPE list, which is optional, names the variable that has as values keywords that indicate either the constraint type for each constraint or the type of special rows in the CONDATA= data set. The values of the TYPE list variable also indicate, in each observation of the CONDATA= data set, how values of the VAR or COEF list variables are to be interpreted and how the type of each constraint or special row name is determined. If the TYPE list is not specified, the CONDATA= data set is searched and a SAS variable with the name _TYPE_ is used. Valid keywords for the TYPE variable are given below. If there is no TYPE statement and no other method is used to furnish type information (see the DEFCONTYPE= option), all constraints are assumed to be of the type "less than or equal to" and no special rows are used. The TYPE list variable must have character values and can be used when the data in the CONDATA= data set is in either the sparse or the dense format. If the TYPE list variable value has a * as its first character, the observation is ignored because it is a comment observation.

TYPE List Variable Values

The following are valid TYPE list variable values. The letters in boldface denote the characters that PROC NETFLOW uses to determine what type the value suggests. You need to have at least these characters. In the following list, the minimal TYPE list variable values have additional characters to aid you in remembering these values.

<	less than or equal to ($\leq$)
=	equal to (=)
>	greater than or equal to ($\geq$)
CAPAC	capacity
COST	cost
EQ	equal to
FREE	free row (used only for linear programs solved by interior point)
GAIN	gain in arc flow (generalized networks)
GE	greater than or equal to
LE	less than or equal to
LOSS	loss in arc flow (generalized networks)
LOWERBD	lower flow or value bound
LOW*blank*	lower flow or value bound
MAXIMIZE	maximize (opposite of cost)
MINIMIZE	minimize (same as cost)
MULT	value of arc multiplier (generalized networks)
OBJECTIVE	objective function (same as cost)
RHS	rhs of constraint
TYPE	type of constraint
UPPCOST	reserved for future use
UNREST	unrestricted variable (used only for linear programs solved by interior point)
UPPER	upper value bound or capacity; second letter must not be N

The valid TYPE list variable values in function order are

- **LE** less than or equal to ($\leq$)
- **EQ** equal to (=)
- **GE** greater than or equal to ($\geq$)
- **CO**ST
  MINIMIZE
  MAXIMIZE
  OBJECTIVE
  cost or objective function coefficient
- **CA**PAC
  UPPER
  capacity or upper value bound
- **LOW**ERBD
  LOW*blank*
  lower flow or value bound
- **R**HS rhs of constraint
- **T**YPE type of constraint
- **MU**LT
  GAIN
  LOSS
  value of arc multiplier in a generalized network

A TYPE list variable value that has the first character $*$ causes the observation to be treated as a comment. If the first character is a negative sign, then $\leq$ is the type. If the first character is a zero, then = is the type. If the first character is a positive number, then $\geq$ is the type.

VAR Statement

VAR *variables* **;**

The VAR variable list is used when the dense data format is used. The names of these SAS variables are also names of the arc and nonarc variables that have data in the CONDATA= data set. If no explicit VAR list is specified, all numeric variables not on other lists are put onto the VAR list. The VAR list variables must have numeric values. The values of the VAR list variables in some observations can be interpreted differently than in other observations. The values can be coefficients in the side constraints, costs and objective function coefficients, or bound data. How these numeric values are interpreted depends on the value of each observation's TYPE or ROW list variable value. If there are no TYPE list variables, the VAR list variable values are all assumed to be side constraint coefficients.

Details: NETFLOW Procedure

Input Data Sets

PROC NETFLOW is designed so that there are as few rules as possible that you must obey when inputting a problem's data. Raw data are acceptable. This should cut the amount of processing required to groom the data before it is input to PROC NETFLOW. Data formats are so flexible that, due to space restrictions, all possible forms for a problem's data are not shown here. Try any reasonable form for your problem's data; it should be acceptable. PROC NETFLOW will outline its objections.

There are several ways to supply the same piece of data. You do not have to restrict yourself to using any particular one. If you use several ways, PROC NETFLOW checks that the data are consistent each time the data are encountered. After all input data sets have been read, data are merged so that the problem is described completely. The order of the observations is not important in any of the input data sets.

ARCDATA= Data Set

See the section "Getting Started: NETFLOW Procedure" on page 314 and the section "Introductory Example" on page 315 for a description of this input data set.

NOTE: Information for an arc or nonarc variable can be specified in more than one observation. For example, consider an arc directed from node A toward node B that has a cost of 50, capacity of 100, and lower flow bound of 10 flow units. Some possible observations in the ARCDATA= data set may be

TAIL	_HEAD_	_COST_	_CAPAC_	_LO_
A	B	50	.	.
A	B	.	100	.
A	B	.	.	10
A	B	50	100	.
A	B	.	100	10
A	B	50	.	10
A	B	50	100	10

Similarly, for a nonarc variable with upperbd=100, lowerbd=10, and objective function coefficient=50, the _TAIL_ and _HEAD_ values are missing.

CONDATA= Data Set

Regardless of whether the data in the CONDATA= data set is in the sparse or dense format, you will receive a warning if PROC NETFLOW finds a constraint row that has no coefficients. You will also be warned if any nonarc variable has no constraint coefficients.

Dense Input Format

If the dense format is used, most SAS variables in the CONDATA= data set belong to the VAR list and have names of arc and nonarc variables. These names can be values of the NAME list SAS variables in the ARCDATA= data set, or names of nonarc variables, or names in the form *tail_head*, or any combination of these three forms. Names in the form *tail_head* are default arc names, and if you use them, you must specify node names in the ARCDATA= data set (values of the TAILNODE and HEADNODE list SAS variables) using no lowercase letters.

There can be three other variables in the CONDATA= data set, belonging, respectively, to the ROW, TYPE, and RHS lists. the section "Introductory Example" on page 315 uses the dense data format.

Consider the SAS code that creates a dense format CONDATA= data set that has data for three constraints. This data set was used in the section "Introductory Example" on page 315.

```
data cond1;
   input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
      _type_ $ _rhs_;
   datalines;
-2  .  1 .  . . >= -15
 . -2  . .  1 . GE -15
 .  . -3 4  . . EQ   0
 .  .  . . -3 4  =   0
;
```

You can use nonconstraint type values to furnish data on costs, capacities, lower flow bounds (and, if there are nonarc variables, objective function coefficients and upper and lower bounds). You need not have such (or as much) data in the ARCDATA= data set. The first three observations in the following data set are examples of observations that provide cost, capacity and lower bound data.

```
data cond1b;
   input m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
         _type_ $ _rhs_;
   datalines;
63 81 200   . 220   . cost   .
95 80 175 140 100 100 capac  .
20 10  50   .  35   . lo     .
-2  .   1   .   .   . >=   -15
 . -2   .   .   1   . GE   -15
 .  .  -3   4   .   . EQ     0
 .  .   .   .  -3   4  =     0
;
```

If a ROW list variable is used, the data for a constraint can be spread over more than one observation. To illustrate, the data for the first constraint, (which is called con1), and the cost and capacity data (in special rows called costrow and caprow, respectively) are spread over more than one observation in the following data set.

```
data cond1c;
   input _row_ $
         m_e_ref1 m_e_ref2 thruput1 r1_gas thruput2 r2_gas
         _type_ $ _rhs_;
   datalines;
costrow 63  .   .   .   .   . .      .
costrow  . 81 200   .   .   . cost   .
.        .  .   .   . 220   . cost   .
caprow   .  .   .   .   .   . capac  .
caprow  95  . 175   . 100 100 .      .
caprow   . 80 175 140   .   . .      .
lorow   20 10  50   .  35   . lo     .
con1    -2  .   1   .   .   . .      .
con1     .  .   .   .   .   . >=   -15
con2     . -2   .   .   1   . GE   -15
```

```
con3      .   .  -3   4    .    . EQ      0
con4      .   .   .   .   -3   4  =      0
;
```

Using both ROW and TYPE lists, you can use special row names. Examples of these are "costrow" and "caprow" in the last data set. It should be restated that in any of the input data sets of PROC NETFLOW, the order of the observations does not matter. However, the CONDATA= data set can be read more quickly if PROC NETFLOW knows what type of constraint or special row a ROW list variable value is. For example, when the first observation is read, PROC NETFLOW does not know whether costrow is a constraint or special row and how to interpret the value 63 for the arc with the name m_e_ref1. When PROC NETFLOW reads the second observation, it learns that costrow has type cost and that the values 81 and 200 are costs. When the entire CONDATA= data set has been read, PROC NETFLOW knows the type of all special rows and constraints. Data that PROC NETFLOW had to set aside (such as the first observation 63 value and the costrow ROW list variable value, which at the time had unknown type, but is then known to be a cost special row) is reprocessed. During this second pass, if a ROW list variable value has unassigned constraint or special row type, it is treated as a constraint with DEFCONTYPE= (or DEFCONTYPE= default) type. Associated VAR list variable values as coefficients of that constraint.

Sparse Input Format

The side constraints usually become sparse as the problem size increases. When the sparse data format of the CONDATA= data set is used, only nonzero constraint coefficients must be specified. Remember to specify the SPARSECONDATA option in the PROC NETFLOW statement. With the sparse method of specifying constraint information, the names of arc and nonarc variables do not have to be valid SAS variable names.

A sparse format CONDATA= data set for the oil industry example in the section "Introductory Example" on page 315 is displayed in the following code.

```
title 'Setting Up Condata = Cond2 for PROC NETFLOW';
data cond2;
   input _column_ $ _row1 $ _coef1 _row2 $ _coef2 ;
   datalines;
m_e_ref1  con1  -2     .    .
m_e_ref2  con2  -2     .    .
thruput1  con1   1  con3  -3
r1_gas       .   .  con3   4
thruput2  con2   1  con4  -3
r2_gas       .   .  con4   4
_type_    con1   1  con2   1
_type_    con3   0  con4   0
_rhs_     con1 -15  con2 -15
;
```

Recall that the COLUMN list variable values "_type_" and "_rhs_" are the default values of the TYPEOBS= and RHSOBS= options. Also, the default rhs value of constraints (con3 and con4) is zero. The third to last observation has the value "_type_" for the COLUMN list variable. The _ROW1 variable value is con1, and the _COEF1_ variable has the value 1. This indicates that the constraint con1 is *greater than* or equal to type (because the value 1 is *greater than* zero). Similarly, the data in the second to last observation's _ROW2 and _COEF2 variables indicate that con2 is an *equality* constraint (0 *equals* zero).

An alternative, using a TYPE list variable is as follows:

```
title 'Setting Up Condata = Cond3 for PROC NETFLOW';
data cond3;
   input _column_ $ _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
m_e_ref1  con1  -2     .    . >=
m_e_ref2  con2  -2     .    . .
thruput1  con1   1  con3  -3 .
r1_gas       .   .  con3   4 .
thruput2  con2   1  con4  -3 .
r2_gas       .   .  con4   4 .
.         con3   .  con4   . eq
.         con1 -15  con2 -15 ge
;
```

If the COLUMN list variable is missing in a particular observation (the last two observations in the data set cond3, for instance), the constraints named in the ROW list variables all have the constraint type indicated by the value in the TYPE list variable. It is for this type of observation that you are allowed more ROW list variables than COEF list variables. If corresponding COEF list variables are not missing (for example, the last observation in the data set cond3), these values are the rhs values of those constraints. Therefore, you can specify both constraint type and rhs in the same observation.

As in the previous CONDATA= data set, if the COLUMN list variable is an arc or nonarc variable, the COEF list variable values are coefficient values for that arc or nonarc variable in the constraints indicated in the corresponding ROW list variables. If in this same observation, the TYPE list variable contains a constraint type, all constraints named in the ROW list variables in that observation have this constraint type (for example, the first observation in the data set cond3). Therefore, you can specify both constraint type and coefficient information in the same observation.

Also note that DEFCONTYPE=EQ could have been specified, saving you from having to include in the data that CON3 and CON4 are of this type.

In the oil industry example, arc costs, capacities, and lower flow bounds are presented in the ARCDATA= data set. Alternatively, you could have used the following input data sets.

```
title3 'Setting Up Arcdata = Arcd2 for PROC NETFLOW';
data arcd2;
   input  _from_&$11. _to_&$15. ;
   datalines;
middle east  refinery 1
middle east  refinery 2
u.s.a.       refinery 1
u.s.a.       refinery 2
refinery 1   r1
refinery 2   r2
r1           ref1 gas
r1           ref1 diesel
r2           ref2 gas
r2           ref2 diesel
ref1 gas     servstn1 gas
ref1 gas     servstn2 gas
ref1 diesel  servstn1 diesel
ref1 diesel  servstn2 diesel
```

```
ref2 gas     servstn1 gas
ref2 gas     servstn2 gas
ref2 diesel  servstn1 diesel
ref2 diesel  servstn2 diesel
;
title 'Setting Up Condata = Cond4 for PROC NETFLOW';
data cond4;
   input _column_&$27. _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
.                              con1 -15   con2 -15    ge
.                           costrow   .      .   .  cost
.                                 .   . caprow   . capac
middle east_refinery 1        con1  -2      .   .     .
middle east_refinery 2        con2  -2      .   .     .
refinery 1_r1                 con1   1   con3  -3     .
r1_ref1 gas                      .   .   con3   4     =
refinery 2_r2                 con2   1   con4  -3     .
r2_ref2 gas                      .   .   con4   4    eq
middle east_refinery 1     costrow  63 caprow  95     .
middle east_refinery 2     costrow  81 caprow  80     .
u.s.a._refinery 1          costrow  55      .   .     .
u.s.a._refinery 2          costrow  49      .   .     .
refinery 1_r1              costrow 200 caprow 175     .
refinery 2_r2              costrow 220 caprow 100     .
r1_ref1 gas                      .   . caprow 140     .
r1_ref1 diesel                   .   . caprow  75     .
r2_ref2 gas                      .   . caprow 100     .
r2_ref2 diesel                   .   . caprow  75     .
ref1 gas_servstn1 gas      costrow  15 caprow  70     .
ref1 gas_servstn2 gas      costrow  22 caprow  60     .
ref1 diesel_servstn1 diesel costrow 18      .   .     .
ref1 diesel_servstn2 diesel costrow 17      .   .     .
ref2 gas_servstn1 gas      costrow  17 caprow  35     .
ref2 gas_servstn2 gas      costrow  31      .   .     .
ref2 diesel_servstn1 diesel costrow 36      .   .     .
ref2 diesel_servstn2 diesel costrow 23      .   .     .
middle east_refinery 1           .  20      .   .    lo
middle east_refinery 2           .  10      .   .    lo
refinery 1_r1                    .  50      .   .    lo
refinery 2_r2                    .  35      .   .    lo
ref2 gas_servstn1 gas            .   5      .   .    lo
;
```

The first observation in the cond4 data set defines con1 and con2 as *greater than or equal to* ($\geq$) constraints that both (by coincidence) have rhs values of -15. The second observation defines the special row costrow as a cost row. When costrow is a ROW list variable value, the associated COEF list variable value is interpreted as a cost or objective function coefficient. PROC NETFLOW has to do less work if constraint names and special rows are defined in observations near the top of a data set, but this is not a strict requirement. The fourth to ninth observations contain constraint coefficient data. Observations 7 and 9 have TYPE list variable values that indicate that constraints con3 and con4 are equality constraints. The last five observations contain lower flow bound data. Observations that have an arc or nonarc variable name in the COLUMN list variable, a nonconstraint type TYPE list variable value, and a value in (one of) the COEF list variables are valid.

The following data set is equivalent to the cond4 data set.

```
title 'Setting Up Condata = Cond5 for PROC NETFLOW';
data cond5;
   input _column_&$27. _row1 $ _coef1 _row2 $ _coef2 _type_ $ ;
   datalines;
middle east_refinery 1          con1  -2 costrow  63     .
middle east_refinery 2          con2  -2   lorow  10     .
refinery 1_r1                      .   .    con3  -3     =
r1_ref1 gas                   caprow 140    con3   4     .
refinery 2_r2                   con2   1    con4  -3     .
r2_ref2 gas                        .   .    con4   4    eq
.                               CON1 -15    CON2 -15    GE
ref2 diesel_servstn1 diesel        .  36 costrow   .  cost
.                                  .   .  caprow   . capac
.                              lorow   .       .   .    lo
middle east_refinery 1        caprow  95   lorow  20     .
middle east_refinery 2        caprow  80 costrow  81     .
u.s.a._refinery 1                  .   .       .  55  cost
u.s.a._refinery 2             costrow  49       .   .     .
refinery 1_r1                   con1   1  caprow 175     .
refinery 1_r1                  lorow  50 costrow 200     .
refinery 2_r2                 costrow 220  caprow 100     .
refinery 2_r2                      .  35       .   .    lo
r1_ref1 diesel                caprow2  75       .   . capac
r2_ref2 gas                        .   .  caprow 100     .
r2_ref2 diesel                caprow2  75       .   .     .
ref1 gas_servstn1 gas         costrow  15  caprow  70     .
ref1 gas_servstn2 gas         caprow2  60 costrow  22     .
ref1 diesel_servstn1 diesel        .   . costrow  18     .
ref1 diesel_servstn2 diesel   costrow  17       .   .     .
ref2 gas_servstn1 gas         costrow  17   lorow   5     .
ref2 gas_servstn1 gas              .   . caprow2  35     .
ref2 gas_servstn2 gas              .  31       .   .  cost
ref2 diesel_servstn2 diesel        .   . costrow  23     .
;
```

If you have data for a linear programming program that has an embedded network, the steps required to change that data into a form that is acceptable by PROC NETFLOW are

1. Identify the nodal flow conservation constraints. The coefficient matrix of these constraints (a submatrix of the LP's constraint coefficient matrix) has only two nonzero elements in each column, -1 and 1.

2. Assign a node to each nodal flow conservation constraint.

3. The rhs values of conservation constraints are the corresponding node's supplies and demands. Use this information to create a NODEDATA= data set.

4. Assign an arc to each column of the flow conservation constraint coefficient matrix. The arc is directed from the node associated with the row that has the 1 element in it and directed toward to the node associated with the row that has the -1 element in it. Set up an ARCDATA= data set that has two SAS variables. This data set could resemble ARCDATA=arcd2. These will eventually be the TAILNODE and HEADNODE list variables when PROC NETFLOW is used. Each observation consists of the tail and head node of each arc.

5. Remove from the data of the linear program all data concerning the nodal flow conservation constraints.
6. Put the remaining data into a CONDATA= data set. This data set will probably resemble CONDATA=cond4 or CONDATA=cond5.

The Sparse Format Summary

The following list illustrates possible CONDATA= data set observation sparse formats. a1, b1, b2, b3 and c1 have as a _COLUMN_ variable value either the name of an arc (possibly in the form *tail_head*) or the name of a nonarc variable.

- If there is no TYPE list variable in the CONDATA= data set, the problem must be constrained and there is no nonconstraint data in the CONDATA= data set.

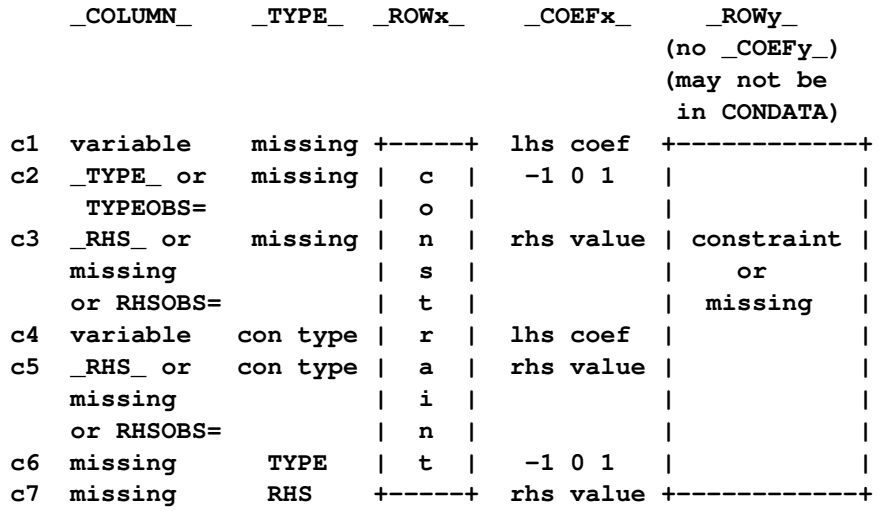
	COLUMN	_ROWx_	_COEFx_	_ROWy_ (no _COEFy_) (may not be in CONDATA)
a1	variable	constraint	lhs coef	
a2	_TYPE_ or TYPEOBS=	constraint	-1 0 1	
a3	_RHS_ or RHSOBS= or missing	constraint	rhs value	constraint or missing
a4	_TYPE_ or TYPEOBS=	constraint	missing	
a5	_RHS_ or RHSOBS= or missing	constraint	missing	

  Observations of the form a4 and a5 serve no useful purpose but are still allowed to make problem generation easier.

- If there are no ROW list variables in the data set, the problem has no constraints and the information is nonconstraint data. There must be a TYPE list variable and only one COEF list variable in this case. The COLUMN list variable has as values the names of arcs or nonarc variables and must not have missing values or special row names as values.

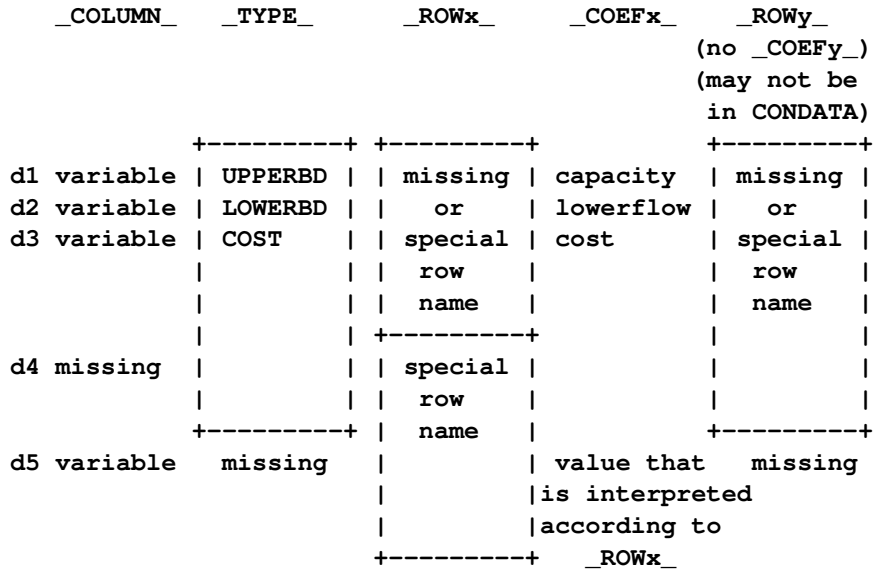
	COLUMN	_TYPE_	_COEFx_
b1	variable	UPPERBD	capacity
b2	variable	LOWERBD	lower flow
b3	variable	COST	cost

- Using a TYPE list variable for constraint data implies the following:



If the observation is of the form c4 or c5, and the _COEFx_ values are missing, the constraint is assigned the type data specified in the _TYPE_ variable.

- Using a TYPE list variable for arc and nonarc variable data implies the following:



Observations with form d1 to d5 can have ROW list variable values. Observation d4 must have ROW list variable values. The ROW value is put into the ROW name tree so that when dealing with observation d4 or d5, the COEF list variable value is interpreted according to the type of ROW list variable value. For example, the following three observations define the _ROWx_ variable values up_row, lo_row and co_row as being an upper value bound row, lower value bound row, and cost row, respectively.

```
_COLUMN_          _TYPE_          _ROWx_          _COEFx_

   .              UPPERBD         up_row             .
variable_a        LOWERBD         lo_row          lower flow
variable_b        COST            co_row          cost
```

PROC NETFLOW is now able to correctly interpret the following observation:

```
_COLUMN_ _TYPE_ _ROW1_ _COEF1_ _ROW2_ _COEF2_ _ROW3_ _COEF3_

 var_c      .    up_row  upval  lo_row  loval  co_row   cost
```

If the TYPE list variable value is a constraint type and the value of the COLUMN list variable equals the value of the TYPEOBS= option or the default value _TYPE_, the TYPE list variable value is ignored.

NODEDATA= Data Set

See the section "Getting Started: NETFLOW Procedure" on page 314 and the section "Introductory Example" on page 315 for a description of this input data set.

Output Data Sets

The procedure determines the flow that should pass through each arc as well as the value assigned to each nonarc variable. The goal is that the minimum flow bounds, capacities, lower and upper value bounds, and side constraints are not violated. This goal is reached when total cost incurred by such a flow pattern and value assignment is feasible and optimal. The solution found must also conserve flow at each node.

The ARCOUT= data set contains a solution obtained when performing optimization that does not consider any constraints. The NODEOUT= data set contains nodal dual variable information for this type of solution. You can choose to have PROC NETFLOW create the ARCOUT= data set and the NODEOUT= data set and save the optimum of the network or the nodal dual variable values before any optimization that considers the side constraints is performed.

If there are side constraints, the CONOUT= data set can be produced and contains a solution obtained after performing optimization that considers constraints. The DUALOUT= data set contains dual variable information for nodes and side constraints from the solution obtained after optimization that considers the constraints. The CONOUT= data set and DUALOUT= data set can be used to save the constrained optimal solution.

ARCOUT= and CONOUT= Data Sets

The ARCOUT= and CONOUT= data sets contain the same variables. Furthermore, the variables in the output data sets depend on whether or not the problem has a network component.

If the problem has a network component, the variables and their possible values in an observation are as follows:

FROM	a tail node of an arc. This is a missing value if an observation has information about a nonarc variable.
_TO_	a head node of an arc. This is a missing value if an observation has information about a nonarc variable.
COST	the cost of an arc or the objective function coefficient of a nonarc variable
CAPAC	the capacity of an arc or upper value bound of a nonarc variable
_LO_	the lower flow bound of an arc or lower value bound of a nonarc variable
NAME	a name of an arc or nonarc variable
SUPPLY	the supply of the tail node of the arc in the observation. This is a missing value if an observation has information about a nonarc variable.
DEMAND	the demand of the head node of the arc in the observation. This is a missing value if an observation has information about a nonarc variable.
FLOW	the flow through the arc or value of the nonarc variable
FCOST	flow cost, the product of _COST_ and _FLOW_
RCOST	the reduced cost of the arc or nonarc variable
ANUMB	the number of the arc (positive) or nonarc variable (nonpositive); used for warm starting PROC NETFLOW
TNUMB	the number of the tail node in the network basis spanning tree; used for warm starting PROC NETFLOW
STATUS	the status of the arc or nonarc variable

If the problem does not have a network component, the variables and their possible values in an observation are as follows:

OBJFN	the objective function coefficient of a variable
_UPPERBD	the upper value bound of a variable
_LOWERBD	the lower value bound of a variable
NAME	the name of a variable
VALUE	the value of the variable
FCOST	objective function value for that variable; the product of _OBJFN_ and _VALUE_

The variables present in the ARCDATA= data set are present in an ARCOUT= data set or a CONOUT= data set. For example, if there is a variable called tail in the ARCDATA= data set and you specified the SAS variable list

```
from tail;
```

then tail is a variable in the ARCOUT= and CONOUT= data sets instead of _FROM_. Any ID list variables also appear in the ARCOUT= and CONOUT= data sets.

NODEOUT= and DUALOUT= Data Sets

There are two types of observations in the NODEOUT= and DUALOUT= data sets. One type of observation contains information about a node. These are called *type N* observations. There is one such observation of this type for each node. The _NODE_ variable has a name of a node, and the _CON_ variable values in these observations are missing values.

The other type of observation contains information about constraints. These are called the *type C* observations. There is one such observation for each constraint. The _CON_ variable has a name of a constraint, and the _NODE_ variable values in these observations are missing values.

Many of the variables in the NODEOUT= and DUALOUT= data sets contain information used to warm start PROC NETFLOW. The variables _NODE_, _SD_, _DUAL_, _VALUE_, _RHS_, _TYPE_, and _CON_ contain information that might be of interest to you.

The NODEOUT= and DUALOUT= data sets look similar, as the same variables are in both. These variables and their values in an observation of each type are

NODE	Type N: the node name Type C: a missing value
_SD_	Type N: the supply (positive) or demand (negative) of the node Type C: a missing value
DUAL	Type N: the dual variable value of the node in _NODE_ Type C: the dual variable value of the constraint named in _CON_
NNUMB	Type N: the number of the node named in _NODE_ Type C: the number of the constraint named in _CON_
PRED	Type N: the predecessor in the network basis spanning tree of the node named in _NODE_ Type C: the number of the node toward which the arc with number in _ARCID_ is directed, or the constraint number associated with the slack, surplus, or artificial variable basic in this row
TRAV	Type N: the traversal thread label of the node named in _NODE_ Type C: a missing value
SCESS	Type N: the number of successors (including itself) in the network basis spanning tree of the node named in _NODE_ Type C: a missing value
ARCID	Type N: if _ARCID_ is nonnegative, _ARCID_ is the number of the network basis spanning tree arc directed from the node with number _PRED_ to the node named in _NODE_. If _ARCID_ is negative, minus _ARCID_ is the number of the network basis spanning tree arc directed from the node named in _NODE_ to the node with number _PRED_.

	Type C: if _ARCID_ is positive, _ARCID_ is the number of the arc basic in a constraint row. If nonpositive, minus _ARCID_ is the number of the nonarc variable basic in a constraint row.
FLOW	Type N: the flow minus the lower flow bound of the arc _ARCID_ Type C: the flow minus lower flow bound of the arc _ARCID_ or value lower bound of the nonarc variable value minus _ARCID_
FBQ	Type N: If _FBQ_ is positive, then _FBQ_ is the subscript in arc length arrays of the first arc directed toward the node named in _NODE_. PROC NETFLOW's arc length arrays are sorted so that data of arcs directed toward the same head node are together. If _FBQ_ is negative, no arcs are directed toward the node named in _NODE_. Arcs directed toward node i have subscripts in the arc length arrays between observations FBQ(i) and (FBQ($i+1$))-1, inclusive. Type C: a missing value
VALUE	Type N: a missing value Type C: the lhs value (the sum of the products of coefficient and flows or values) of the constraint named in _CON_
RHS	Type N: a missing value Type C: the rhs value of the constraint named in _CON_
TYPE	Type N: a missing value Type C: the type of the constraint named in _CON_
CON	Type N: a missing value Type C: the name of the constraint

If specified in variable lists, the variables in the input data sets are used instead of some of the previous variables. These variables are specified in the NODE, SUPDEM, RHS, TYPE, and ROW (if there is only one variable in the ROW list) lists and are used instead of _NODE_, _SD_, _RHS_, _TYPE_, and _CON_, respectively.

MPSOUT= Data Set

The MPSOUT= data set contains problem data converted from a PROC NETFLOW format into an MPS-format SAS data set. The six fields, FIELD1 to FIELD6, in the MPSOUT= data set correspond to the six columns in MPS standard. For more information about the MPS-format SAS data set, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

Converting Any PROC NETFLOW Format to an MPS-Format SAS Data Set

The MPSOUT= option enables you to convert an input data set for the NETFLOW procedure into an MPS-format SAS data set. The converted data set is readable by the OPTLP procedure.

The conversion can handle linear programs and network flow formulations. If you specify a network flow formulation, it will be converted into an equivalent linear program. When multiple objective row names are present, rows with the name encountered first are combined into the objective row. The remaining rows are marked as free rows.

For information about how the contents of the MPS-format SAS data set are interpreted, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*).

For an example demonstrating the use of the MPSOUT= option, see Example 6.15. For examples that demonstrate how to migrate to the OPTMODEL procedure, see the section "Examples: NETFLOW Procedure" on page 466.

Case Sensitivity

Whenever the NETFLOW procedure has to compare character strings, whether they are node names, arc names, nonarc names, or constraint names, if the two strings have different lengths, or on a character by character basis the character is different *or has different cases*, PROC NETFLOW judges the character strings to be different.

Not only is this rule enforced when one or both character strings are obtained as values of SAS variables in PROC NETFLOW's input data sets, it also should be obeyed if one or both character strings were originally SAS variable names, or were obtained as the values of options or statements parsed to PROC NETFLOW. For example, if the network has only one node that has supply capability, or if you are solving a MAXFLOW or SHORTPATH problem, you can indicate that node using the SOURCE= option.

If you specify

```
proc netflow source=NotableNode
```

then PROC NETFLOW looks for a value of the TAILNODE list variable that is `NotableNode`.

Version 6 of the SAS System converts text that makes up statements into uppercase. The name of the node searched for would be NOTABLENODE, even if this was your SAS code:

```
proc netflow source=NotableNode
```

If you want PROC NETFLOW to behave as it did in Version 6, specify

```
options validvarname=v6;
```

If the SPARSECONDATA option is not specified, and you are running SAS software Version 6 or have specified options validvarname=v6; using a later version, all NAME list variable values in the ARCDATA= data set are uppercased. This is because the SAS System has uppercased all SAS variable names, particularly those in the VAR list of the CONDATA= data set.

Entities that contain blanks must be enclosed in single or double quotes.

See the section "Cautions" on page 349 for additional discussion of case sensitivity.

Loop Arcs

When using the primal simplex network algorithm, loop arcs (arcs directed toward nodes from which they originate) are prohibited. Rather, introduce a dummy intermediate node in loop arcs. For example, replace arc (A,A) with (A,B) and (B,A). B is the name of a new node, and it must be distinct for each loop arc.

Multiple Arcs

Multiple arcs with the same tail and head nodes are prohibited. PROC NETFLOW checks to ensure there are no such arcs before proceeding with the optimization. Introduce a new dummy intermediate node in multiple arcs. This node must be distinct for each multiple arc. For example, if some network has three arcs directed from node A toward node B, then replace one of these three with arcs (A,C) and (C,B) and replace another one with (A,D) and (D,B). C and D are new nodes added to the network.

Pricing Strategies

The pricing strategy is the part of the simplex iteration that selects the nonbasic arc, constraint slack, surplus, or nonarc variable that should have a flow or value change, and perhaps enter the basis so that the total cost incurred is improved.

The pricing mechanism takes a large amount of computational effort, so it is important to use the appropriate pricing strategy for the problem under study. As in other large scale mathematical programming software, network codes can spend more than half of their execution time performing simplex iterations in the pricing step. Some compromise must be made between using a fast strategy and improving the quality of the flow or value change candidate selection, although more simplex iterations may need to be executed.

The configuration of the problem to be optimized has a great effect on the choice of strategy. If a problem is to be run repeatedly, experimentation on that problem to determine which scheme is best may prove worthwhile. The best pricing strategy to use when there is a large amount of work to do (for example, when a cold start is used) may not be appropriate when there is little work required to reach the optimum (such as when a warm start is used). If paging is necessary, then a pricing strategy that reduces the number of simplex iterations performed might have the advantage. The proportion of time spent doing the pricing step during stage 1 optimization is usually less than the same proportion when doing stage 2 optimization. Therefore, it is more important to choose a stage 2 pricing strategy that causes fewer, but not necessarily the fewest, iterations to be executed.

There are many similarities between the pricing strategies for optimizing an unconstrained problem (or when constraints are temporarily ignored) and the pricing mechanisms for optimizing considering constraints. To prevent repetition, options have a suffix or embedded *x*. Replace *x* with 1 for optimization without constraint consideration and 2 for optimization with constraint consideration.

There are three main types of pricing strategies:

- PRICETYPE*x*=NOQ
- PRICETYPE*x*=BLAND
- PRICETYPE*x*=Q

The pricing strategy that usually performs better than the others is PRICETYPE*x*=Q. For this reason, PRICETYPE*x*=Q is the default.

PRICETYPE*x*=NOQ

PRICETYPE*x*=NOQ is the least complex pricing strategy, but it is nevertheless quite efficient. In contrast to the specification of PRICETYPE*x*=Q, a candidate queue is not set up.

The P*x*SCAN= option controls the amount of additional candidate selection work done to find a better candidate after an eligible candidate has been found.

If P*x*SCAN=FIRST is specified, the search for candidates finishes when the first eligible candidate is found, with this exception: if a node has more than one eligible arc directed toward it, the best such arc is chosen.

If P*x*SCAN=BEST is specified, everything that is nonbasic is examined, and the best candidate of all is chosen.

If P*x*SCAN=PARTIAL is specified, once an eligible candidate is found, the scan continues for another P*x*NPARTIAL= cycles in the hope that during the additional scan, a better candidate is found. Examining all nonbasic arcs directed toward a single node is counted as only one cycle.

If P*x*SCAN=FIRST or P*x*SCAN=PARTIAL is specified, the scan for entering candidates starts where the last iteration's search left off. For example, if the last iteration's scan terminated after examining arcs that are directed toward the node with internal number i, the next iteration's scan starts by examining arcs directed toward the node with internal number $i + 1$. If i is the largest node number, next iterations scan begins by scanning arcs directed toward node 1 (during stage 1) or scanning constraint slack or surplus variables, if any, or nonarc variables, if any, (during stage 2). During stage 2, if the scan terminated after examining the slack or surplus of constraint i, next iterations scan starts by examining the slack or surplus of the constraint with the internal number greater than i that has such a logical variable. If the scan terminated after examining the nonarc variable i, the next iterations scan starts by examining the nonarc variable with internal number $i + 1$, (or arcs directed to the node with the smallest internal number if the nonarc variable with the greatest number has been examined). This is termed a *wraparound search*.

PRICETYPE*x*=Q

If PRICETYPE*x*=Q, a queue is set up. Candidates currently on the queue are tested at each iteration and either enter the basis or are removed from the queue. The size of the queue can be specified by using the QSIZE*x*= option. The default value for QSIZE1= is

```
QSIZE1=number of arcs/200
if (QSIZE1<24) QSIZE1=24
else if (QSIZE1>100) QSIZE1=100
```

The default value for QSIZE2= is

```
QSIZE2=(number of arcs+number of nonarc variables)/200
if (QSIZE2<24) QSIZE2=24
else if (QSIZE2>100) QSIZE2=100
```

controls the amount of additional candidate selection work done to find a better candidate after an eligible candidate has been found *in the queue*.

If you specify P*x*SCAN=BEST, the best eligible candidate found is removed from the queue. It can sustain a flow or value change and possibly enter the basis.

If you specify P*x*SCAN=FIRST, the first eligible candidate found is removed from the queue, and possibly sustains a flow or value change and enters the basis.

If you specify P*x*SCAN=PARTIAL, P*x*NPARTIAL= can then be also specified. After an eligible candidate has been found, P*x*NPARTIAL= more queue members are examined and the best of the eligible candidates found is chosen.

When P*x*SCAN=FIRST or P*x*SCAN=PARTIAL, the scan of the queue is wraparound. When the member last added to the queue has been examined, the scan continues from the member that was first added to the queue.

When the queue is empty, or after QSIZE*x*= times REFRESHQ*x*= iterations have been executed since the queue was last refreshed, new candidates are found and put onto the queue. Valid values for the REFRESHQ*x*= options are greater than 0.0 and less than or equal to 1.0. The default for REFRESHQ*x* is 0.75. If the scan cannot find enough candidates to fill the queue, the procedure reduces the value of QSIZE*x*=. If $qfound$ is the number of candidates found, the new QSIZE*x*= value is $qfound + ((old\ QSIZEx - qfound) \times REDUCEQSIZEx)$. Valid values of the REDUCEQSIZE*x*= option are between 0.0 and 1.0, inclusive. The default for REDUCEQSIZE*x*= is 1.0.

The Q*x*FILLSCAN= option controls the amount of additional candidate selection work performed to find better candidates to put into the queue after the queue has been filled.

If you specify Q*x*FILLSCAN=FIRST, the nonbasic arcs, and during stage 2 optimization, nonbasic constraint slack and surplus variables, and nonbasic nonarc variables are scanned; the scan stops when the queue is filled. If a node has more than one eligible arc directed toward it, the best such arc is put onto the queue. Q*x*FILLSCAN=FIRST is the default.

If Q*x*FILLSCAN=BEST is specified, everything that is nonbasic is scanned and the best eligible candidates are used to fill the queue.

If Q*x*FILLSCAN=PARTIAL is specified, after the queue is full, the scan continues for another Q*x*FILLNPARTIAL= cycles in the hope that during the additional scan, better candidates are found to replace other candidates previously put onto the queue. Q*x*FILLNPARTIAL=10 is the default. If Q*x*FILLSCAN=FIRST or Q*x*FILLSCAN=PARTIAL, the scan starts where the previous iteration ended; that is, it is wraparound.

In the following section, dual variables and reduced costs are explained. These help PROC NETFLOW determine whether an arc, constraint slack, surplus, or nonarc variable should have a flow or value change. P2SCAN=ANY and the DUALFREQ= option can be specified to control stage 2 pricing, and how often dual variables and reduced costs are calculated.

What usually happens when PRICETYPE2=Q is specified is that before the first iteration, the queue is filled with nonbasic variables that are eligible to enter the basis. At the start of each iteration, a candidate on the queue is examined and its reduced cost is calculated to ensure that it is still eligible to enter the basis. If it is ineligible to enter the basis, it is removed from the queue and another candidate on the queue is examined, until a candidate on the queue is found that can enter the basis. When this happens, a *minor* iteration occurs. If there are no candidates left on the queue, or several iterations have been performed since the queue was refreshed, new nonbasic variables that are eligible to enter the basis are found and are placed on the queue. When this occurs, the iteration is termed a *major* iteration. Dual variables are calculated or maintained every iteration.

During most optimizations, if a variable is put onto the queue during a major iteration, it usually remains eligible to enter the basis in later minor iterations. Specifying P2SCAN=ANY indicates that PROC NETFLOW should choose *any* candidate on the queue and use that as the entering variable. Reduced costs are not

calculated. It is simply hoped that the chosen candidate is eligible. Sometimes, a candidate on the queue is chosen that has become ineligible and the optimization takes "a step backward" rather than "a step forward" toward the optimum. However, the disadvantages of incurring an occasional step backwards and the possible danger of never converging to the optimum are offset by not having to calculate reduced costs and, more importantly, not having to maintain dual variable values. The calculation of dual variables is one of two large linear equation systems that must be solved each iteration in the simplex iteration.

If P2SCAN=ANY is specified, dual variables are calculated after DUALFREQ= iterations have been performed since they were last calculated. These are used to calculate the reduced costs of all the candidates currently on the queue. Any candidate found to be ineligible to enter the basis is removed from the queue. DUALFREQ=4 is the default.

Once again, the practice of not maintaining correct dual variable values is dangerous because backward steps are allowed, so the optimization is not guaranteed to converge to the optimum. However, if PROC NETFLOW does not run forever, it can find the optimum much more quickly than when the P2SCAN= option is not ANY. Before concluding that any solution is optimal, PROC NETFLOW calculates true dual variable values and reduced costs and uses these to verify that the optimum is really at hand.

Whether P2SCAN=ANY is specified or not, dual variables are always calculated at the start of major iterations.

PRICETYPE*x*=BLAND

PRICETYPE*x*=BLAND is equivalent to specifying in the PROC NETFLOW or RESET statement all three options PRICETYPE*x*=NOQ, P*x*SCAN=FIRST, and LRATIO*x*, and the scans are not wraparound. Bland (1977) proved that this pivot rule prevents the simplex algorithm from cycling. However, because the pivots concentrate on the lower indexed arcs, constraint slack, surplus, and nonarc variables, optimization with PRICETYPE*x*=BLAND can make the optimization execute slowly.

Dual Variables, Reduced Costs, and Status

During optimization, dual variables and reduced costs are used to determine whether an arc, constraint slack, surplus, or nonarc variable should have a flow or value change. The ARCOUT= and CONOUT= data sets each have a variable called _RCOST_ that contains reduced cost values. In the CONOUT= data set, this variable also has the reduced costs of nonarc variables. For an arc, the reduced cost is the amount that would be added to the total cost if that arc were made to convey one more unit of flow. For a nonarc variable, the reduced cost is the amount that would be added to the total cost if the value currently assigned to that nonarc variable were increased by one.

During the optimization of a minimization problem, if an arc has a positive reduced cost, PROC NETFLOW takes steps to decrease the flow through it. If an arc has a negative reduced cost, PROC NETFLOW takes steps to increase the flow through it. At optimality, the reduced costs of arcs with flow at their respective lower bounds are nonnegative; otherwise, the optimizer would have tried to increase the flow, thereby decreasing the total cost. The _STATUS_ of each such nonbasic arc is LOWERBD NONBASIC. The reduced costs of arcs with flow at capacity are nonpositive. The _STATUS_ of each such nonbasic arc is UPPERBD NONBASIC. Even though it would decrease total cost, the optimizer cannot increase the flows through such arcs because of the capacity bound. Similar arguments apply for nonarc variables.

The reduced cost is also the amount that would be subtracted from the total cost if that arc was made to convey one less unit of flow. Similarly, a reduced cost is the amount subtracted from the total cost if the value currently assigned to that nonarc variable is decreased by one.

The dual variables and reduced costs can be used to detect whether multiple optimal solutions exist. A zero reduced cost of a nonbasic arc indicates the existence of multiple optimal solutions. A zero reduced cost indicates, by definition, that the flow through such arcs can be changed with zero change to the total cost. (Basic arcs and basic nonarc variables technically have zero reduced costs. A missing value is used for these so that reduced costs of nonbasic arcs and nonbasic nonarc variables that are zero are highlighted.)

The range over which costs can vary before the present solution becomes nonoptimal can be determined through examination of the reduced costs. For any nonbasic arc with assigned flow equal to its lower bound, the amount by which the cost must be decreased before it becomes profitable for this arc to convey additional flow is the value of its reduced cost. The cost reduction necessary for a nonbasic arc currently assigned capacity flow to undergo a worthwhile flow decrease is the absolute value of its reduced cost. In both cases, this minimum cost reduction changes the reduced cost to zero. Any further reduction promotes a possible basis change.

The reduced cost of an arc (t, h) is $rc_{t,h} = c_{t,h} - \pi_t + \pi_h$ where π_i is the dual value for node i and $c_{t,h}$ is the cost of the arc with tail node t and head node h.

If the problem has side constraints and arc (t, h) has nonzero lhs coefficients, then the following term must be subtracted from $rc_{t,h}$:

$$\sum_i condual_i \, H_{i,(t,h)}$$

where $condual_i$ is the dual variable of constraint i, and $H_{i,(t,h)}$ is the coefficient of arc (t, h) in constraint i.

If d_n is the objective function coefficient of nonarc variable n, the reduced cost is $rc_n = d_n - \sum_i condual_i \, Q_{i,n}$, where $Q_{i,n}$ is the coefficient of nonarc variable n in constraint i.

The Working Basis Matrix

Let **T** be the basis matrix of NPSC. The following partitioning is done:

$$\mathbf{T}=\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

where

- n is the number of nodes.
- k is the number of side constraints.
- **A** $(n \times n)$ is the network component of the basis. Most of the columns of this matrix are columns of the problem's node-arc incidence matrix. The arcs associated with columns of **A**, called key basic variables or key arcs, form a spanning tree. The data structures of the spanning tree of this submatrix of the basis **T** enable the computations involving **T** and the manner in which **T** is updated to be very efficient, especially those dealing with **A** (or $\mathbf{A}^{-1}$).

- $\mathbf{C}$ $(k \times n)$ are the key arcs' side constraint coefficient columns.
- $\mathbf{B}$ $(n \times k)$ are the node-arc incidence matrix columns of the nontree arcs. The columns of $\mathbf{B}$ having nonzero elements are associated with basic nonspanning tree arcs.
- $\mathbf{D}$ $(k \times k)$ are the constraint coefficient columns of nonkey basic variables. Nonkey basic variables not only include nontree basic arcs but also basic slack, surplus, artificial, or nonarc variables.

It is more convenient to factor $\mathbf{T}$ by block triangular matrices $\mathbf{P}$ and $\mathbf{M}$, such that $\mathbf{P} = \mathbf{TM}$. The matrices $\mathbf{P}$ and $\mathbf{M}$ are used instead of $\mathbf{T}$ because they are less burdensome to work with. You can perform block substitution when solving the simplex iteration linear systems of equations

$$\mathbf{P}=\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & D_w \end{bmatrix}$$

$$\mathbf{M}=\begin{bmatrix} \mathbf{I} & -A^{-1}B \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

where $\mathbf{D_w} = \mathbf{D} - \mathbf{CA^{-1}B}$ and is called the working basis matrix.

To perform block substitution, you need the tree data structure of the $\mathbf{A}$ matrix, and also the $\mathbf{C}$, $\mathbf{B}$, and $\mathbf{D_w}$ matrices. Because the $\mathbf{C}$ matrix consists of columns of the constraint coefficient matrix, the maintenance of $\mathbf{C}$ from iteration to iteration simply entails changing information specifying which columns of the constraint coefficient matrix compose $\mathbf{C}$.

The $\mathbf{A^{-1}B}$ matrix is usually very sparse. Fortunately, the information in $\mathbf{A^{-1}B}$ can be initialized easily using the tree structures. In most iterations, only one column is replaced by a new one. The values of the elements of the new column may already be known from preceding steps of the simplex iteration.

The working basis matrix is the submatrix that presents the most computational complexity. However, PROC NETFLOW usually can use classical simplex pivot techniques. In many iterations, only one column of $\mathbf{D_w}$ changes. Sometimes it is not necessary to update $\mathbf{D_w}$ or its inverse at all.

If INVD_2D is specified in the PROC NETFLOW statement, only one row and one column may need to be changed in the $\mathbf{D_w^{-1}}$ before the next simplex iteration can begin. The new contents of the changed column are already known. The new elements of the row that changes are influenced by the contents of a row of $\mathbf{A^{-1}B}$ that is very sparse.

If INVD_2D is not specified in the PROC NETFLOW statement, the Bartels-Golub update can be used to update the $\mathbf{LU}$ factors of $\mathbf{D_w}$. The choice must be made whether to perform a series of updates (how many depends on the number of nonzeros in a row of $\mathbf{A^{-1}B}$), or refactorization.

Flow and Value Bounds

The capacity and lower flow bound of an arc can be equal. Negative arc capacities and lower flow bounds are permitted. If both arc capacities and lower flow bounds are negative, the lower flow bound must be at least as negative as the capacity. An arc (A,B) that has a negative flow of $-f$ units can be interpreted as an arc that conveys f units of flow from node B to node A.

The upper and lower value bounds of a nonarc variable can be equal. Negative upper and lower bounds are permitted. If both are negative, the lower bound must be at least as negative as the upper bound.

Tightening Bounds and Side Constraints

If any piece of data is furnished to PROC NETFLOW more than once, PROC NETFLOW checks for consistency so that no conflict exists concerning the data values. For example, if the cost of some arc is seen to be one value and as more data are read, the cost of the same arc is seen to be another value, PROC NETFLOW issues an error message on the SAS log and stops. There are two exceptions:

- The bounds of arcs and nonarc variables are made as tight as possible. If several different values are given for the lower flow bound of an arc, the greatest value is used. If several different values are given for the lower bound of a nonarc variable, the greatest value is used. If several different values are given for the capacity of an arc, the smallest value is used. If several different values are given for the upper bound of a nonarc variable, the smallest value is used.
- Several values can be given for inequality constraint right-hand sides. For a particular constraint, the lowest rhs value is used for the rhs if the constraint is of *less than or equal to* type. For a particular constraint, the greatest rhs value is used for the rhs if the constraint is of *greater than or equal to* type.

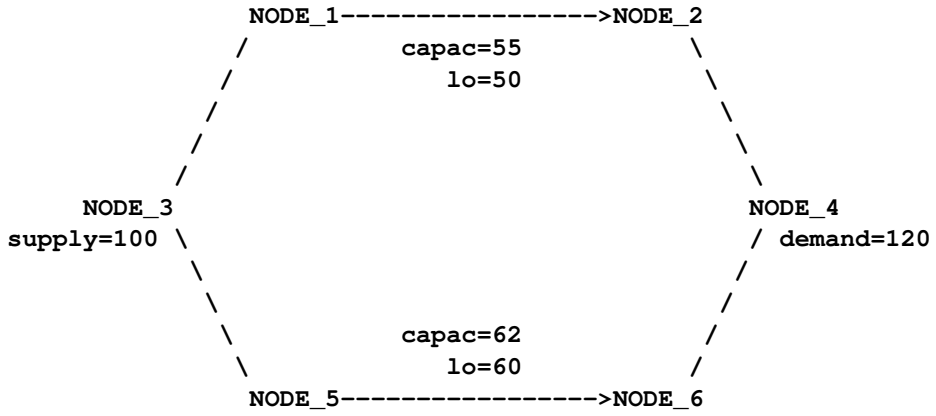
Reasons for Infeasibility

Before optimization commences, PROC NETFLOW tests to ensure that the problem is not infeasible by ensuring that, with respect to supplies, demands, and arc flow bounds, flow conservation can be obeyed at each node.

- Let *IN* be the sum of lower flow bounds of arcs directed toward a node plus the node's supply. Let *OUT* be the sum of capacities of arcs directed from that node plus the node's demand. If *IN* exceeds *OUT*, not enough flow can leave the node.
- Let *OUT* be the sum of lower flow bounds of arcs directed from a node plus the node's demand. Let *IN* be the total capacity of arcs directed toward the node plus the node's supply. If *OUT* exceeds *IN*, not enough flow can arrive at the node.

Reasons why a network problem can be infeasible are similar to those previously mentioned but apply to a set of nodes rather than for an individual node. Consider the network illustrated in Figure 6.13.

Figure 6.13 An Infeasible Network



The demand of NODE_4 is 120. That can never be satisfied because the maximal flow through arcs (NODE_1, NODE_2) and (NODE_5, NODE_6) is 117. More specifically, the implicit supply of NODE_2 and NODE_6 is only 117, which is insufficient to satisfy the demand of other nodes (real or implicit) in the network.

Furthermore, the lower flow bounds of arcs (NODE_1, NODE_2) and (NODE_5, NODE_6) are greater than the flow that can reach the tail nodes of these arcs, that, by coincidence, is the total supply of the network. The implicit demand of nodes NODE_1 and NODE_5 is 110, which is greater than the amount of flow that can reach these nodes.

When PROC NETFLOW detects that the problem is infeasible, it indicates why the solution, obtained after optimization stopped, is infeasible. It can report that the solution cannot obey flow conservation constraints and which nodes these conservation constraints are associated with. If applicable, the side constraints that the solution violates are also output.

If stage 1 optimization obtains a feasible solution to the network, stage 2 optimization can determine that the problem is infeasible and note that some flow conservation constraint is broken while all side constraints are satisfied. The infeasibility messages issued by PROC NETFLOW pertain to why the *current* solution is infeasible, not quite the same as the reasons why the *problem* is infeasible. However, the messages highlight *areas* in the problem where the infeasibility can be tracked down. If the problem is infeasible, make PROC NETFLOW do a stage 1 unconstrained optimization by removing the CONDATA= data set specification in the PROC NETFLOW statement. If a feasible network solution is found, then the side constraints are the source of the infeasibility in the problem.

Missing S Supply and Missing D Demand Values

In some models, you may want a node to be either a supply or demand node but you want the node to supply or demand the optimal number of flow units. To indicate that a node is such a supply node, use a missing S value in the SUPPLY list variable in the ARCDATA= data set or the SUPDEM list variable in the NODEDATA= data set. To indicate that a node is such a demand node, use a missing D value in the DEMAND list variable in the ARCDATA= data set or the SUPDEM list variable in the NODEDATA= data set.

Suppose the oil example in the section "Introductory Example" on page 315 is changed so that crude oil can be obtained from either the Middle East or U.S.A. in any amounts. You should specify that the node "middle east" is a supply node, but you do not want to stipulate that it supplies 100 units, as before. The node "u.s.a." should also remain a supply node, but you do not want to stipulate that it supplies 80 units. You must specify that these nodes have missing S supply capabilities.

```
title  'Oil Industry Example';
title3 'Crude Oil can come from anywhere';
data miss_s;
  missing S;
  input   _node_&$15. _sd_;
  datalines;
middle east          S
u.s.a.               S
servstn1 gas       -95
servstn1 diesel    -30
servstn2 gas       -40
servstn2 diesel    -15
;
```

The following PROC NETFLOW run uses the same ARCDATA= and CONDATA= data sets used in the section "Introductory Example" on page 315.

```
proc netflow
  nodedata=miss_s       /* the supply (missing S) and */
                        /* demand data                */
  arcdata=arcd1         /* the arc descriptions       */
  condata=cond1         /* the side constraints       */
  conout=solution;      /* the solution data set      */
  run;
print some_arcs('middle east' 'u.s.a.',_all_)/short;

proc print;
  sum _fcost_;
  run;
```

The following messages appear on the SAS log:

```
NOTE: Number of nodes= 14 .
NOTE: All supply nodes have unspecified (.S) supply capability. Number of these
      nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 0 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of iterations performed (neglecting any constraints)= 15 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 50040 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 8 .
NOTE: Number of iterations, optimizing with constraints= 3 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= 50075 .
NOTE: The data set WORK.SOLUTION has 18 observations and 14 variables.
```

The PRINT statement reports the arcs directed away from the supply nodes, shown in Figure 6.14. The amount of crude obtained from the Middle East and U.S.A. is 30 and 150 units, respectively.

Figure 6.14 Print Statement, Oil Example, Missing S Supplies

Oil Industry Example

Crude Oil can come from anywhere

The NETFLOW Procedure

_N_	_from_	_to_	_cost_	_capac_	_lo_	_name_	_FLOW_
1	middle east	refinery 1	63	95	20	m_e_ref1	20
2	u.s.a.	refinery 1	55	99999999	0		125
3	middle east	refinery 2	81	80	10	m_e_ref2	10
4	u.s.a.	refinery 2	49	99999999	0		25

The CONOUT= data set is shown in Figure 6.15.

Figure 6.15 Missing S SUPDEM Values in NODEDATA

Obs	_from_	_to_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_
1	refinery 1	r1	200	175	50	thruput1	.	.
2	refinery 2	r2	220	100	35	thruput2	.	.
3	r1	ref1 diesel	0	75	0		.	.
4	r1	ref1 gas	0	140	0	r1_gas	.	.
5	r2	ref2 diesel	0	75	0		.	.
6	r2	ref2 gas	0	100	0	r2_gas	.	.
7	middle east	refinery 1	63	95	20	m_e_ref1	S	.
8	u.s.a.	refinery 1	55	99999999	0		S	.
9	middle east	refinery 2	81	80	10	m_e_ref2	S	.
10	u.s.a.	refinery 2	49	99999999	0		S	.
11	ref1 diesel	servstn1 diesel	18	99999999	0		.	30
12	ref2 diesel	servstn1 diesel	36	99999999	0		.	30
13	ref1 gas	servstn1 gas	15	70	0		.	95
14	ref2 gas	servstn1 gas	17	35	5		.	95
15	ref1 diesel	servstn2 diesel	17	99999999	0		.	15
16	ref2 diesel	servstn2 diesel	23	99999999	0		.	15
17	ref1 gas	servstn2 gas	22	60	0		.	40
18	ref2 gas	servstn2 gas	31	99999999	0		.	40

Obs	_FLOW_	_FCOST_	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	
1	145.00	29000.00	.	7	2	KEY_ARC	BASIC
2	35.00	7700.00	17	8	3	LOWERBD	NONBASIC
3	36.25	0.00	.	10	5	KEY_ARC	BASIC
4	108.75	0.00	.	9	5	KEY_ARC	BASIC
5	8.75	0.00	.	12	6	KEY_ARC	BASIC
6	26.25	0.00	.	11	6	KEY_ARC	BASIC
7	20.00	1260.00	8	2	1	LOWERBD	NONBASIC
8	125.00	6875.00	.	3	4	KEY_ARC	BASIC
9	10.00	810.00	32	4	1	LOWERBD	NONBASIC
10	25.00	1225.00	.	5	4	KEY_ARC	BASIC
11	30.00	540.00	.	17	8	KEY_ARC	BASIC
12	0.00	0.00	12	18	10	LOWERBD	NONBASIC
13	68.75	1031.25	.	13	7	KEY_ARC	BASIC
14	26.25	446.25	.	14	9	NONKEY ARC	BASIC
15	6.25	106.25	.	19	8	KEY_ARC	BASIC
16	8.75	201.25	.	20	10	KEY_ARC	BASIC
17	40.00	880.00	.	15	7	KEY_ARC	BASIC
18	0.00	0.00	7	16	9	LOWERBD	NONBASIC
		50075.00					

The optimal supplies of nodes “middle east” and “u.s.a.” are 30 and 150 units, respectively. For this example, the same optimal solution is obtained if these nodes had supplies less than these values (each supplies 1 unit, for example) and the THRUNET option was specified in the PROC NETFLOW statement. With the THRUNET option active, when total supply exceeds total demand, the specified nonmissing demand values

are the lowest number of flow units that must be absorbed by the corresponding node. This is demonstrated in the following PROC NETFLOW run. The missing S is most useful when nodes are to supply optimal numbers of flow units and it turns out that for some nodes, the optimal supply is 0.

```
data miss_s_x;
   missing S;
   input   _node_&$15. _sd_;
   datalines;
middle east         1
u.s.a.              1
servstn1 gas      -95
servstn1 diesel   -30
servstn2 gas      -40
servstn2 diesel   -15
;

proc netflow
   thrunet
   nodedata=miss_s_x     /* No supply (missing S)      */
   arcdata=arcd1         /* the arc descriptions       */
   condata=cond1         /* the side constraints       */
   conout=solution;      /* the solution data set      */
   run;
print some_arcs('middle east' 'u.s.a.',_all_)/short;

proc print;
   sum _fcost_;
   run;
```

The following messages appear on the SAS log. Note that the Total supply= 2, not 0 as in the last run.

```
NOTE: Number of nodes= 14 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 2 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of iterations performed (neglecting any constraints)= 20 .
NOTE: Of these, 1 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 50040 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 8 .
NOTE: Number of iterations, optimizing with constraints= 3 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= 50075 .
NOTE: The data set WORK.SOLUTION has 18 observations and 14 variables.
```

The PRINT statement and the CONDATA= data set are very similar; the supplies of the supply nodes are 1, not missing S. Otherwise, the solutions are identical.

If total supply exceeds total demand, any missing S values are ignored. If total demand exceeds total supply, any missing D values are ignored.

Balancing Total Supply and Total Demand

When Total Supply Exceeds Total Demand

When total supply of a network problem exceeds total demand, PROC NETFLOW can add an extra node (called the *excess node*) to the problem and set the demand at that node equal to the difference between total supply and total demand. There are three ways that this excess node can be joined to the network. All three ways entail PROC NETFLOW generating a set of arcs (henceforth referred to as the *generated* arcs) that are directed toward the excess node. The total amount of flow in generated arcs equals the demand of the excess node. The generated arcs originate from one of three sets of nodes.

When you specify the THRUNET option, the set of nodes that generated arcs originate from are all demand nodes, even those demand nodes with unspecified demand capability. You indicate that a node has unspecified demand capability by using a missing D value instead of an actual value for demand data (discussed in the section "Missing S Supply and Missing D Demand Values" on page 401). The value specified as the demand of a demand node is in effect a lower bound of the number of flow units that node can actually demand. For missing D demand nodes, this lower bound is zero.

If you do not specify the THRUNET option, the way in which the excess node is joined to the network depends on whether there are demand nodes with unspecified demand capability (nodes with missing D demand).

If there are missing D demand nodes, these nodes are the set of nodes that generated arcs originate from. The value specified as the demand of a demand node, if not missing D, is the number of flow units that node actually demands. For a missing D demand node, the actual demand of that node may be zero or greater.

If there are no missing D demand nodes, the set of nodes that generated arcs originate from are the set of supply nodes. The value specified as the supply of a supply node is in effect an upper bound of the number of flow units that node can actually supply. For missing S supply nodes (discussed in the section "Missing S Supply and Missing D Demand Values" on page 401), this upper bound is zero, so missing S nodes when total supply exceeds total demand are transshipment nodes, nodes that neither supply nor demand flow.

When Total Supply Is Less Than Total Demand

When total supply of a network problem is less than total demand, PROC NETFLOW can add an extra node (called the *excess node*) to the problem and set the supply at that node equal to the difference between total demand and total supply. There are three ways that this excess node can be joined to the network. All three ways entail PROC NETFLOW generating a set of arcs (henceforth referred to as the *generated* arcs) that originate from the excess node. The total amount of flow in generated arcs equals the supply of the excess node. The generated arcs are directed toward one of three sets of nodes.

When you specify the THRUNET option, the set of nodes that generated arcs are directed toward are all supply nodes, even those supply nodes with unspecified supply capability. You indicate that a node has unspecified supply capability by using a missing S value instead of an actual value for supply data (discussed in the section "Missing S Supply and Missing D Demand Values" on page 401). The value specified as the supply of a supply node is in effect a lower bound of the number of flow units that node can actually supply. For missing S supply nodes, this lower bound is zero.

If you do not specify the THRUNET option, the way in which the excess node is joined to the network depends on whether there are supply nodes with unspecified supply capability (nodes with missing S supply).

If there are missing S supply nodes, these nodes are the set of nodes that generated arcs are directed toward. The value specified as the supply of a supply node, if not missing S, is the number of flow units that node actually supplies. For a missing S supply node, the actual supply of that node may be zero or greater.

If there are no missing S supply nodes, the set of nodes that generated arcs are directed toward are the set of demand nodes. The value specified as the demand of a demand node is in effect an upper bound of the number of flow units that node can actually demand. For missing D demand nodes, (discussed in the section "Missing S Supply and Missing D Demand Values" on page 401), this upper bound is zero, so missing D nodes when total supply is less than total demand are transshipment nodes, nodes that neither supply nor demand flow.

Warm Starts

Using a warm start can increase the overall speed of PROC NETFLOW when it is used repetitively on problems with similar structure. It is most beneficial when a solution of a previous optimization is close to the optimum of the same network with some of its parameters, for example, arc costs, changed. Whether a problem is changed or not, a nonoptimal solution resulting from a previous optimization can be used to restart optimization, thereby saving PROC NETFLOW from having to repeat work to reach the warm start already available.

Time also is saved in the data structure initialization part of the NETFLOW procedure's execution. Information about the previous optimal solution, particularly concerning the size of the problem, a description of the basis spanning tree structure, and what is basic in constraint rows, is known. Information about which nonbasic arcs have capacity flow and which nonbasic nonarc variables are at their respective upper bounds also makes up part of the warm start. The procedure can place arc data into the internal arc length arrays in precisely defined locations, in order of ascending head node internal number. It is not necessary to have multiple passes through the data because literals such as node, nonarc variable, arc, constraint, and special row names are defined and meaning is attached to each. This also saves a considerable amount of memory. None of the pre-optimization feasibility checks need be repeated.

Warm starts also are useful if you want to determine the effect of arcs being closed to carrying flow. The costs of these arcs are set high enough to ensure that the next optimal solution never has flow through them. Similarly, the effect of opening arcs can be determined by changing the cost of such arcs from an extreme to a reasonable value.

Specify the FUTURE1 or FUTURE2 option to ensure that additional data about a solution to be used as a warm start are output to output data sets. If the FUTURE1 option is specified, extra observations with information on what is to be the warm start are set up for the NODEOUT= and ARCOUT= data sets. The warm start solution in these data sets is a solution obtained after optimization neglecting side constraints. Any cost list variable value in the ARCOUT= data set (and, if there are side constraints, any constraint data in the CONDATA= data set) can be changed before the solution is used as a warm start in a subsequent PROC NETFLOW run. Any nonarc variable data in the CONDATA= data set can be changed at this time as well. New nonarc variables not present in the original problem when the warm start was generated can also be added to the CONDATA= data set before the problem is warm started.

If the FUTURE2 option is specified, extra variables containing information on what will be the warm start solution are set up for the DUALOUT= and CONOUT= data sets. The warm start solution in these data sets is obtained after optimization that considers side constraints has been performed. Part of the warm start is concerned with the constraint part of the basis. Only cost list variable values in the CONOUT= data set can be changed before the solution is used as a warm start in a subsequent PROC NETFLOW run.

If a primal simplex optimization is to use a warm start, the WARM option must be specified in the PROC NETFLOW statement. Otherwise, the primal simplex network algorithm processes the data for a cold start and the extra information is not used.

The ARCDATA= data set is either the ARCOUT= data set from a previous run of PROC NETFLOW with the FUTURE1 option specified (if an unconstrained warm start is used) or the CONOUT= data set from a previous run of PROC NETFLOW with the FUTURE2 option specified (if the warm start was obtained after optimization that considers side constraints was used).

The NODEDATA= data set is the NODEOUT= data set from a previous run of PROC NETFLOW with FUTURE1 specified if an unconstrained warm start is being used. Otherwise, the DUALIN= is the DUALOUT= data sets from a previous run of PROC NETFLOW with FUTURE2 specified, if the warm start was obtained after optimization that considers side constraints was used.

You never need to alter the NODEOUT= data set or the DUALOUT= data set between the time they are generated and when they are used as a warm start. The results would be unpredictable if incorrect changes were made to these data sets, or if a NODEDATA= or a DUALIN= data set were used with an ARCDATA= data set of a different solution.

It is possible, and often useful, to specify WARM and either FUTURE1 or FUTURE2, or both, in the same PROC NETFLOW statement if a new warm start is to be generated from the present warm start.

The extent of the changes allowed to a primal simplex warm start between the time it is generated and when it is used depends on whether the warm start describes an unconstrained or constrained solution. The following list describes parts of a constrained or an unconstrained warm start that can be altered:

- COST list variable values
- the value of an arc's capacity, as long as the new capacity value is not less than the lower flow bound or the flow through the arc
- any nonarc variable information, in an unconstrained warm start
- for an unconstrained warm start, any side constraint data

The changes that can be made in constraint data for a constrained warm start are more restrictive than those for an unconstrained warm start. The lhs coefficients, type, and rhs value of a constraint can be changed as long as that constraint's slack, surplus, or artificial variable is basic. The constraint name cannot be changed.

Example of a Warm Start

The following sample SAS session demonstrates how the warm start facilities are used to obtain optimal solutions to an unconstrained network where some arc cost changes occur or optimization is halted before the optimum is found.

```
                /* data already in data sets node0 and arc0 */
proc netflow
   nodedata=node0  /* if supply_demand information  */
                   /* is in this SAS data set       */
   arcdata=arc0;
               /* variable list specifications go here    */
               /* assume that they are not necessary here */
               /* if they are, they must be included in   */
               /* all the PROC NETFLOW calls that follow  */
   reset
      future1
      nodeout=node2  /* nodeout and arcout are necessary  */
                     /* when FUTURE1 is used              */
      arcout=arc1;
proc print
      data=arc1;     /* display the optimal solution      */
proc fsedit
      data=arc1;     /* change some arc costs             */
data arc2;
   reset arc1;
      oldflow=_flow_;
      oldfc=_fcost_;
               /* make duplicates of the flow and flowcost*/
               /* variables. If a id list was explicitly  */
               /* specified, add oldflow and oldfc to this*/
               /* list so that they appear in subsequently*/
               /* created arcout= data sets               */
```

The following PROC NETFLOW uses the warm start created previously, performs 250 stage 2 iterations and saves that solution, which (as FUTURE1, ARCOUT=, and NODEOUT= are specified) can be used as a warm start in another PROC NETFLOW run.

```
proc netflow
     warm
     nodedata=node2
     arcdata=arc2;
   reset
     maxit1=250
     future1;
   run;
   save
     nodeout=savelib.node3
     arcout=savelib.arc3;
            /* optimization halted because 250 iterations */
            /* were performed to resume optimization,     */
            /* possibly in another session (the output    */
            /* data sets were saved in a SAS library      */
            /* called savelib)                            */
```

Using the latest warm start, PROC NETFLOW is re-invoked to find the optimal solution.

```
proc netflow
     warm
     nodedata=savelib.node3
     arcdata=savelib.arc3;
   reset
     future1
     nodeout=node4
     arcout=arc4;
   run;
```

If this problem has constraints with data in a data set called CON0, then in each of the previous PROC NETFLOW statements, specify CONDATA=CON0. Between PROC NETFLOW runs, you can change constraint data. In each of the RESET statements, you could specify the CONOUT= data set to save the last (possibly optimal) solution reached by the optimizer if it reaches stage 2. You could specify FUTURE2 and the DUALOUT= data set to generate a constrained warm start.

```
proc netflow
     warm
     nodedata=node4
     arcdata=arc4
     condata=con0;
   reset
     maxit2=125 /* optional, here as a reason why         */
                /* optimum will not be obtained          */
     scratch    /* optional, but warm start might be good */
                /* enough to start stage 2 optimization   */
     future2
   run;
     /* optimization halted after 125 stage 2 iterations  */
   save dualout=dual1 conout=conout1;
```

Stage 2 optimization halted before optimum was reached. Now you can make cost and nonarc variable objective function coefficient changes. Then to restart optimization, use

```
proc netflow
     warm
     condata=con0
           /* NB. NETFLOW reads constraint data only     */
     dualin=dual1
     arcdata=con1;
   reset
     future2
     dualout=dual2
     conout=con2;
   run;
```

How to Make the Data Read of PROC NETFLOW More Efficient

This section contains information useful when you want to solve large constrained network problems. However, much of this information is also useful if you have a large linear programming problem. All of the options described in this section that are not directly applicable to networks (options such as ARCS_ONLY_ARCDATA, ARC_SINGLE_OBS, NNODES=, and NARCS=) can be specified to improve the speed at which LP data are read.

Large Constrained Network Problems

Many of the models presented to PROC NETFLOW are enormous. They can be considered large by linear programming standards; problems with thousands of variables and constraints. When dealing with side constrained network programming problems, models can have not only a linear programming component of that magnitude, but also a larger, possibly *much* larger, network component.

The majority of a network problem's decision variables are arcs. Like an LP decision variable, an arc has an objective function coefficient, upper and lower value bounds, and a name. Arcs can have coefficients in constraints. Therefore, an arc is quite similar to an LP variable and places the same memory demands on optimization software as an LP variable. But a typical network model has many more arcs and nonarc variables than the typical LP model has variables. And arcs have tail and head nodes. Storing and processing node names require huge amounts of memory. To make matters worse, node names occupy memory at times when a large amount of other data should also reside in memory.

While memory requirements are lower for a model with embedded network component compared with the equivalent LP *once optimization starts*, the same is usually not true *during the data read*. Even though nodal flow conservation constraints in the LP should not be specified in the constrained network formulation, the memory requirements to read the latter are greater because each arc (unlike an LP variable) originates at one node, and is directed toward another.

Paging

PROC NETFLOW has facilities to read data when the available memory is insufficient to store all the data at once. PROC NETFLOW does this by allocating memory for different purposes, for example, to store an array or receive data read from an input SAS data set. After that memory has filled, the information is sent to disk and PROC NETFLOW can resume filling that memory with new information. Often, information must be retrieved from disk so that data previously read can be examined or checked for consistency. Sometimes, to prevent any data from being lost, or to retain any changes made to the information in memory, the contents of the memory must be sent to disk before other information can take its place. This process of swapping information to and from disk is called paging. Paging can be very time-consuming, so it is crucial to minimize the amount of paging performed.

There are several steps you can take to make PROC NETFLOW read the data of network and linear programming models more efficiently, particularly when memory is scarce and the amount of paging must be reduced. PROC NETFLOW will then be able to tackle large problems in what can be considered reasonable amounts of time.

The Order of Observations

PROC NETFLOW is quite flexible in the ways data can be supplied to it. Data can be given by any reasonable means. PROC NETFLOW has convenient defaults that can save you work when generating the data. There can be several ways to supply the same piece of data, and some pieces of data can be given more than once. PROC NETFLOW reads everything, then merges it all together. However, this flexibility and convenience come at a price; PROC NETFLOW may not assume the data has a characteristic that, if possessed by the data, could save time and memory during the data read. There are several options that indicate the data has some exploitable characteristic.

For example, an arc cost can be specified once or several times in the ARCDATA= or CONDATA= data set, or both. Every time it is given in ARCDATA, a check is made to ensure that the new value is the same as any corresponding value read in a previous observation of ARCDATA. Every time it is given in CONDATA, a check is made to ensure that the new value is the same as the value read in a previous observation of CONDATA, or previously in ARCDATA. It would save PROC NETFLOW time if it knew that arc cost data would be encountered only once while reading ARCDATA, so performing the time-consuming check for consistency would not be necessary. Also, if you indicate that CONDATA contains data for constraints only, PROC NETFLOW will not expect any arc information, so memory will not be allocated to receive such data while reading CONDATA. This memory is used for other purposes and this might lead to a reduction in paging. If applicable, use the ARC_SINGLE_OBS or the CON_SINGLE_OBS option, or both, and the NON_REPLIC=COEFS specification to improve how ARCDATA and CONDATA are read.

PROC NETFLOW allows the observations in input data sets to be in any order. However, major time savings can result if you are prepared to order observations in particular ways. Time spent by the SORT procedure to sort the input data sets, particularly the CONDATA= data set, may be more than made up for when PROC NETFLOW reads them, because PROC NETFLOW has in memory information possibly used when the previous observation was read. PROC NETFLOW can assume a piece of data is either similar to that of the last observation read or is new. In the first case, valuable information such as an arc or a nonarc variable number or a constraint number is retained from the previous observation. In the last case, checking the data with what has been read previously is not necessary.

Even if you do not sort the CONDATA= data set, grouping observations that contain data for the same arc or nonarc variable or the same row pays off. PROC NETFLOW establishes whether an observation being read is similar to the observation just read.

Practically, several input data sets for PROC NETFLOW might have this characteristic, because it is natural for data for each constraint to be grouped together (dense format of CONDATA) or data for each column to be grouped together (sparse format of CONDATA). If data for each arc or nonarc is spread over more than one observation of the ARCDATA= data set, it is natural to group these observations together.

Use the GROUPED= option to indicate whether observations of the ARCDATA= data set, CONDATA= data set, or both are grouped in a way that can be exploited during data read.

Time is saved if the type data for each row appears near the top of the CONDATA= data set, especially if it has the sparse format. Otherwise, when reading an observation, if PROC NETFLOW does not know if a row is a constraint or special row, the data are set aside. Once the data set has been completely read, PROC NETFLOW must reprocess the data it set aside. By then, it knows the type of each constraint or row or, if its type was not provided, it is assumed to have a default type.

Better Memory Utilization

In order for PROC NETFLOW to make better utilization of available memory, you can now specify options that indicate the approximate size of the model. PROC NETFLOW then knows what to expect. For example, if you indicate that the problem has no nonarc variables, PROC NETFLOW will not allocate memory to store nonarc data. That memory is utilized better for other purposes. Memory is often allocated to receive or store data of some type. If you indicate that the model does not have much data of a particular type, the memory that would otherwise have been allocated to receive or store that data can be used to receive or store data of another type.

- NNODES= approximate number of nodes
- NARCS= approximate number of arcs
- NNAS= approximate number of nonarc variables or LP variables
- NCONS= approximate number of constraints
- NCOEFS= approximate number of constraint coefficients

These options will sometimes be referred to as Nxxxx= options.

You do not need to specify all these options for the model, but the more you do, the better. If you do not specify some or all of these options, PROC NETFLOW guesses the size of the problem by using what it already knows about the model. Sometimes PROC NETFLOW guesses the size of the model by looking at the number of observations in the ARCDATA= and CONDATA= data sets. However, PROC NETFLOW uses rough rules of thumb; that typical models are proportioned in certain ways (for example, if there are constraints, then arcs and nonarcs usually have 5 constraint coefficients). If your model has an unusual shape or structure, you are encouraged to use these options.

If you do use the options and you do not know the exact values to specify, *overestimate* the values. For example, if you specify NARCS=10000 but the model has 10100 arcs, when dealing with the last 100 arcs, PROC NETFLOW might have to page out data for 10000 arcs each time one of the last arcs must be dealt with. Memory could have been allocated for all 10100 arcs without affecting (much) the rest of the data read, so NARCS=10000 could be more of a hindrance than a help.

The point of these Nxxxx= options is to indicate the model size when PROC NETFLOW does not know it. When PROC NETFLOW knows the "real" value, that value is used instead of Nxxxx= .

When PROC NETFLOW is given a constrained solution warm start, PROC NETFLOW knows from the warm start information all model size parameters, so Nxxxx= options are not used. When an unconstrained warm start is used and the SAME_NONARC_DATA is specified, PROC NETFLOW knows the number of nonarc variables, so that is used instead of the value of the NNAS= option.

ARCS_ONLY_ARCDATA indicates that data for only arcs are in the ARCDATA= data set. Memory would not be wasted to receive data for nonarc and LP variables.

Use the memory usage parameters:

- The BYTES= option specifies the size of PROC NETFLOW main working memory in number of bytes.

- The MAXARRAYBYTES= option specifies the maximum number of bytes that an array can occupy.
- The MEMREP option indicates that memory usage report is to be displayed on the SAS log.

Specifying the BYTES= parameter is particularly important. Specify as large a number as possible, but not such a large number of bytes that will cause PROC NETFLOW (rather, the SAS System running underneath PROC NETFLOW) to run out of memory. Use the MAXARRAYBYTES= option if the model is very large or "disproportionate." Try increasing or decreasing the MAXARRAYBYTES= option. Limiting the amount of memory for use by big arrays is good if they would take up too much memory to the detriment of smaller arrays, buffers, and other things that require memory. However, too small a value of the MAXARRAYBYTES= option might cause PROC NETFLOW to page a big array excessively. Never specify a value for the MAXARRAYBYTES= option that is smaller than the main node length array. PROC NETFLOW reports the size of this array on the SAS log if you specify the MEMREP option. The MAXARRAYBYTES= option influences paging not only in the data read, but also during optimization. It is often better if optimization is performed as fast as possible, even if the read is made slower as a consequence.

Use Defaults to Reduce the Amount of Data

Use as much as possible the parameters that specify default values. For example, if there are several arcs with the same cost value *c*, use DEFCOST=*c* for arcs that have that cost. Use missing values in the COST variable in ARCDATA instead of *c*. PROC NETFLOW ignores missing values, but must read, store, and process nonmissing values, even if they are equal to a default option or could have been equal to a default parameter had it been specified. Sometimes, using default parameters makes the need for some SAS variables in the ARCDATA= and CONDATA= data sets no longer necessary, or reduces the quantity of data that must be read. The default options are

- DEFCOST= default cost of arcs, objective function of nonarc variables or LP variables
- DEFMINFLOW= default lower flow bound of arcs, lower bound of nonarc variables or LP variables
- DEFCAPACITY= default capacity of arcs, upper bound of nonarc variables or LP variables
- DEFCONTYPE=LE DEFCONTYPE= <=
  DEFCONTYPE=EQ DEFCONTYPE= =
  DEFCONTYPE=GE DEFCONTYPE= >= (default constraint type)

The default options themselves have defaults. For example, you do not need to specify DEFCOST=0 in the PROC NETFLOW statement. You should still have missing values in the COST variable in ARCDATA for arcs that have zero costs.

If the network has only one supply node, one demand node, or both, use

- SOURCE= name of single node that has supply capability
- SUPPLY= the amount of supply at SOURCE
- SINK= name of single node that demands flow
- DEMAND= the amount of flow SINK demands

Do not specify that a constraint has zero right-hand-side values. That is the default. The only time it might be practical to specify a zero rhs is in observations of CONDATA read early so that PROC NETFLOW can infer that a row is a constraint. This could prevent coefficient data from being put aside because PROC NETFLOW did not know the row was a constraint.

Names of Things

To cut data read time and memory requirements, reduce the number of bytes in the longest node name, longest arc name, and longest constraint name to 8 bytes or less. The longer a name, the more bytes must be stored and compared with other names.

If an arc has no constraint coefficients, do not give it a name in the NAME list variable in the ARCDATA= data set. Names for such arcs serve no purpose.

PROC NETFLOW can have a default name for each arc. If an arc is directed from node *tailname* toward node *headname*, the default name for that arc is *tailname_headname*. If you do not want PROC NETFLOW to use these default arc names, specify NAMECTRL=1. Otherwise, PROC NETFLOW must use memory for storing node names and these node names must be searched often.

If you want to use the default *tailname_headname name*, that is, NAMECTRL=2 or NAMECTRL=3, do not use underscores in node names. If a CONDATA has a dense format and has a variable in the VAR list A_B_C_D, or if the value A_B_C_D is encountered as a value of the COLUMN list variable when reading CONDATA that has the sparse format, PROC NETFLOW first looks for a node named A. If it finds it, it looks for a node called B_C_D. It then looks for a node with the name A_B and possibly a node with name C_D. A search for a node named A_B_C and possibly a node named D is done. Underscores could have caused PROC NETFLOW to look unnecessarily for nonexistent nodes. Searching for node names can be expensive, and the amount of memory to store node names large. It might be better to assign the arc name A_B_C_D directly to an arc by having that value as a NAME list variable value for that arc in ARCDATA and specify NAMECTRL=1.

Other Ways to Speed Up Data Reads

Use warm starts as much as possible.

- WARM indicates that the input SAS data sets contain a warm start.

The data read of a warm start is much faster than a cold start data read. The model size is known before the read starts. The observations of the NODEDATA= or DUALIN= data sets have observations ordered by node name and constraint name. Information is stored directly in the data structures used by PROC NETFLOW. For a cold start, much of preprocessing must be performed before the information can be stored in the same way. And using a warm start can greatly reduce the time PROC NETFLOW spends doing optimization.

- SAME_NONARC_DATA is an option that excludes data from processing.

This option indicates that the warm start nonarc variable data in ARCDATA is read and any nonarc variable data in CONDATA is to be ignored. Use this option if it is applicable, or when CONDATA has no nonarc variable data, or such data are duplicated in ARCDATA. ARCDATA is always read before CONDATA.

Arcs and nonarc variables can have associated with them values or quantities that have no bearing with the optimization. This information is given in ARCDATA in the ID list variables. For example, in a distribution

problem, information such as truck number and driver's name can be associated with each arc. This is useful when a solution is saved in an output SAS data set. However, PROC NETFLOW needs to reserve memory to process this information when data are being read. For large problems when memory is scarce, it might be better to remove ancillary data from ARCDATA. After PROC NETFLOW runs, use SAS software to merge this information into the output data sets that contain the optimal solution.

Macro Variable _ORNETFL

The NETFLOW procedure creates and initializes a SAS macro variable called _ORNETFL. After exiting the procedure, you can use `%put &_ORNETFL;` to view details about the optimization.

When the network simplex method is used, the value of _ORNETFL consists of the following parts:

- ERROR_STATUS, indicating the existence or absence of any errors
- OPT_STATUS, the stage of the optimization, or what solution has been found

Ideally, at the end of a PROC NETFLOW run in which the network simplex method is used, _ORNETFL has the following value:

```
ERROR_STATUS=OK OPT_STATUS=OPTIMAL OBJECTIVE=x
SOLUTION=OPTIMAL
```

At the end of a PROC NETFLOW run in which the interior point algorithm is used, _ORNETFL should have the following value:

```
ERROR_STATUS=OK SOLUTION=OPTIMAL OBJECTIVE=x
ITERATIONS=x ITERATING_TIME=x SOLUTION_TIME=x
```

Nontrailing blank characters that are unnecessary are removed. If the preprocessor detects that a problem with a network component is infeasible, and you specify that the interior point algorithm should be used, _ORNETFL has the following value:

```
ERROR_STATUS=OK SOLUTION=INFEASIBLE
ITERATIONS=0 ITERATING_TIME=0 SOLUTION_TIME=0
```

The same value is assigned to the _ORNETFL macro variable if the preprocessor detects that an LP problem is infeasible.

Table 6.11 lists alternate values for the _ORNETFL value parts.

Table 6.11 PROC NETFLOW _ORNETFL Macro Values

Keyword	Value	Meaning
ERROR_STATUS	OK	No errors
	MEMORY	Memory request failed
	IO	Input/output error
	DATA	Error in the data
	BUG	Error with PROC NETFLOW
	SEMANTIC	Semantic error
	SYNTAX	Syntax error
	UNKNOWN	Unknown error
OPT_STATUS	START	No optimization has been done
	STAGE_1	Performing stage 1 optimization
	UNCON_OPT	Reached unconstrained optimum, but there are side constraints
	STAGE_2	Performing stage 2 optimization
	OPTIMAL	Reached the optimum
OBJECTIVE	*objective*	Total cost or profit
MINFLOW	*minflow*	If MAXFLOW and MAXIMIZE are specified at the same time
MAXFLOW	*maxflow*	If MAXFLOW is specified
SHORTEST_PATH	*shortpath*	If SHORTPATH is specified
LONGEST_PATH	*longpath*	If SHORTPATH and MAXIMIZE are specified at the same time
SOLUTION	NONOPTIMAL	More optimization is required
	STAGE_2_REQUIRED	Reached unconstrained optimum, stage 2 optimization is required
	OPTIMAL	Have determined the optimum
	INFEASIBLE	Infeasible; no solution exists
	UNRESOLVED_OPTIMALITY _OR_FEASIBILITY	The optimization process stops before optimality or infeasibility can be proven.
	MAXITERB_OPTION _STOPPED_OPTIMIZATION	The interior point algorithm stops after performing maximal number of iterations specified by the MAXITERB= option

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory used by the SAS System. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit, which is sufficient for many applications. However, to solve most realistic optimization problems, the NETFLOW procedure might require more memory. Increasing the memory limit can reduce the chance of an out-of-memory condition.

NOTE: The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify -MEMSIZE 0 to indicate all available memory should be used, but this setting should be used with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify -MEMSIZE 0. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, -MEMSIZE 500M might be sufficient to enable the procedure to run without an out-of-memory condition. When problems have millions of variables, -MEMSIZE 1000M or higher might be needed. These are "rules of thumb"—problems with atypical structure, density, or other characteristics can increase the optimizer's memory requirements.

The MEMSIZE option can be specified at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the SAS Companion for your operating environment.

To report a procedure's memory consumption, you can use the FULLSTIMER option. The syntax is described in the SAS Companion for your operating environment.

The Interior Point Algorithm: NETFLOW Procedure

Introduction

The simplex algorithm, developed shortly after World War II, was the main method used to solve linear programming problems. Over the last fifteen years, the interior point algorithm has been developed to also solve linear programming problems. From the start it showed great theoretical promise, and considerable research in the area resulted in practical implementations that performed competitively with the simplex algorithm. More recently, interior point algorithms have evolved to become superior to the simplex algorithm, in general, especially when the problems are large.

The interior point algorithm has been implemented in PROC NETFLOW. This algorithm can be used to solve linear programs as well as network problems. When PROC NETFLOW detects that the problem has no network component, it automatically invokes the interior point algorithm to solve the problem. The data required by PROC NETFLOW for a linear program resembles the data for nonarc variables and constraints for constrained network problems.

If PROC NETFLOW does detect a network component to the problem (the problem has arcs), you must specify the option INTPOINT in the PROC NETFLOW statement if you want to use the interior point algorithm. PROC NETFLOW first converts the constrained network model into an equivalent linear programming formulation, solves that, then converts the LP back to the network model. These models remain conceptually easy since they are based on network diagrams that represent the problem pictorially. This procedure accepts the network specification in a format that is particularly suited to networks. This not only simplifies problem description but also aids in the interpretation of the solution. The conversions to and from the equivalent LP are done "behind the scenes."

There are many variations of interior point algorithms. PROC NETFLOW uses the Primal-Dual with Predictor-Corrector algorithm. This algorithm and related theory can be found in the texts by Roos, Terlaky, and Vial (1997), Wright (1997), and Ye (1996).

The remainder of this section is split into two parts. In the first part, how you use PROC NETFLOW's interior point algorithm to solve network problems is described. In the second part, using PROC NETFLOW to solve linear programming problems (its interior point algorithm must be used) is described. Both parts are organized similarly:

- The way data are supplied to PROC NETFLOW is outlined in a "Getting Started" subsection.
- An "Introductory Example" is solved to demonstrate how the data are set up, how PROC NETFLOW is used to compute the solution, and how the optimum is saved.
- More sophisticated ways to use PROC NETFLOW interactively are detailed in an "Interactivity" subsection.
- A "Functional Summary" lists the statements and options that can be used to control PROC NETFLOW. Of particular interest are the options used to control the optimizer, and the way the solution is saved into output data sets or is displayed.

The Linear Programs section has additional subsections:

- "Mathematical Description of LP"
- "Interior Point Algorithmic Details," a brief theory of the algorithm containing information about the options that can be specified to control the interior point algorithm.
- "Syntax" subsection, which is a subset of the syntax when the simplex algorithm is used. Gone are the statements and lists relevant only when the simplex algorithm is used.

Network Models: Interior Point Algorithm

The data required by PROC NETFLOW for a network problem is identical whether the simplex algorithm or the interior point algorithm is used as the optimizer. By default, the simplex algorithm is used for problems with a network component. To use the interior point algorithm, all you need to do is specify the INTPOINT option in the PROC NETFLOW statement. You can optionally specify some options that control the interior point algorithm, of which there are only a few. The interior point algorithm is remarkably robust when reasonable choices are made during the design and implementation, so it does not need to be tuned to the same extent as the simplex algorithm.

When to Use INTPOINT: Network Models: Interior Point Algorithm

PROC NETFLOW uses the primal simplex network algorithm and the primal partitioning algorithm to solve constrained network problems. These algorithms are fast, since they take advantage of algebraic properties of the network component of the problem.

If the network component of the model is large compared to the side constraint component, PROC NETFLOW's optimizer can store what would otherwise be a large matrix as a spanning tree computer data structure. Computations involving the spanning tree data structure can be performed much faster than those using matrices. Only the nonnetwork part of the problem, hopefully quite small, needs to be manipulated by PROC NETFLOW as matrices.

In contrast, LP optimizers must contend with matrices that can be large for large problems. Arithmetic operations on matrices often accumulate rounding errors that cause difficulties for the algorithm. So in

addition to the performance improvements, network optimization is generally more numerically stable than LP optimization.

The nodal flow conservation constraints do not need to be specified in the network model. They are implied by the network structure. However, flow conservation constraints do make up the data for the equivalent LP model. If you have an LP that is small after the flow conservation constraints are removed, that problem is a definite candidate for solution by PROC NETFLOW's specialized simplex method.

However, some constrained network problems are solved more quickly by the interior point algorithm than the network optimizer in PROC NETFLOW. Usually, they have a large number of side constraints or nonarc variables. These models are more like LPs than network problems. The network component of the problem is so small that PROC NETFLOW's network simplex method cannot recoup the effort to exploit that component rather than treat the whole problem as an LP. If this is the case, it is worthwhile to get PROC NETFLOW to convert a constrained network problem to the equivalent LP and use its interior point algorithm. This conversion must be done before any optimization has been performed (specify the INTPOINT option in the PROC NETFLOW statement).

Even though some network problems are better solved by converting them to an LP, the input data and the output solution are more conveniently maintained as networks. You retain the advantages of casting problems as networks: ease of problem generation and expansion when more detail is required. The model and optimal solutions are easy to understand, as a network can be drawn.

Getting Started: Network Models: Interior Point Algorithm

To solve network programming problems with side constraints using PROC NETFLOW, you save a representation of the network and the side constraints in three SAS data sets. These data sets are then passed to PROC NETFLOW for solution. There are various forms that a problem's data can take. You can use any one or a combination of several of these forms.

The NODEDATA= data set contains the names of the supply and demand nodes and the supply or demand associated with each. These are the elements in the column vector b in problem (NPSC).

The ARCDATA= data set contains information about the variables of the problem. Usually these are arcs, but there can also be data related to nonarc variables in the ARCDATA= data set. If there are no arcs, this is a linear programming problem.

An arc is identified by the names of its tail node (where it originates) and head node (where it is directed). Each observation can be used to identify an arc in the network and, optionally, the cost per flow unit across the arc, the arc's lower flow bound, capacity, and name. These data are associated with the matrix F and the vectors c, l, and u in problem (NPSC).

NOTE: Although F is a node-arc incidence matrix, it is specified in the ARCDATA= data set by arc definitions. Do not explicitly specify these flow conservation constraints as constraints of the problem.

In addition, the ARCDATA= data set can be used to specify information about nonarc variables, including objective function coefficients, lower and upper value bounds, and names. These data are the elements of the vectors d, m, and v in problem (NPSC). Data for an arc or nonarc variable can be given in more than one observation.

Supply and demand data also can be specified in the ARCDATA= data set. In such a case, the NODEDATA= data set may not be needed.

The CONDATA= data set describes the side constraints and their right-hand sides. These data are elements of the matrices H and Q and the vector r. Constraint types are also specified in the CONDATA= data set. You can include in this data set upper bound values or capacities, lower flow or value bounds, and costs or

objective function coefficients. It is possible to give all information about some or all nonarc variables in the CONDATA= data set.

An arc or nonarc variable is identified in this data set by its name. If you specify an arc's name in the ARCDATA= data set, then this name is used to associate data in the CONDATA= data set with that arc. Each arc also has a default name that is the name of the tail and head node of the arc concatenated together and separated by an underscore character; *tail_head*, for example.

If you use the dense side constraint input format and want to use the default arc names, these arc names are names of SAS variables in the VAR list of the CONDATA= data set.

If you use the sparse side constraint input format (also described later) and want to use the default arc names, these arc names are values of the COLUMN list SAS variable of the CONDATA= data set.

When using the interior point algorithm, the execution of PROC NETFLOW has two stages. In the preliminary (zeroth) stage, the data are read from the NODEDATA= data set, the ARCDATA= data set, and the CONDATA= data set. Error checking is performed. The model is converted into an equivalent linear program.

In the next stage, the linear program is preprocessed. This is optional but highly recommended. Preprocessing analyzes the model and tries to determine before optimization whether variables can be "fixed" to their optimal values. Knowing that, the model can be modified and these variables dropped out. It can be determined that some constraints are redundant. Sometimes, preprocessing succeeds in reducing the size of the problem, thereby making the subsequent optimization easier and faster.

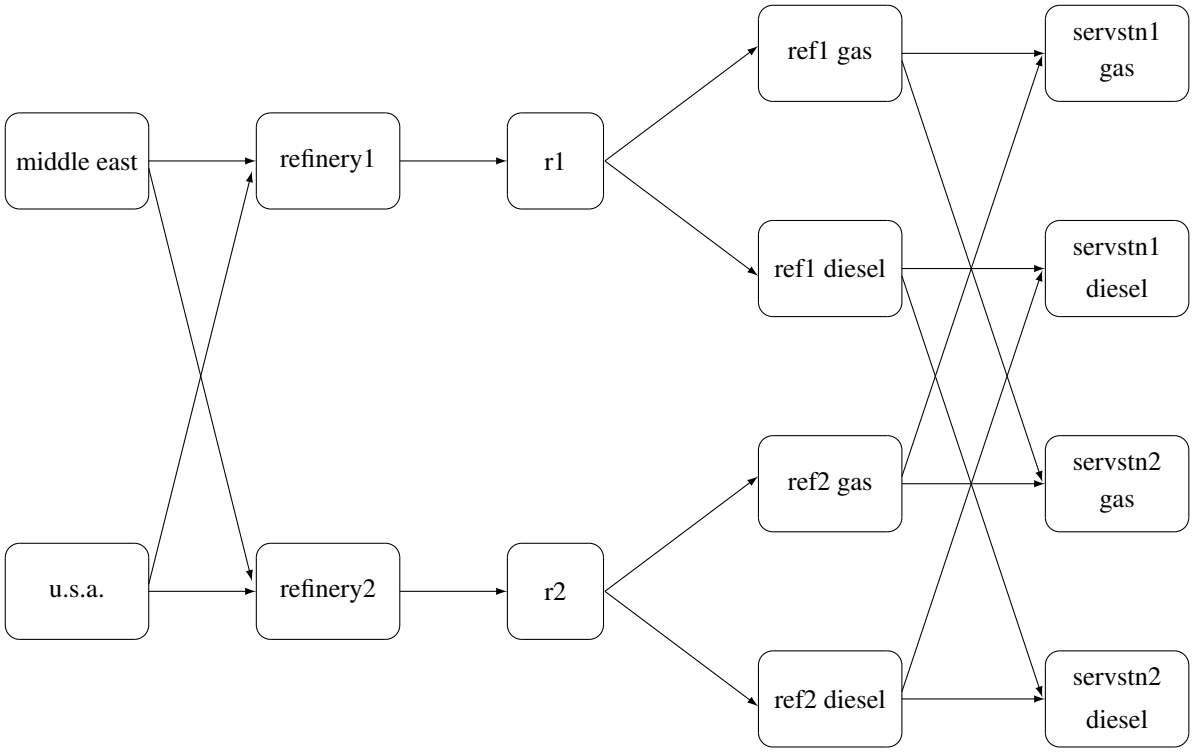
The optimal solution to the linear program is then found. The linear program is converted back to the original constrained network problem, and the optimum for this is derived from the optimum of the equivalent linear program. If the problem was preprocessed, the model is now post-processed, where fixed variables are reintroduced. The solution can be saved in the CONOUT= data set. This data set is also named in the PROC NETFLOW, RESET, and SAVE statements.

The interior point algorithm cannot efficiently be warm started, so options such as FUTURE1 and FUTURE2 options are irrelevant.

Introductory Example: Network Models: Interior Point Algorithm

Consider the following transshipment problem for an oil company in the section "Introductory Example" on page 315. Recall that crude oil is shipped to refineries where it is processed into gasoline and diesel fuel. The gasoline and diesel fuel are then distributed to service stations. At each stage there are shipping, processing, and distribution costs. Also, there are lower flow bounds and capacities. In addition, there are side constraints to model crude mix stipulations, and model the limitations on the amount of Middle Eastern crude that can be processed by each refinery and the conversion proportions of crude to gasoline and diesel fuel. The network diagram is reproduced in Figure 6.16.

Figure 6.16 Oil Industry Example



To solve this problem with PROC NETFLOW, a representation of the model is saved in three SAS data sets that are identical to the data sets supplied to PROC NETFLOW when the simplex algorithm was used.

To find the minimum cost flow through the network that satisfies the supplies, demands, and side constraints, invoke PROC NETFLOW as follows:

```
proc netflow
   intpoint              /* <<<--- Interior Point used */
   nodedata=noded        /* the supply and demand data */
   arcdata=arcd1         /* the arc descriptions       */
   condata=cond1         /* the side constraints       */
   conout=solution;      /* the solution data set      */
run;
```

The following messages, which appear on the SAS log, summarize the model as read by PROC NETFLOW and note the progress toward a solution:

```
NOTE: Number of nodes= 14 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 180 , total demand= 180 .
NOTE: Number of arcs= 18 .
NOTE: Number of <= side constraints= 0 .
NOTE: Number of == side constraints= 2 .
NOTE: Number of >= side constraints= 2 .
NOTE: Number of side constraint coefficients= 8 .
NOTE: The following messages relate to the equivalent Linear Programming
      problem solved by the Interior Point algorithm.
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 16 .
NOTE: Number of >= constraints= 2 .
NOTE: Number of constraint coefficients= 44 .
NOTE: Number of variables= 18 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 3.
NOTE: After preprocessing, number of >= constraints= 2.
NOTE: The preprocessor eliminated 13 constraints from the problem.
NOTE: The preprocessor eliminated 33 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 5.
NOTE: The preprocessor eliminated 13 variables from the problem.
NOTE: 4 columns, 0 rows and 4 coefficients were added to the problem to handle
      unrestricted variables, variables that are split, and constraint slack or
      surplus variables.
NOTE: There are 10 sub-diagonal nonzeroes in the unfactored A Atranspose matrix.
NOTE: The 5 factor nodes make up 1 supernodes
NOTE: There are 0 nonzero sub-rows or sub-columns outside the supernodal
      triangular regions along the factors leading diagonal.
NOTE: Bound feasibility attained by iteration 1.
NOTE: Dual feasibility attained by iteration 1.
NOTE: Constraint feasibility attained by iteration 1.
NOTE: The Primal-Dual Predictor-Corrector Interior Point algorithm performed 6
      iterations.
NOTE: Optimum reached.
NOTE: Objective= 50875.
NOTE: The data set WORK.SOLUTION has 18 observations and 10 variables.
```

The first set of messages provide statistics on the size of the equivalent linear programming problem. The number of variables may not equal the number of arcs if the problem has nonarc variables. This example has none. To convert a network to an equivalent LP problem, a flow conservation constraint must be created for each node (including an excess or bypass node, if required). This explains why the number of equality side constraints and the number of constraint coefficients change when the interior point algorithm is used.

If the preprocessor was successful in decreasing the problem size, some messages will report how well it did. In this example, the model size was cut in half!

The following set of messages describe aspects of the interior point algorithm. Of particular interest are those concerned with the Cholesky factorization of AA^T where A is the coefficient matrix of the final LP. It

is crucial to preorder the rows and columns of this matrix to prevent *fill-in* and reduce the number of row operations to undertake the factorization. See the section "Interior Point Algorithmic Details" on page 429 for more explanation.

Unlike PROC LP, which displays the solution and other information as output, PROC NETFLOW saves the optimum in output SAS data sets you specify. For this example, the solution is saved in the SOLUTION data set. It can be displayed with PROC PRINT as

```
proc print data=solution;
   var _from_ _to_ _cost_ _capac_ _lo_ _name_
      _supply_ _demand_ _flow_ _fcost_ ;
   sum _fcost_;
   title3 'Constrained Optimum';
run;
```

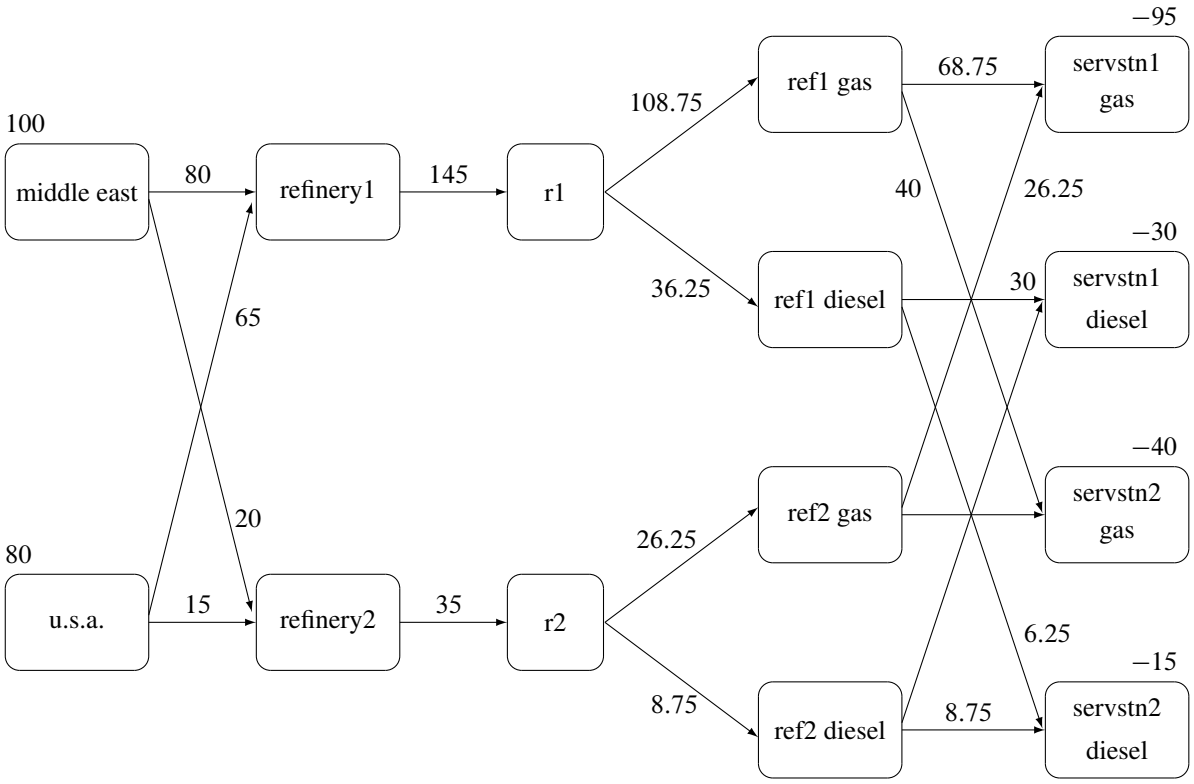
Figure 6.17 CONOUT=SOLUTION

Constrained Optimum

Obs	_from_	_to_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	refinery 1	r1	200	175	50	thruput1	.	.	145.000	29000.00
2	refinery 2	r2	220	100	35	thruput2	.	.	35.000	7700.00
3	r1	ref1 diesel	0	75	0		.	.	36.250	0.00
4	r1	ref1 gas	0	140	0	r1_gas	.	.	108.750	0.00
5	r2	ref2 diesel	0	75	0		.	.	8.750	0.00
6	r2	ref2 gas	0	100	0	r2_gas	.	.	26.250	0.00
7	middle east	refinery 1	63	95	20	m_e_ref1	100	.	80.000	5040.00
8	u.s.a.	refinery 1	55	99999999	0		80	.	65.000	3575.00
9	middle east	refinery 2	81	80	10	m_e_ref2	100	.	20.000	1620.00
10	u.s.a.	refinery 2	49	99999999	0		80	.	15.000	735.00
11	ref1 diesel	servstn1 diesel	18	99999999	0		.	30	30.000	540.00
12	ref2 diesel	servstn1 diesel	36	99999999	0		.	30	0.000	0.00
13	ref1 gas	servstn1 gas	15	70	0		.	95	68.750	1031.25
14	ref2 gas	servstn1 gas	17	35	5		.	95	26.250	446.25
15	ref1 diesel	servstn2 diesel	17	99999999	0		.	15	6.250	106.25
16	ref2 diesel	servstn2 diesel	23	99999999	0		.	15	8.750	201.25
17	ref1 gas	servstn2 gas	22	60	0		.	40	40.000	880.00
18	ref2 gas	servstn2 gas	31	99999999	0		.	40	0.000	0.00
										50875.00

Notice that, in the solution data set (Figure 6.17), the optimal flow through each arc in the network is given in the variable named _FLOW_, and the cost of flow through each arc is given in the variable _FCOST_. As expected, the minimal total cost of the solution found by the interior point algorithm is equal to the minimal total cost of the solution found by the simplex algorithm. In this example, the solutions are the same (within several significant digits), but sometimes the solutions can be different.

Figure 6.18 Oil Industry Solution



Interactivity: Network Models: Interior Point Algorithm

PROC NETFLOW can be used interactively. You begin by giving the PROC NETFLOW statement with INTPOINT specified, and you must specify the ARCDATA= data set. The CONDATA= data set must also be specified if the problem has side constraints. If necessary, specify the NODEDATA= data set.

The variable lists should be given next. If you have variables in the input data sets that have special names (for example, a variable in the ARCDATA= data set named _TAIL_ that has tail nodes of arcs as values), it may not be necessary to have many or any variable lists.

So far, this is the same as when the simplex algorithm is used, except the INTPOINT option is specified in the PROC NETFLOW statement. The PRINT, QUIT, SAVE, SHOW, RESET, and RUN statements follow and can be listed in any order. The QUIT statements can be used only once. The others can be used as many times as needed.

The CONOPT and PIVOT statements are not relevant to the interior point algorithm and should not be used.

Use the RESET or SAVE statement to change the name of the output data set. There is only one output data set, the CONOUT= data set. With the RESET statement, you can also indicate the reasons why optimization should stop (for example, you can indicate the maximum number of iterations that can be performed). PROC NETFLOW then has a chance to either execute the next statement, or, if the next statement is one that PROC NETFLOW does not recognize (the next PROC or DATA step in the SAS session), do any allowed

optimization and finish. If no new statement has been submitted, you are prompted for one. Some options of the RESET statement enable you to control aspects of the interior point algorithm. Specifying certain values for these options can reduce the time it takes to solve a problem. Note that any of the RESET options can be specified in the PROC NETFLOW statement.

The RUN statement starts optimization. Once the optimization has started, it runs until the optimum is reached. The RUN statement should be specified at most once.

The QUIT statement immediately stops PROC NETFLOW. The SAVE statement has options that enable you to name the output data set; information about the current solution is put in this output data set. Use the SHOW statement if you want to examine the values of options of other statements. Information about the amount of optimization that has been done and the STATUS of the current solution can also be displayed using the SHOW statement.

The PRINT statement makes PROC NETFLOW display parts of the problem. The way the PRINT statements are specified are identical whether the interior point algorithm or the simplex algorithm is used, however there are minor differences in what is displayed for each arc, nonarc variable or constraint coefficient.

PRINT ARCS produces information on all arcs. PRINT SOME_ARCS limits this output to a subset of arcs. There are similar PRINT statements for nonarc variables and constraints:

```
PRINT NONARCS;
PRINT SOME_NONARCS;
PRINT CONSTRAINTS;
PRINT SOME_CONS;
```

PRINT CON_ARCS enables you to limit constraint information that is obtained to members of a set of arcs and that have nonzero constraint coefficients in a set of constraints. PRINT CON_NONARCS is the corresponding statement for nonarc variables.

For example, an interactive PROC NETFLOW run might look something like this:

```
proc netflow
       intpoint      /* use the Interior Point algorithm */
       arcdata=data set
       other options;
   variable list specifications;  /* if necessary        */
   reset options;
   print options;                 /* look at problem     */
run;                              /* do the optimization */
   print options;    /* look at the optimal solution     */
   save options;     /* keep optimal solution            */
```

If you are interested only in finding the optimal solution, have used SAS variables that have special names in the input data sets, and want to use default settings for everything, then the following statement is all you need.

proc netflow intpoint arcdata= *data set* **;**

Functional Summary: Network Models, Interior Point Algorithm

The following table outlines the options available for the NETFLOW procedure when the interior point algorithm is being used, classified by function.

Table 6.12 Functional Summary, Network Models

Description	Statement	Option
Input Data Set Options:		
Arcs input data set	PROC NETFLOW	ARCDATA=
Nodes input data set	PROC NETFLOW	NODEDATA=
Constraint input data set	PROC NETFLOW	CONDATA=
Output Data Set Option:		
Constrained solution data set	PROC NETFLOW	CONOUT=
Data Set Read Options:		
CONDATA has sparse data format	PROC NETFLOW	SPARSECONDATA
Default constraint type	PROC NETFLOW	DEFCONTYPE=
Special COLUMN variable value	PROC NETFLOW	TYPEOBS=
Special COLUMN variable value	PROC NETFLOW	RHSOBS=
Used to interpret arc and nonarc variable names	PROC NETFLOW	NAMECTRL=
No new nonarc variables	PROC NETFLOW	SAME_NONARC_DATA
No nonarc data in ARCDATA	PROC NETFLOW	ARCS_ONLY_ARCDATA
Data for an arc found once in ARCDATA	PROC NETFLOW	ARC_SINGLE_OBS
Data for a constraint found once in CONDATA	PROC NETFLOW	CON_SINGLE_OBS
Data for a coefficient found once in CONDATA	PROC NETFLOW	NON_REPLIC=
Data are grouped, exploited during data read	PROC NETFLOW	GROUPED=
Problem Size Specification Options:		
Approximate number of nodes	PROC NETFLOW	NNODES=
Approximate number of arcs	PROC NETFLOW	NARCS=
Approximate number of nonarc variables	PROC NETFLOW	NNAS=
Approximate number of coefficients	PROC NETFLOW	NCOEFS=
Approximate number of constraints	PROC NETFLOW	NCONS=
Network Options:		
Default arc cost	PROC NETFLOW	DEFCOST=
Default arc capacity	PROC NETFLOW	DEFCAPACITY=
Default arc lower flow bound	PROC NETFLOW	DEFMINFLOW=
Network's only supply node	PROC NETFLOW	SOURCE=
SOURCE's supply capability	PROC NETFLOW	SUPPLY=
Network's only demand node	PROC NETFLOW	SINK=
SINK's demand	PROC NETFLOW	DEMAND=
Convey excess supply/demand through network	PROC NETFLOW	THRUNET
Find maximal flow between SOURCE and SINK	PROC NETFLOW	MAXFLOW
Cost of bypass arc for MAXFLOW problem	PROC NETFLOW	BYPASSDIVIDE=

Description	Statement	Option
Find shortest path from SOURCE to SINK	PROC NETFLOW	SHORTPATH
Memory Control Options:		
Issue memory usage messages to SAS log	PROC NETFLOW	MEMREP
Number of bytes to use for main memory	PROC NETFLOW	BYTES=
Proportion of memory for arrays	PROC NETFLOW	COREFACTOR=
Maximum bytes for a single array	PROC NETFLOW	MAXARRAYBYTES=
Interior Point Algorithm Options:		
Use interior point algorithm	PROC NETFLOW	INTPOINT
Factorization method	RESET	FACT_METHOD=
Allowed amount of dual infeasibility	RESET	TOLDINF=
Allowed amount of primal infeasibility	RESET	TOLPINF=
Allowed total amount of dual infeasibility	RESET	TOLTOTDINF=
Allowed total amount of primal infeasibility	RESET	TOLTOTPINF=
Cut-off tolerance for Cholesky factorization	RESET	CHOLTINYTOL=
Density threshold for Cholesky processing	RESET	DENSETHR=
Step-length multiplier	RESET	PDSTEPMULT=
Preprocessing type	RESET	PRSLTYPE=
Print optimization progress on SAS log	RESET	PRINTLEVEL2=
Write optimization time to SAS log	RESET	OPTIM_TIMER
Interior Point Stopping Criteria Options:		
Maximum number of interior point iterations	RESET	MAXITERB=
Primal-dual (duality) gap tolerance	RESET	PDGAPTOL=
Stop because of complementarity	RESET	STOP_C=
Stop because of duality gap	RESET	STOP_DG=
Stop because of $infeas_b$	RESET	STOP_IB=
Stop because of $infeas_c$	RESET	STOP_IC=
Stop because of $infeas_d$	RESET	STOP_ID=
Stop because of complementarity	RESET	AND_STOP_C=
Stop because of duality gap	RESET	AND_STOP_DG=
Stop because of $infeas_b$	RESET	AND_STOP_IB=
Stop because of $infeas_c$	RESET	AND_STOP_IC=
Stop because of $infeas_d$	RESET	AND_STOP_ID=
Stop because of complementarity	RESET	KEEPGOING_C=
Stop because of duality gap	RESET	KEEPGOING_DG=
Stop because of $infeas_b$	RESET	KEEPGOING_IB=
Stop because of $infeas_c$	RESET	KEEPGOING_IC=
Stop because of $infeas_d$	RESET	KEEPGOING_ID=
Stop because of complementarity	RESET	AND_KEEPGOING_C=
Stop because of duality gap	RESET	AND_KEEPGOING_DG=
Stop because of $infeas_b$	RESET	AND_KEEPGOING_IB=
Stop because of $infeas_c$	RESET	AND_KEEPGOING_IC=
Stop because of $infeas_d$	RESET	AND_KEEPGOING_ID=

Description	Statement	Option
PRINT Statement Options:		
Display everything	PRINT	PROBLEM
Display arc information	PRINT	ARCS
Display nonarc variable information	PRINT	NONARCS
Display variable information	PRINT	VARIABLES
Display constraint information	PRINT	CONSTRAINTS
Display information for some arcs	PRINT	SOME_ARCS
Display information for some nonarc variables	PRINT	SOME_NONARCS
Display information for some variables	PRINT	SOME_VARIABLES
Display information for some constraints	PRINT	SOME_CONS
Display information for some constraints associated with some arcs	PRINT	CON_ARCS
Display information for some constraints associated with some nonarc variables	PRINT	CON_NONARCS
Display information for some constraints associated with some variables	PRINT	CON_VARIABLES
PRINT Statement Qualifiers:		
Produce a short report	PRINT	/ SHORT
Produce a long report	PRINT	/ LONG
Display arcs/variables with zero flow/value	PRINT	/ ZERO
Display arcs/variables with nonzero flow/value	PRINT	/ NONZERO
SHOW Statement Options:		
Show problem, optimization status	SHOW	STATUS
Show network model parameters	SHOW	NETSTMT
Show data sets that have been or will be created	SHOW	DATASETS
Miscellaneous Options:		
Infinity value	PROC NETFLOW	INFINITY=
Scale constraint row, nonarc variable column coefficients, or both	PROC NETFLOW	SCALE=
Maximization instead of minimization	PROC NETFLOW	MAXIMIZE

Linear Programming Models: Interior Point Algorithm

By default, the interior point algorithm is used for problems without a network component, that is, a linear programming problem. You do not need to specify the INTPOINT option in the PROC NETFLOW statement (although you will do no harm if you do).

Data for a linear programming problem resembles the data for side constraints and nonarc variables supplied to PROC NETFLOW when solving a constrained network problem. It is also very similar to the data required by the LP procedure.

Mathematical Description of LP

If the network component of NPSC is removed, the result is the mathematical description of the linear programming problem. If an LP has g variables, and k constraints, then the formal statement of the problem solved by PROC NETFLOW is

$$\begin{array}{ll} \text{minimize} & d^T z \\ \text{subject to} & Qz \{\geq, =, \leq\} r \\ & m \leq z \leq v \end{array}$$

where

- d is the $g \times 1$ objective function coefficient vector
- z is the $g \times 1$ variable value vector
- Q is the $k \times g$ constraint coefficient matrix for variables, where $Q_{i,j}$ is the coefficient of variable j in the ith constraint
- r is the $k \times 1$ side constraint right-hand-side vector
- m is the $g \times 1$ variable value lower bound vector
- v is the $g \times 1$ variable value upper bound vector

Interior Point Algorithmic Details

After preprocessing, the linear program to be solved is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

This is the *primal* problem. The matrices d, z, and Q of NPSC have been renamed c, x, and A respectively, as these symbols are by convention used more, the problem to be solved is different from the original because of preprocessing, and there has been a change of primal variable to transform the LP into one whose variables have zero lower bounds. To simplify the algebra here, assume that variables have infinite bounds, and constraints are equalities. (Interior point algorithms do efficiently handle finite bounds, and it is easy to introduce primal slack variables to change inequalities into equalities.) The problem has n variables; i is a variable number, k is an iteration number, and if used as a subscript or superscript it denotes "of iteration k".

There exists an equivalent problem, the *dual* problem, stated as

$$\begin{array}{ll} \text{maximize} & b^T y \\ \text{subject to} & A^T y + s = c \\ & s \geq 0 \end{array}$$

where y are dual variables, and s are dual constraint slacks.

The interior point algorithm solves the system of equations to satisfy the Karush-Kuhn-Tucker (KKT) conditions for optimality:

$$\begin{aligned} Ax &= b \\ A^T y + s &= c \\ x^T s &= 0 \\ x &\geq 0 \\ s &\geq 0 \end{aligned}$$

These are the conditions for feasibility, with the *complementarity* condition $x^T s = 0$ added. Complementarity forces the optimal objectives of the primal and dual to be equal, $c^T x_{opt} = b^T y_{opt}$, as

$$\begin{aligned} 0 &= x_{opt}^T s_{opt} = s_{opt}^T x_{opt} = (c - A^T y_{opt})^T x_{opt} \\ &= c^T x_{opt} - y_{opt}^T (A x_{opt}) = c^T x_{opt} - b^T y_{opt} \end{aligned}$$

Before the optimum is reached, a solution (x, y, s) may not satisfy the KKT conditions:

- Primal constraints may be violated, $infeas_c = b - Ax \neq 0$.
- Dual constraints may be violated, $infeas_d = c - A^T y - s \neq 0$.
- Complementarity may not be satisfied, $x^T s = c^T x - b^T y \neq 0$.
  This is called the *duality gap*.

The interior point algorithm works by using Newton's method to find a direction to move $(\Delta x^k, \Delta y^k, \Delta s^k)$ from the current solution (x^k, y^k, s^k) toward a better solution:

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where α is the *step length* and is assigned a value as large as possible but ≤ 1.0 and not so large that an x_i^{k+1} or s_i^{k+1} is "too close" to zero. The direction in which to move is found using the following:

$$\begin{aligned} A\Delta x^k &= -infeas_c \\ A^T \Delta y^k + \Delta s^k &= -infeas_d \\ S^k \Delta x^k + X^k \Delta s^k &= -X^k S^k e \end{aligned}$$

where $S = \mathrm{diag}(s)$, $X = \mathrm{diag}(x)$, and e is a vector with all elements equal to 1.

To greatly improve performance, the third equation is changed to

$$S^k \Delta x^k + X^k \Delta s^k = -X^k S^k e + \sigma_k \mu_k e$$

where $\mu_k = 1/nX^k S^k e$, the average complementarity, and $0 \leq \sigma_k \leq 1$.

The effect now is to find a direction in which to move to reduce infeasibilities and to reduce the complementarity toward zero, but if any $x_i^k s_i^k$ is too close to zero, it is "nudged out" to μ, and any $x_i^k s_i^k$ that is larger than μ is "nudged into" μ. A σ_k close to or equal to 0.0 biases a direction toward the optimum, and a value for σ_k close to or equal to 1.0 "centers" the direction toward a point where all pairwise products $x_i^k s_i^k = \mu$. Such points make up the *central path* in the interior. Although centering directions make little, if any, progress in reducing μ and moving the solution closer to the optimum, substantial progress toward the optimum can usually be made in the next iteration.

The central path is crucial to why the interior point algorithm is so efficient. This path "guides" the algorithm to the optimum through the interior of feasible space. Without centering, the algorithm would find a series of solutions near each other close to the boundary of feasible space. Step lengths along the direction would be small and many more iterations would probably be required to reach the optimum.

The calculation of the direction is the most time-consuming step of the interior point algorithm. Assume the kth iteration is being performed, so the subscript and superscript k can be dropped from the algebra:

$$\begin{aligned} A\Delta x &= -infeas_c \\ A^T \Delta y + \Delta s &= -infeas_d \\ S\Delta x + X\Delta s &= -XSe + \sigma\mu e \end{aligned}$$

Rearranging the second equation,

$$\Delta s = -infeas_d - A^T \Delta y$$

Rearranging the third equation,

$$\begin{aligned} \Delta s &= X^{-1}(-S\Delta x - XSe + \sigma\mu e) \\ \Delta s &= -\Theta\Delta x - Se + X^{-1}\sigma\mu e \end{aligned}$$

where $\Theta = SX^{-1}$.

Equating these two expressions for Δs and rearranging,

$$\begin{aligned} -\Theta\Delta x - Se + X^{-1}\sigma\mu e &= -infeas_d - A^T \Delta y \\ -\Theta\Delta x &= Se - X^{-1}\sigma\mu e - infeas_d - A^T \Delta y \\ \Delta x &= \Theta^{-1}(-Se + X^{-1}\sigma\mu e + infeas_d + A^T \Delta y) \\ \Delta x &= \rho + \Theta^{-1} A^T \Delta y \end{aligned}$$

where $\rho = \Theta^{-1}(-Se + X^{-1}\sigma\mu e + infeas_d)$.

Substituting into the first direction equation,

$$\begin{aligned} A\Delta x &= -infeas_c \\ A(\rho + \Theta^{-1}A^T\Delta y) &= -infeas_c \\ A\Theta^{-1}A^T\Delta y &= -infeas_c - A\rho \\ \Delta y &= (A\Theta^{-1}A^T)^{-1}(-infeas_c - A\rho) \end{aligned}$$

Θ, ρ, Δy, Δx and Δs are calculated in that order. The hardest term is the factorization of the $(A\Theta^{-1}A^T)$ matrix to determine Δy. Fortunately, although the *values* of $(A\Theta^{-1}A^T)$ are different for each iteration, the *locations* of the nonzeros in this matrix remain fixed; the nonzero locations are the same as those in the matrix (AA^T). This is due to $\Theta^{-1} = XS^{-1}$ being a diagonal matrix, which has the effect of merely scaling the columns of (AA^T).

The fact that the nonzeros in $A\Theta^{-1}A^T$ have a constant pattern is exploited by all interior point algorithms, and is a major reason for their excellent performance. Before iterations begin, AA^T is examined and its rows and columns are permuted so that during Cholesky Factorization, the number of *fill-ins* created is smaller. A list of arithmetic operations to perform the factorization is saved in concise computer data structures (working with memory locations rather than actual numerical values). This is called *symbolic factorization*. During iterations, when memory has been initialized with numerical values, the operations list is performed sequentially. Determining how the factorization should be performed again and again is unnecessary.

The Primal-Dual Predictor-Corrector Interior Point Algorithm

The variant of the interior point algorithm implemented in PROC NETFLOW is a Primal-Dual Predictor-Corrector interior point algorithm. At first, Newton's method is used to find a direction to move $(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff})$, but calculated as if μ is zero, that is, a step with no centering, known as an *affine* step:

$$\begin{aligned} A\Delta x^k_{aff} &= -infeas_c \\ A^T\Delta y^k_{aff} + \Delta s^k_{aff} &= -infeas_d \\ S^k\Delta x^k_{aff} + X^k\Delta s^k_{aff} &= -X^kS^ke \\ (x^k_{aff}, y^k_{aff}, s^k_{aff}) &= (x^k, y^k, s^k) + \alpha(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff}) \end{aligned}$$

where α is the *step length* as before.

Complementarity x^Ts is calculated at $(x^k_{aff}, y^k_{aff}, s^k_{aff})$ and compared with the complementarity at the starting point (x^k, y^k, s^k), and the success of the affine step is gauged. If the affine step was successful in reducing the complementarity by a substantial amount, the need for centering is not great, and the value of σ_k in the following linear system is assigned a value close to zero. If, however, the affine step was unsuccessful, centering would be beneficial, and the value of σ_k in the following linear system is assigned a value closer to 1.0. The value of σ_k is therefore adaptively altered depending on the progress made toward the optimum.

A second linear system is solved to determine a centering vector $(\Delta x_c^k, \Delta y_c^k, \Delta s_c^k)$ from $(x_{aff}^k, y_{aff}^k, s_{aff}^k)$:

$$\begin{aligned} A\Delta x_c^k &= 0 \\ A^T \Delta y_c^k + \Delta s_c^k &= 0 \\ S^k \Delta x_c^k + X^k \Delta s_c^k &= -X^k S^k e \\ S^k \Delta x^k + X^k \Delta s^k &= -X_{aff}^k S_{aff}^k e + \sigma_k \mu_k e \end{aligned}$$

Then

$$(\Delta x^k, \Delta y^k, \Delta s^k) = (\Delta x_{aff}^k, \Delta y_{aff}^k, \Delta s_{aff}^k) + (\Delta x_c^k, \Delta y_c^k, \Delta s_c^k)$$

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where, as before, α is the *step length* assigned a value as large as possible but not so large that an x_i^{k+1} or s_i^{k+1} is "too close" to zero.

Although the Predictor-Corrector variant entails solving two linear system instead of one, fewer iterations are usually required to reach the optimum. The additional overhead of calculating the second linear system is small, as the factorization of the $(A\Theta^{-1}A^T)$ matrix has already been performed to solve the first linear system.

Stopping Criteria

There are several reasons why PROC NETFLOW stops interior point optimization. Optimization stops when

- the number of iteration equals MAXITERB=*m*
- the relative gap $(duality\ gap/(c^T x))$ between the primal and dual objectives is smaller than the value of the PDGAPTOL= option, and both the primal and dual problems are feasible. Duality gap is defined in the section "Interior Point Algorithmic Details" on page 429.

PROC NETFLOW may stop optimization when it detects that the rate at which the complementarity or dualitygap is being reduced is too slow, that is, there are consecutive iterations when the complementarity or duality gap has stopped getting smaller and the infeasibilities, if nonzero, have also stalled. Sometimes, this indicates the problem is infeasible.

The reasons to stop optimization outlined in the previous paragraph will be termed the *usual* stopping conditions in the following explanation.

However, when solving some problems, especially if the problems are large, the usual stopping criteria are inappropriate. PROC NETFLOW might stop prematurely. If it were allowed to perform additional optimization, a better solution would be found. On other occasions, PROC NETFLOW might do too much work. A sufficiently good solution might be reached several iterations before PROC NETFLOW eventually stops.

You can see PROC NETFLOW's progress to the optimum by specifying PRINTLEVEL2=2. PROC NETFLOW will produce a table on the SAS log. A row of the table is generated during each iteration and consists of values of the affine step complementarity, the complementarity of the solution for the next iteration, the

total bound infeasibility $\sum_{i=1}^{n} infeas_{bi}$ (see the $infeas_b$ array in the section "Interior Point: Upper Bounds" on page 436), the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci}$ (see the $infeas_c$ array in the section "Interior Point Algorithmic Details" on page 429), and the total dual infeasibility $\sum_{i=1}^{n} infeas_{di}$ (see the $infeas_d$ array in the section "Interior Point Algorithmic Details" on page 429). As optimization progresses, the values in all columns should converge to zero.

To tailor stopping criteria to your problem, you can use two sets of parameters: the STOP_x and the KEEPGOING_x parameters. The STOP_x parameters (STOP_C, STOP_DG, STOP_IB, STOP_IC, and STOP_ID) are used to test for some condition at the beginning of each iteration and if met, to stop immediately. The KEEPGOING_x parameters (KEEPGOING_C, KEEPGOING_DG, KEEPGOING_IB, KEEPGOING_IC, and KEEPGOING_ID) are used when PROC NETFLOW would ordinarily stop but does not if some conditions are not met.

For the sake of conciseness, a set of options will be referred to as the part of the option name they have in common followed by the suffix x. For example, STOP_C, STOP_DG, STOP_IB, STOP_IC, and STOP_ID will collectively be referred to as STOP_x.

At the beginning of each iteration, PROC NETFLOW will test whether complementarity is $\leq$ STOP_C (provided you have specified a STOP_C parameter) and if it is, PROC NETFLOW will stop. If the duality gap is $\leq$ STOP_DG (provided you have specified a STOP_DG parameter), PROC NETFLOW will stop immediately. This is also true for the other STOP_x parameters that are related to infeasibilities, STOP_IB, STOP_IC, and STOP_ID.

For example, if you want PROC NETFLOW to stop optimizing for the usual stopping conditions, plus the additional condition, complementarity ≤ 100 *or* duality gap ≤ 0.001, then use

```
proc netflow stop_c=100 stop_dg=0.001
```

If you want PROC NETFLOW to stop optimizing for the usual stopping conditions, plus the additional condition, complementarity ≤ 1000 *and* duality gap ≤ 0.001 *and* constraint infeasibility ≤ 0.0001, then use

```
proc netflow
   and_stop_c=1000 and_stop_dg=0.01 and_stop_ic=0.0001
```

Unlike the STOP_x parameters that cause PROC NETFLOW to stop when any one of them is satisfied, the corresponding AND_STOP_x parameters (AND_STOP_C, AND_STOP_DG, AND_STOP_IB, AND_STOP_IC, and AND_STOP_ID) cause PROC NETFLOW to stop only if all (more precisely, all that are specified) options are satisfied. For example, if PROC NETFLOW should stop when

- complementarity ≤ 100 *or* duality gap ≤ 0.001 or
- complementarity ≤ 1000 *and* duality gap ≤ 0.001 *and* constraint infeasibility ≤ 0.000

then use

```
proc netflow
   stop_c=100 stop_dg=0.001
   and_stop_c=1000 and_stop_dg=0.01 and_stop_ic=0.0001
```

Just as the STOP_x parameters have AND_STOP_x partners, the KEEPGOING_x parameters have AND_KEEPGOING_x partners. The role of the KEEPGOING_x and AND_KEEPGOING_x parameters is to prevent optimization from stopping too early, even though a usual stopping criterion is met.

When PROC NETFLOW detects that it should stop for a usual stopping condition, it performs the following tests:

- It will test whether complementarity is > KEEPGOING_C (provided you have specified a KEEPGOING_C parameter), and if it is, PROC NETFLOW will perform more optimization.
- Otherwise, PROC NETFLOW will then test whether the primal-dual gap is > KEEPGOING_DG (provided you have specified a KEEPGOING_DG parameter), and if it is, PROC NETFLOW will perform more optimization.
- Otherwise, PROC NETFLOW will then test whether the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi} >$ KEEPGOING_IB (provided you have specified a KEEPGOING_IB parameter), and if it is, PROC NETFLOW will perform more optimization.
- Otherwise, PROC NETFLOW will then test whether the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci} >$ KEEPGOING_IC (provided you have specified a KEEPGOING_IC parameter), and if it is, PROC NETFLOW will perform more optimization.
- Otherwise, PROC NETFLOW will then test whether the total dual infeasibility $\sum_{i=1}^{n} infeas_{di} >$ KEEPGOING_ID (provided you have specified a KEEPGOING_ID parameter), and if it is, PROC NETFLOW will perform more optimization.
- Otherwise it will test whether complementarity is > AND_KEEPGOING_C (provided you have specified an AND_KEEPGOING_C parameter), *and* the primal-dual gap is > AND_KEEPGOING_DG (provided you have specified an AND_KEEPGOING_DG parameter), *and* the total bound infeasibility $\sum_{i=1}^{n} infeas_{bi} >$ AND_KEEPGOING_IB (provided you have specified an AND_KEEPGOING_IB parameter), *and* the total constraint infeasibility $\sum_{i=1}^{m} infeas_{ci} >$ AND_KEEPGOING_IC (provided you have specified an AND_KEEPGOING_IC parameter) *and* the total dual infeasibility $\sum_{i=1}^{n} infeas_{di} >$ AND_KEEPGOING_ID (provided you have specified an AND_KEEPGOING_ID parameter), and if it is, PROC NETFLOW will perform more optimization.

If all these tests to decide whether more optimization should be performed are false, optimization is stopped.

For example,

```
proc netflow
   stop_c=1000
   and_stop_c=2000 and_stop_dg=0.01
   and_stop_ib=1 and_stop_ic=1 and_stop_id=1
   keepgoing_c=1500
   and_keepgoing_c=2500 and_keepgoing_dg=0.05
   and_keepgoing_ib=1 and_keepgoing_ic=1 and_keepgoing_id=1
```

At the beginning of each iteration, PROC NETFLOW will stop if

- complementarity $\leq$ 1000 or
- complementarity $\leq$ 2000 and duality gap $\leq$ 0.01 and the total bound, constraint, and dual infeasibilities are each $\leq$ 1

When PROC NETFLOW determines it should stop because a usual stopping condition is met, it will stop only if

- complementarity ≤ 1500 or
- complementarity ≤ 2500 and duality gap ≤ 0.05 and the total bound, constraint, and dual infeasibilities are each ≤ 1

Interior Point: Upper Bounds

If the LP model had upper bounds ($0 \leq x \leq u$ where u is the upper bound vector), then the primal and dual problems, the duality gap, and the KKT conditions would have to be expanded.

The primal linear program to be solved is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & 0 \leq x \leq u \end{array}$$

where $0 \leq x \leq u$ is split into $x \geq 0$ and $x \leq u$. Let z be primal slack so that $x + z = u$, and associate dual variables w with these constraints. The interior point algorithm solves the system of equations to satisfy the Karush-Kuhn-Tucker (KKT) conditions for optimality:

$$\begin{aligned} Ax &= b \\ x + z &= u \\ A^T y + s - w &= c \\ x^T s &= 0 \\ z^T w &= 0 \\ x, s, z, w &\geq 0 \end{aligned}$$

These are the conditions for feasibility, with the *complementarity* conditions $x^T s = 0$ and $z^T w = 0$ added. Complementarity forces the optimal objectives of the primal and dual to be equal, $c^T x_{opt} = b^T y_{opt} - u^T w_{opt}$, as

$$0 = z_{opt}^T w_{opt} = (u - x_{opt})^T w_{opt} = u^T w_{opt} - x_{opt}^T w_{opt}$$

$$\begin{aligned} 0 &= x_{opt}^T s_{opt} = s_{opt}^T x_{opt} = (c - A^T y_{opt} + w_{opt})^T x_{opt} \\ &= c^T x_{opt} - y_{opt}^T (A x_{opt}) + w_{opt})^T x_{opt} = c^T x_{opt} - b^T y_{opt} + u^T w_{opt} \end{aligned}$$

Before the optimum is reached, a solution (x, y, s, z, w) might not satisfy the KKT conditions:

- Primal bound constraints may be violated, $infeas_b = u - x - z \neq 0$.

- Primal constraints may be violated, $infeas_c = b - Ax \neq 0$.
- Dual constraints may be violated, $infeas_d = c - A^T y - s + w \neq 0$.
- Complementarity conditions may not be satisfied, $x^T s \neq 0$ and $z^T w \neq 0$.

The calculations of the interior point algorithm can easily be derived in a fashion similar to calculations for when an LP has no upper bounds. See the paper by Lustig, Marsten, and Shanno (1992). An important point is that upper bounds can be handled by specializing the algorithm and *not* by generating the constraints $x + z = u$ and adding these to the main primal constraints $Ax = b$.

Getting Started: Linear Programming Models: Interior Point Algorithm

To solve linear programming problem using PROC NETFLOW, you save a representation of the variables and the constraints in one or two SAS data sets. These data sets are then passed to PROC NETFLOW for solution. There are various forms that a problem's data can take. You can use any one or a combination of several of these forms.

The ARCDATA= data set contains information about the variables of the problem. Although this data set is called ARCDATA, it contains data for no arcs. Instead, all data in this data set are related to variables.

The ARCDATA= data set can be used to specify information about variables, including objective function coefficients, lower and upper value bounds, and names. These data are the elements of the vectors d, m, and v in problem (NPSC). Data for a variable can be given in more than one observation.

When the data for a constrained network problem is being provided, the ARCDATA= data set always contains information necessary for arcs, their tail and head nodes, and optionally the supply and demand information of these nodes. When the data for a linear programming problem is being provided, none of this information is present, as the model has no arcs. This is the way PROC NETFLOW decides which type of problem it is to solve.

PROC NETFLOW was originally designed to solve models with networks, so an ARCDATA= data set is always expected. If an ARCDATA= data set is not specified, by default the last data set created before PROC NETFLOW is invoked is assumed to be an ARCDATA= data set. However, these characteristics of PROC NETFLOW are not helpful when a linear programming problem is being solved and all data are provided in a single data set specified by the CONDATA= data set, and that data set is not the last data set created before PROC NETFLOW starts. In this case, you must specify that an ARCDATA= data set and a CONDATA= data set are both equal to the input data set. PROC NETFLOW then knows that a linear programming problem is to be solved, and the data reside in one data set.

The CONDATA= data set describes the constraints and their right-hand sides. These data are elements of the matrix Q and the vector r.

Constraint types are also specified in the CONDATA= data set. You can include in this data set variable data such as upper bound values, lower value bounds, and objective function coefficients. It is possible to give all information about some or all variables in the CONDATA= data set.

A variable is identified in this data set by its name. If you specify a variable's name in the ARCDATA= data set, then this name is used to associate data in the CONDATA= data set with that variable.

If you use the dense constraint input format, these variable names are names of SAS variables in the VAR list of the CONDATA= data set.

If you use the sparse constraint input format, these variable names are values of the COLUMN list SAS variable of CONDATA= data set.

When using the interior point algorithm, the execution of PROC NETFLOW has two stages. In the preliminary (zeroth) stage, the data are read from the ARCDATA= data set (if used) and the CONDATA= data set. Error checking is performed. In the next stage, the linear program is preprocessed, then the optimal solution to the linear program is found. The solution is saved in the CONOUT= data set. This data set is also named in the PROC NETFLOW, RESET, and SAVE statements.

See the section "Getting Started: Network Models: Interior Point Algorithm" on page 419 for a fuller description of the stages of the interior point algorithm.

Introductory Example: Linear Programming Models: Interior Point Algorithm

Consider the linear programming problem in the section "An Introductory Example" on page 171 in the chapter on the LP procedure.

```
data dcon1;
  input _id_ $17.
        a_light a_heavy brega naphthal naphthai
        heatingo jet_1 jet_2
        _type_ $ _rhs_;
  datalines;
profit            -175 -165 -205  0  0  0 300 300 max     .
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq     0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq     0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq     0
recipe_1             0    0    0  0 .3 .7  -1   0  eq     0
recipe_2             0    0    0 .2  0 .8   0  -1  eq     0
available          110  165   80  .  .  .   .   . upperbd .
;
```

To find the minimum cost solution and to examine all or parts of the optimum, you use PRINT statements.

- **print problem/short;** outputs information for all variables and all constraint coefficients. See Figure 6.19.
- **print some_variables(j:)/short;** is information about a set of variables, (in this case, those with names that start with the character string preceding the colon). See Figure 6.20.
- **print some_cons(recipe_1)/short;** is information about a set of constraints (here, that set only has one member, the constraint called recipe_1). See Figure 6.21.
- **print con_variables(_all_,brega)/short;** lists the constraint information for a set of variables (here, that set only has one member, the variable called brega). See Figure 6.22.
- **print con_variables(recipe:,n: jet_1)/short;** coefficient information for those in a set of constraints belonging to a set of variables. See Figure 6.23.

```
proc netflow
   arcdata=dcon1
   condata=dcon1
   conout=solutn1;
run;
print problem/short;

print some_variables(j:)/short;
print some_cons(recipe_1)/short;
print con_variables(_all_,brega)/short;
print con_variables(recipe:,n: jet_1)/short;
```

The following messages, which appear on the SAS log, summarize the model as read by PROC NETFLOW and note the progress toward a solution:

```
NOTE: ARCDATA (or the last data set created if ARCDATA was not specified) and
      CONDATA are the same data set WORK.DCON1 so will assume a Linear
      Programming problem is to be solved.
NOTE: Number of variables= 8 .
NOTE: Number of <= constraints= 0 .
NOTE: Number of == constraints= 5 .
NOTE: Number of >= constraints= 0 .
NOTE: Number of constraint coefficients= 18 .
NOTE: After preprocessing, number of <= constraints= 0.
NOTE: After preprocessing, number of == constraints= 0.
NOTE: After preprocessing, number of >= constraints= 0.
NOTE: The preprocessor eliminated 5 constraints from the problem.
NOTE: The preprocessor eliminated 18 constraint coefficients from the problem.
NOTE: After preprocessing, number of variables= 0.
NOTE: The preprocessor eliminated 8 variables from the problem.
NOTE: The optimum has been determined by the Preprocessor.
NOTE: Objective= 1544.
NOTE: The data set WORK.SOLUTN1 has 8 observations and 6 variables.
```

Figure 6.19 PRINT PROBLEM/SHORT;

The NETFLOW Procedure

_N_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_
1	a_heavy	-165	165	0	0
2	a_light	-175	110	0	110
3	brega	-205	80	0	80
4	heatingo	0	99999999	0	77.3
5	jet_1	300	99999999	0	60.65
6	jet_2	300	99999999	0	63.33
7	naphthai	0	99999999	0	21.8
8	naphthal	0	99999999	0	7.45

Figure 6.19 *continued*

The NETFLOW Procedure

_N_	_id_	_type_	_rhs_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_COEF_
1	heating_o_conv	EQ	0	a_light	-175	110	0	110	0.39
2	heating_o_conv	EQ	0	a_heavy	-165	165	0	0	0.3
3	heating_o_conv	EQ	0	brega	-205	80	0	80	0.43
4	heating_o_conv	EQ	0	heatingo	0	99999999	0	77.3	-1
5	naphtha_i_conv	EQ	0	a_light	-175	110	0	110	0.1
6	naphtha_i_conv	EQ	0	a_heavy	-165	165	0	0	0.075
7	naphtha_i_conv	EQ	0	brega	-205	80	0	80	0.135
8	naphtha_i_conv	EQ	0	naphthai	0	99999999	0	21.8	-1
9	naphtha_l_conv	EQ	0	a_light	-175	110	0	110	0.035
10	naphtha_l_conv	EQ	0	a_heavy	-165	165	0	0	0.03
11	naphtha_l_conv	EQ	0	brega	-205	80	0	80	0.045
12	naphtha_l_conv	EQ	0	naphthal	0	99999999	0	7.45	-1
13	recipe_1	EQ	0	naphthai	0	99999999	0	21.8	0.3
14	recipe_1	EQ	0	heatingo	0	99999999	0	77.3	0.7
15	recipe_1	EQ	0	jet_1	300	99999999	0	60.65	-1
16	recipe_2	EQ	0	naphthal	0	99999999	0	7.45	0.2
17	recipe_2	EQ	0	heatingo	0	99999999	0	77.3	0.8
18	recipe_2	EQ	0	jet_2	300	99999999	0	63.33	-1

Figure 6.20 PRINT SOME_VARIABLES(J:)/SHORT;

The NETFLOW Procedure

_N_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_
1	jet_1	300	99999999	0	60.65
2	jet_2	300	99999999	0	63.33

Figure 6.21 PRINT SOME_CONS(RECIPE_1)/SHORT;

The NETFLOW Procedure

_N_	_id_	_type_	_rhs_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_COEF_
1	recipe_1	EQ	0	naphthai	0	99999999	0	21.8	0.3
2	recipe_1	EQ	0	heatingo	0	99999999	0	77.3	0.7
3	recipe_1	EQ	0	jet_1	300	99999999	0	60.65	-1

Figure 6.22 PRINT CON_VARIABLES(_ALL_,BREGA)/SHORT;

The NETFLOW Procedure

_N_	_id_	_type_	_rhs_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_COEF_
1	heating_o_conv	EQ	0	brega	-205	80	0	80	0.43
2	naphtha_i_conv	EQ	0	brega	-205	80	0	80	0.135
3	naphtha_l_conv	EQ	0	brega	-205	80	0	80	0.045

Figure 6.23 PRINT CON_VARIABLES(RECIPE:,N: JET_1)/SHORT;

The NETFLOW Procedure

_N_	_id_	_type_	_rhs_	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_COEF_
1	recipe_1	EQ	0	naphthai	0	99999999	0	21.8	0.3
2	recipe_1	EQ	0	jet_1	300	99999999	0	60.65	-1
3	recipe_2	EQ	0	naphthal	0	99999999	0	7.45	0.2

Unlike PROC LP, which displays the solution and other information as output, PROC NETFLOW saves the optimum in output SAS data sets you specify. For this example, the solution is saved in the SOLUTN1 data set. It can be displayed with PROC PRINT as

```
proc print data=solutn1;
   var _name_ _objfn_ _upperbd _lowerbd _value_ _fcost_;
   sum _fcost_;
   title3 'LP Optimum';
run;
```

Notice, in the CONOUT=SOLUTN1 (Figure 6.24), the optimal value through each variable in the linear program is given in the variable named _VALUE_, and the cost of value for each variable is given in the variable _FCOST_.

Figure 6.24 CONOUT=SOLUTN1

LP Optimum

Obs	_NAME_	_OBJFN_	_UPPERBD	_LOWERBD	_VALUE_	_FCOST_
1	a_heavy	-165	165	0	0.00	0
2	a_light	-175	110	0	110.00	-19250
3	brega	-205	80	0	80.00	-16400
4	heatingo	0	99999999	0	77.30	0
5	jet_1	300	99999999	0	60.65	18195
6	jet_2	300	99999999	0	63.33	18999
7	naphthai	0	99999999	0	21.80	0
8	naphthal	0	99999999	0	7.45	0
						1544

The same model can be specified in the sparse format as in the following scon2 data set. This format enables you to omit the zero coefficients.

```
data scon2;
   format _type_ $8. _col_ $8. _row_ $16. ;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
max       .             profit                .
eq        .             napha_l_conv          .
eq        .             napha_i_conv          .
eq        .             heating_oil_conv      .
eq        .             recipe_1              .
eq        .             recipe_2              .
upperbd   .             available             .
```

```
.        a_light       profit              -175
.        a_light       napha_l_conv         .035
.        a_light       napha_i_conv         .100
.        a_light       heating_oil_conv     .390
.        a_light       available             110
.        a_heavy       profit              -165
.        a_heavy       napha_l_conv         .030
.        a_heavy       napha_i_conv         .075
.        a_heavy       heating_oil_conv     .300
.        a_heavy       available             165
.        brega         profit              -205
.        brega         napha_l_conv         .045
.        brega         napha_i_conv         .135
.        brega         heating_oil_conv     .430
.        brega         available             80
.        naphthal      napha_l_conv          -1
.        naphthal      recipe_2               .2
.        naphthai      napha_i_conv          -1
.        naphthai      recipe_1               .3
.        heatingo      heating_oil_conv      -1
.        heatingo      recipe_1               .7
.        heatingo      recipe_2               .8
.        jet_1         profit               300
.        jet_1         recipe_1              -1
.        jet_2         profit               300
.        jet_2         recipe_2              -1
;
```

To find the minimum cost solution, invoke PROC NETFLOW (note the SPARSECONDATA option which must be specified) as follows:

```
proc netflow
   sparsecondata
   condata=scon2
   conout=solutn2;
run;
```

A data set that is used as an ARCDATA= data set can be initialized as follows:

```
data vars3;
   input _name_ $ profit available;
   datalines;
a_heavy  -165 165
a_light  -175 110
brega    -205  80
heatingo    0   .
jet_1     300   .
jet_2     300   .
naphthai    0   .
naphthal    0   .
;
```

The following CONDATA= data set is the original dense format CONDATA= dcon1 data set with the variable information removed. (You could have left some or all of that information in CONDATA as PROC NETFLOW "merges" data, but doing that and checking for consistency uses time.)

```
data dcon3;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2
         _type_ $ _rhs_;
   datalines;
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0  eq     0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0  eq     0
heating_o_conv    .390 .300 .430  0  0 -1   0   0  eq     0
recipe_1             0    0    0  0 .3 .7  -1   0  eq     0
recipe_2             0    0    0 .2  0 .8   0  -1  eq     0
;
```

It is important to note that it is now necessary to specify the MAXIMIZE option; otherwise, PROC NETFLOW will optimize to the minimum (which, incidentally, has a total objective = -3539.25). You must indicate that the SAS variable profit in the ARCDATA=vars3 data set has values that are objective function coefficients, by specifying the OBJFN statement. The UPPERBD must be specified as the SAS variable available that has as values upper bounds.

```
proc netflow
     maximize          /* ***** necessary ***** */
     arcdata=vars3
     condata=dcon3
     conout=solutn3;
   objfn profit;
   upperbd available;
   run;
```

The ARCDATA=vars3 data set can become more concise by noting that the model variables heatingo, naphthai, and naphthal have zero objective function coefficients (the default) and default upper bounds, so those observations need not be present.

```
data vars4;
   input _name_ $ profit available;
   datalines;
a_heavy  -165 165
a_light  -175 110
brega    -205  80
jet_1     300   .
jet_2     300   .
;
```

The CONDATA=dcon3 data set can become more concise by noting that all the constraints have the same type (eq) and zero (the default) rhs values. This model is a good candidate for using the DEFCONTYPE= option.

The DEFCONTYPE= option can be useful not only when *all* constraints have the same type as is the case here, but also when *most* constraints have the same type, or if you prefer to change the default type from $\leq$ to = or $\geq$. The essential constraint type data in CONDATA= data set is that which overrides the DEFCONTYPE= type you specified.

```
data dcon4;
   input _id_ $17.
         a_light a_heavy brega naphthal naphthai
         heatingo jet_1 jet_2;
   datalines;
naphtha_l_conv    .035 .030 .045 -1  0  0   0   0
naphtha_i_conv    .100 .075 .135  0 -1  0   0   0
heating_o_conv    .390 .300 .430  0  0 -1   0   0
recipe_1             0    0    0  0 .3 .7  -1   0
recipe_2             0    0    0 .2  0 .8   0  -1
;
proc netflow
     maximize defcontype=eq
     arcdata=vars3
     condata=dcon3
     conout=solutn3;
   objfn profit;
   upperbd available;
   run;
```

Several different ways of using an ARCDATA= data set and a sparse format CONDATA= data set for this linear program follow. The following CONDATA= data set is the result of removing the profit and available data from the original sparse format CONDATA=scon2 data set.

```
data scon5;
   format _type_ $8. _col_ $8. _row_ $16. ;
   input _type_ $ _col_ $ _row_ $ _coef_;
   datalines;
eq       .             napha_l_conv             .
eq       .             napha_i_conv             .
eq       .             heating_oil_conv         .
eq       .             recipe_1                 .
eq       .             recipe_2                 .
.        a_light       napha_l_conv          .035
.        a_light       napha_i_conv          .100
.        a_light       heating_oil_conv      .390
.        a_heavy       napha_l_conv          .030
.        a_heavy       napha_i_conv          .075
.        a_heavy       heating_oil_conv      .300
.        brega         napha_l_conv          .045
.        brega         napha_i_conv          .135
.        brega         heating_oil_conv      .430
.        naphthal      napha_l_conv            -1
.        naphthal      recipe_2                .2
.        naphthai      napha_i_conv            -1
.        naphthai      recipe_1                .3
.        heatingo      heating_oil_conv        -1
.        heatingo      recipe_1                .7
.        heatingo      recipe_2                .8
.        jet_1         recipe_1                -1
.        jet_2         recipe_2                -1
;
```

```
proc netflow
     maximize
     sparsecondata
     arcdata=vars3    /* or arcdata=vars4 */
     condata=scon5
     conout=solutn5;
   objfn profit;
   upperbd available;
   run;
```

The CONDATA=scon5 data set can become more concise by noting that all the constraints have the same type (eq) and zero (the default) rhs values. Use the DEFCONTYPE= option again. Once the first 5 observations of the CONDATA=scon5 data set are removed, the _type_ SAS variable has values that are missing in the remaining observations. Therefore, this SAS variable can be removed.

```
data scon6;
   input _col_ $ _row_&$16. _coef_;
   datalines;
a_light  napha_l_conv          .035
a_light  napha_i_conv          .100
a_light  heating_oil_conv      .390
a_heavy  napha_l_conv          .030
a_heavy  napha_i_conv          .075
a_heavy  heating_oil_conv      .300
brega    napha_l_conv          .045
brega    napha_i_conv          .135
brega    heating_oil_conv      .430
naphthal napha_l_conv           -1
naphthal recipe_2                .2
naphthai napha_i_conv           -1
naphthai recipe_1                .3
heatingo heating_oil_conv       -1
heatingo recipe_1                .7
heatingo recipe_2                .8
jet_1    recipe_1               -1
jet_2    recipe_2               -1
;
proc netflow
     maximize
     defcontype=eq
     sparsecondata
     arcdata=vars3     /* or arcdata=vars4 */
     condata=scon6
     conout=solutn6;
   objfn profit;
   upperbd available;
   run;
```

Interactivity: Linear Programming Models: Interior Point algorithm

PROC NETFLOW can be used interactively. You begin by giving the PROC NETFLOW statement, and you must specify the CONDATA= data set. If necessary, specify the ARCDATA= data set.

The variable lists should be given next. If you have variables in the input data sets that have special names (for example, a variable in the ARCDATA= data set named _COST_ that has objective function coefficients as values), it may not be necessary to have many or any variable lists.

The PRINT, QUIT, SAVE, SHOW, RESET, and RUN statements follow and can be listed in any order. The QUIT statements can be used only once. The others can be used as many times as needed.

The CONOPT and PIVOT are not relevant to the interior point algorithm and should not be used.

Use the RESET or SAVE statement to change the name of the output data set. There is only one output data set, the CONOUT= data set. With the RESET statement, you can also indicate the reasons why optimization should stop, (for example, you can indicate the maximum number of iterations that can be performed). PROC NETFLOW then has a chance to either execute the next statement or, if the next statement is one that PROC NETFLOW does not recognize (the next PROC or DATA step in the SAS session), do any allowed optimization and finish. If no new statement has been submitted, you are prompted for one. Some options of the RESET statement enable you to control aspects of the interior point algorithm. Specifying certain values for these options can reduce the time it takes to solve a problem. Note that any of the RESET options can be specified in the PROC NETFLOW statement.

The RUN statement starts optimization. Once the optimization has started, it runs until the optimum is reached. The RUN statement should be specified at most once.

The QUIT statement immediately stops PROC NETFLOW. The SAVE statement has options that enable you to name the output data set; information about the current solution is saved in this output data set. Use the SHOW statement if you want to examine the values of options of other statements. Information about the amount of optimization that has been done and the STATUS of the current solution can also be displayed using the SHOW statement.

The PRINT statement instructs PROC NETFLOW to display parts of the problem. The ways the PRINT statements are specified are identical whether the interior point algorithm or the simplex algorithm is used; however, there are minor differences in what is displayed for each variable or constraint coefficient.

PRINT VARIABLES produces information on all arcs. PRINT SOME_VARIABLES limits this output to a subset of variables. There are similar PRINT statements for constraints:

```
PRINT CONSTRAINTS;
PRINT SOME_CONS;
```

PRINT CON_VARIABLES enables you to limit constraint information that is obtained to members of a set of variables that have nonzero constraint coefficients in a set of constraints.

For example, an interactive PROC NETFLOW run might look something like this:

```
proc netflow
       condata=data set
       other options;
   variable list specifications;     /* if necessary */
   reset options;
```

```
   print options;     /* look at problem              */
run;                  /* do some optimization         */
   print options;     /* look at the optimal solution */
   save options;      /* keep optimal solution        */
```

If you are interested only in finding the optimal solution, have used SAS variables that have special names in the input data sets, and want to use default setting for everything, then the following statement is all you need:

proc netflow condata= *data set* **;**

Functional Summary: Linear Programming Models: Interior Point Algorithm

The following table outlines the options available for the NETFLOW procedure when the interior point algorithm is being used to solve a linear programming problem, classified by function.

Table 6.13 Functional Summary, Linear Programming Models

Description	Statement	Option
Input Data Set Options:		
Arcs input data set	PROC NETFLOW	ARCDATA=
Constraint input data set	PROC NETFLOW	CONDATA=
Output Data Set Option:		
Solution data set	PROC NETFLOW	CONOUT=
Data Set Read Options:		
CONDATA has sparse data format	PROC NETFLOW	SPARSECONDATA
Default constraint type	PROC NETFLOW	DEFCONTYPE=
Special COLUMN variable value	PROC NETFLOW	TYPEOBS=
Special COLUMN variable value	PROC NETFLOW	RHSOBS=
Data for a constraint found once in CONDATA	PROC NETFLOW	CON_SINGLE_OBS
Data for a coefficient found once in CONDATA	PROC NETFLOW	NON_REPLIC=
Data are grouped, exploited during data read	PROC NETFLOW	GROUPED=
Problem Size Specification Options:		
Approximate number of variables	PROC NETFLOW	NNAS=
Approximate number of coefficients	PROC NETFLOW	NCOEFS=
Approximate number of constraints	PROC NETFLOW	NCONS=
Network Options:		
Default variable objective function coefficient	PROC NETFLOW	DEFCOST=
Default variable upper bound	PROC NETFLOW	DEFCAPACITY=
Default variable lower bound	PROC NETFLOW	DEFMINFLOW=
Memory Control Options:		
Issue memory usage messages to SAS log	PROC NETFLOW	MEMREP

Description	Statement	Option
Number of bytes to use for main memory	PROC NETFLOW	BYTES=
Proportion of memory for arrays	PROC NETFLOW	COREFACTOR=
maximum bytes for a single array	PROC NETFLOW	MAXARRAYBYTES=
Interior Point Algorithm Options:		
Use interior point algorithm	PROC NETFLOW	INTPOINT
Factorization method	RESET	FACT_METHOD=
Allowed amount of dual infeasibility	RESET	TOLDINF=
Allowed amount of primal infeasibility	RESET	TOLPINF=
Allowed total amount of dual infeasibility	RESET	TOLTOTDINF=
Allowed total amount of primal infeasibility	RESET	TOLTOTPINF=
Cut-off tolerance for Cholesky factorization	RESET	CHOLTINYTOL=
Density threshold for Cholesky processing	RESET	DENSETHR=
Step-length multiplier	RESET	PDSTEPMULT=
Preprocessing type	RESET	PRSLTYPE=
Print optimization progress on SAS log	RESET	PRINTLEVEL2=
Write optimization time to SAS log	RESET	OPTIM_TIMER
Interior Point Stopping Criteria Options:		
Maximum number of interior point iterations	RESET	MAXITERB=
Primal-dual (duality) gap tolerance	RESET	PDGAPTOL=
Stop because of complementarity	RESET	STOP_C=
Stop because of duality gap	RESET	STOP_DG=
Stop because of $infeas_b$	RESET	STOP_IB=
Stop because of $infeas_c$	RESET	STOP_IC=
Stop because of $infeas_d$	RESET	STOP_ID=
Stop because of complementarity	RESET	AND_STOP_C=
Stop because of duality gap	RESET	AND_STOP_DG=
Stop because of $infeas_b$	RESET	AND_STOP_IB=
Stop because of $infeas_c$	RESET	AND_STOP_IC=
Stop because of $infeas_d$	RESET	AND_STOP_ID=
Stop because of complementarity	RESET	KEEPGOING_C=
Stop because of duality gap	RESET	KEEPGOING_DG=
Stop because of $infeas_b$	RESET	KEEPGOING_IB=
Stop because of $infeas_c$	RESET	KEEPGOING_IC=
Stop because of $infeas_d$	RESET	KEEPGOING_ID=
Stop because of complementarity	RESET	AND_KEEPGOING_C=
Stop because of duality gap	RESET	AND_KEEPGOING_DG=
Stop because of $infeas_b$	RESET	AND_KEEPGOING_IB=
Stop because of $infeas_c$	RESET	AND_KEEPGOING_IC=
Stop because of $infeas_d$	RESET	AND_KEEPGOING_ID=
PRINT Statement Options:		
Display everything	PRINT	PROBLEM
Display variable information	PRINT	VARIABLES

Description	Statement	Option
Display constraint information	PRINT	CONSTRAINTS
Display information for some variables	PRINT	SOME_VARIABLES
Display information for some constraints	PRINT	SOME_CONS
Display information for some constraints associated with some variables	PRINT	CON_VARIABLES
PRINT Statement Qualifiers:		
Produce a short report	PRINT	/ SHORT
Produce a long report	PRINT	/ LONG
Display arcs/variables with zero flow/value	PRINT	/ ZERO
Display arcs/variables with nonzero flow/value	PRINT	/ NONZERO
SHOW Statement Options:		
Show problem, optimization status	SHOW	STATUS
Show LP model parameters	SHOW	NETSTMT
Show data sets that have been or will be created	SHOW	DATASETS
Miscellaneous Options:		
Infinity value	PROC NETFLOW	INFINITY=
Scale constraint row, variable column coefficients, or both	PROC NETFLOW	SCALE=
Maximization instead of minimization	PROC NETFLOW	MAXIMIZE

Generalized Networks: NETFLOW Procedure

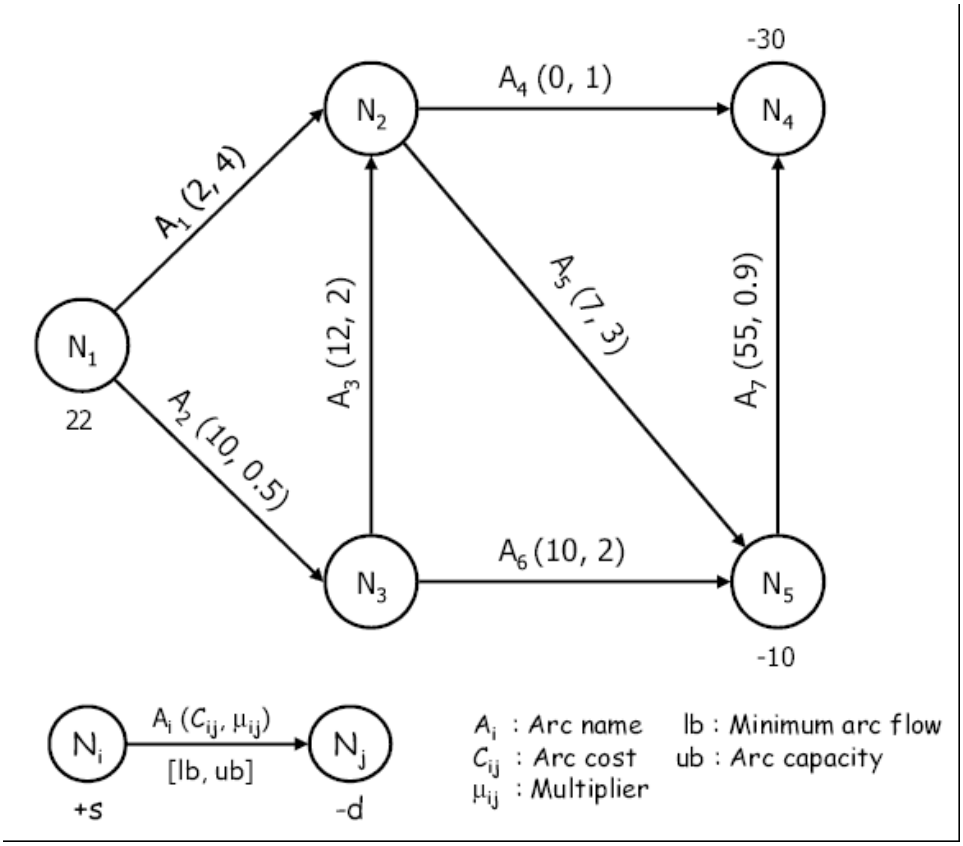
In this section we introduce how to use the NETFLOW procedure to solve *generalized* network programming problems.

What Is a Generalized Network?

It is well known that in a pure network the sum of flows entering an arc is equal to the sum of flows leaving it. However, in a generalized network there may be a gain or a loss as flow traverses an arc. Each arc has a multiplier to represent these gains or losses.

To illustrate what is meant, consider the network shown in Figure 6.25.

Figure 6.25 Generalized Network Example



You can think of this as a network representing a supply node (N_1), trans-shipment nodes (N_2, N_3), and demand nodes (N_4, N_5). As indicated by the legend, the number below a node represents its supdem value. Above each arc is its name, followed by the arc cost and arc multiplier in parentheses. The lower and upper bounds on flow allowed to **enter** an arc are represented in square brackets below it. When no bounds are specified (as in Figure 6.25), they are assumed to be [0, 99999999].

Now consider the node pair (N_1, N_2). The information on arc A_1 says that it costs 2 per unit of flow to traverse it, and for each unit of flow entering the arc, four units get accumulated at node N_2. The corresponding component in the objective function is two times the flow through arc A_1 leaving node N_1, not two times the flow through arc A_1 arriving at node N_2.

A commonly encountered example of a generalized network is in power generation: as electricity is transmitted over wires, there is some unavoidable loss along the way. This loss is represented by a multiplier less than 1.0.

Arc multipliers need not always be less than 1.0. For instance, in financial models, a flow through an arc could represent money in a bank account earning interest. In that case, the arc would have a multiplier greater than 1.0.

Generalized networks offer convenience when flow commodity changes. For a pure network, care must be taken to ensure the flow commodity is the same throughout the entire model. For example, in a model to determine how sugar should be grown, refined, packaged, and sold, the flow commodity might be kilograms of sugar, and all numerical parameters throughout the model (all supplies, arc costs, capacities, bounds, demands, etc.) must be in terms of kilograms of sugar. Some arcs might correspond to the movement of

5-kilogram bags of sugar. If a generalized network formulation is used, the arc that represents packaging could be given a multiplier of 0.2, so flow through arcs that convey flow corresponding to bags of sugar will have arc costs in terms of dollars per bag, and capacities, bounds, demands, etc. in terms of number of bags.

In the following sections we describe in detail how to provide data for arc multipliers, how to deal with excess supply or demand in pure and generalized networks, how to model maximum flow problems, and how to handle networks with missing supply and demand nodes, and ranges on supply and demand.

How to Specify Data for Arc Multipliers

If you are familiar with using the NETFLOW procedure to solve pure network problems, then solving generalized network problems is fairly simple. You just need to provide the additional data for the arc multipliers. Arcs by default have a multiplier of 1.0, so you only need to provide arc multipliers that are not equal to 1.0. You can specify the arc multiplier data in either or both of the ARCDATA= and CONDATA= data sets. The procedure scans the SAS variables in the ARCDATA= data set, and if it finds a name _MULT_ (or a similar name with letters of different case), then it assumes that the SAS variable contains data for arc multipliers. CONDATA= is scanned for special type values that indicates data are arc multipliers.

The rest of this section describes the various ways in which you can specify data for the arc multipliers. The network in Figure 6.25 is used for illustration.

All Arc Multiplier Data in the ARCDATA= Data Set

You can specify all the arc multiplier data in the ARCDATA= data set. The following code creates the input SAS data sets:

```
/*************************************************************
 *                                                           *
 * Generalized Networks:                                     *
 * How to Specify Data for Arc Multipliers                   *
 *                                                           *
 *************************************************************/

/*  All Arc Multiplier Data in the ARCDATA= Data Set  */

data nodes;
   input _node_ $ _sd_ ;
datalines;
N1  22
N4 -30
N5 -10
;

data arcs;
   input _from_ $ _to_ $  _cost_ _mult_;
datalines;
N1 N2   2   4
N1 N3  10 0.5
N2 N4   0   1
N2 N5   7   3
N3 N2  12   2
```

```
N3 N5  10   2
N5 N4  55 0.9
;
```

Let us first look at the data for this problem. There is a variable named _mult_ in the ARCDATA= data set, so PROC NETFLOW assumes it represents the arc multipliers. The SAS variable _sd_ represents the supdem value of a node. A positive or missing S value indicates supply, and a negative or missing D value indicates demand.

The optimal solution can be obtained from the CONOUT= data set. Note that you need to specify the CONOUT= data set even if the network has no side constraints; you cannot use the ARCOUT= data set.

You can use the following SAS code to run PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   bytes    = 100000
   nodedata = nodes
   arcdata  = arcs
   conout   = solution;
run;
```

The optimal solution is displayed in Output 6.26.

Figure 6.26 Output of the Example Problem

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	N1	N2	2	99999999	0	4.0	22	.	6.0000	12.000
2	N3	N2	12	99999999	0	2.0	.	.	3.0000	36.000
3	N1	N3	10	99999999	0	0.5	22	.	16.0000	160.000
4	N2	N4	0	99999999	0	1.0	.	30	30.0000	0.000
5	N5	N4	55	99999999	0	0.9	.	30	-0.0000	-0.000
6	N2	N5	7	99999999	0	3.0	.	10	0.0000	0.000
7	N3	N5	10	99999999	0	2.0	.	10	5.0000	50.000

All Arc Multiplier Data in CONDATA= Data Set

Let us now solve the same problem, but with all the arc multipliers specified in the CONDATA= data set. The CONDATA= data set can have either a sparse format or a dense format. The following code illustrates the dense format representation:

```
data arcs1b;
   input _from_ $ _to_ $  _cost_;
datalines;
N1 N2   2
N1 N3  10
N2 N4   0
N2 N5   7
N3 N2  12
N3 N5  10
N5 N4  55
;
```

```
data MUdense;
   input _type_ $ N1_N2 N1_N3 N2_N4 N2_N5 N3_N2 N3_N5 N5_N4;
datalines;
mult 4.0  0.5  1.0  0.3  2.0  2.0  0.9
;
```

You can use the following SAS code to obtain the solution:

```
proc netflow
   gennet
   nodedata = nodes
   arcdata  = arcs1b
   condata  = MUdense
   conout   = soln1b;
run;
```

Note that a new option, GENNET, has been specified in the call to PROC NETFLOW. This option is necessary when the network is generalized and there are no arc multiplier data in the ARCDATA= data set. If this option is not specified, then the procedure assumes that the network is pure (without arc multipliers) and sets up the excess supply node and the excess arcs.

The sparse format representation is as follows:

```
data MUsparse;
   input _type_ $ _col_ $ _coef_;
datalines;
mult N1_N2 4.0
mult N1_N3 0.5
mult N2_N4 1.0
mult N2_N5 0.3
mult N3_N2 2.0
mult N3_N5 2.0
mult N5_N4 0.9
;
```

You can use the following SAS code to obtain the solution:

```
proc netflow
   gennet   sparsecondata
   nodedata = nodes
   arcdata  = arcs1b
   condata  = MUsparse
   conout   = soln1c;
run;
```

Note that you need to specify the SPARSECONDATA option in the call to PROC NETFLOW.

Arc Multiplier Data in Both ARCDATA= and CONDATA= Data Sets

You can also provide some multiplier data in the ARCDATA= data set, and the rest in the CONDATA= data set as follows:

```
data arcs1c;
   input _from_ $ _to_ $  _cost_ _mult_;
datalines;
N1 N2   2   4
N1 N3  10  .5
N2 N4   0   .
N2 N5   7   .
N3 N2  12   .
N3 N5  10   .
N5 N4  55   .
;
data MUdense1;
   input _type_ $ N2_N4 N2_N5 N3_N2 N3_N5 N5_N4;
datalines;
mult 1.0  0.3  2.0  2.0  0.9
;
```

The procedure merges the data when all the input data sets have been read.

Specifying Arc Multiplier Data Using a List Statement

You can also specify the name of the multiplier variable in the list statement MULT, or MULTIPLIER. For example, if the name of the variable is lossrate, then use the following:

```
proc netflow
...
;
mult lossrate;
run;
```

You may also use MULT, GAIN, or LOSS (or similar names with letters of different case) as a value of the TYPE list SAS variable.

Using the New EXCESS= Option in Pure Networks: NETFLOW Procedure

In this section we describe how to use the new EXCESS= option to solve a wide variety of problems. These include the following:

- networks with excess supply or demand
- networks containing nodes with unknown supply and demand values
- maximum flow problems
- networks with nodes having supply and demand ranges

Handling Excess Supply or Demand

The supdem value of a node can be specified in the following formats:

- in the NODEDATA= data set, using the _supdem_ or _sd_ list variable
- in the ARCDATA= data set, using the _SUPPLY_ and _DEMAND_ list variables

If there is only one supply (demand) node, then use the SOURCE= (SINK=) option to refer to it, and use the SUPPLY= (DEMAND=) option to specify its supdem value.

To ensure feasibility, there are different methods by which flow can be added to or drained from the network. This extra flow can be added to or drained from the network at either the supply or demand nodes. The new EXCESS= option is used to address such instances.

For pure networks there are two valid values that can be specified for the EXCESS= option: EXCESS=ARCS and EXCESS=SLACKS.

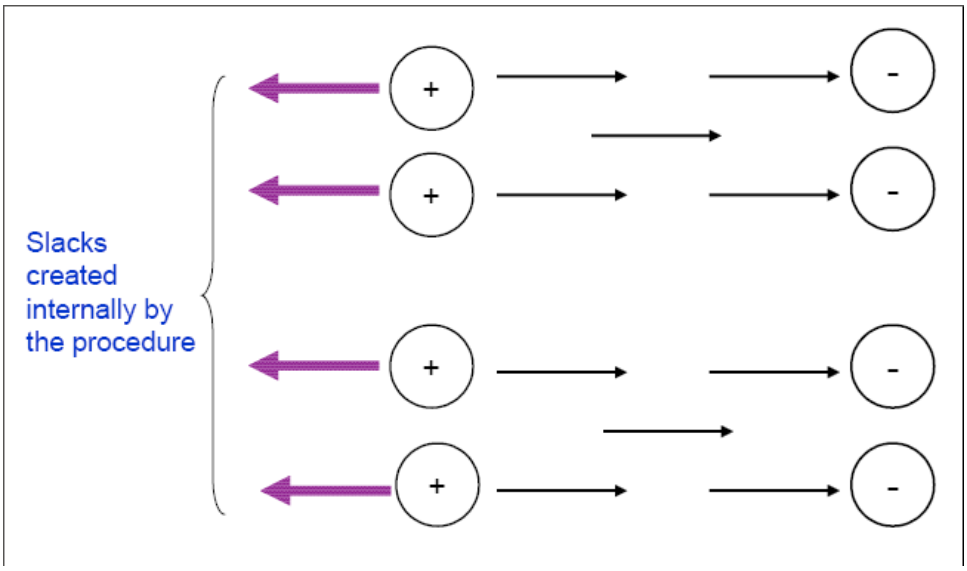
EXCESS=ARCS is the default value. An extra node, referred to as _EXCESS_, is added to the network and is connected to the actual network by "excess" arcs.

- If total supply exceeds total demand, then _EXCESS_ is an extra demand node with demand equal to total supply minus total demand.
  - If the THRUNET option is specified, the "excess" arcs are directed away from any actual demand node (even nodes with missing D demand) and toward _EXCESS_.
  - Or else if there are demand nodes with missing D demands, the "excess" arcs are directed away from these nodes and toward _EXCESS_.
  - Or else the "excess" arcs are directed away from the supply nodes and toward _EXCESS_.
- If the total demand exceeds the total supply, then _EXCESS_ is an extra supply node with supply equal to the total demand minus the total supply.
  - If the THRUNET option is specified, the "excess" arcs are directed away from _EXCESS_ and toward any actual supply node (even nodes with missing S supply.)
  - Or else if there are supply nodes with missing S supplies, the "excess" arcs are directed away from _EXCESS_ and toward these nodes.
  - Or else the "excess" arcs are directed away from _EXCESS_ and toward the demand nodes.

The node _EXCESS_ and the associated arcs are created to ensure that the problem presented to the optimizer has a total supply equal to the total demand. They are neither displayed in the optimal solution nor saved in any output SAS data set.

If EXCESS=SLACKS is specified, then slack variables are created for some flow conservation constraints instead of having the node _EXCESS_ and "excess" arcs. The flow conservation constraint (which was an inequality) is now converted to an equality with the addition of the slack variable. Alternatively, you can think of these slacks as arcs with one of their end nodes missing — they are directed from a node but not toward any other node (or directed toward a node but not from any other node). Figure 6.27 presents a clearer picture of this.

Figure 6.27 EXCESS=SLACKS, Total Supply Exceeds Total Demand, THRUNET Not Specified, No Nodes with Missing Demand



NOTE: When you specify EXCESS=SLACKS, the interior point solver is used. The output SAS data set needs to be specified by the CONOUT= data set, even if side constraints are not present. Also, when you specify the EXCESS=SLACKS option, the size of the model submitted to the optimizer is smaller than with EXCESS=ARCS since it no longer has the _EXCESS_ node and the excess arcs associated with it.

Handling Missing Supply and Demand Simultaneously

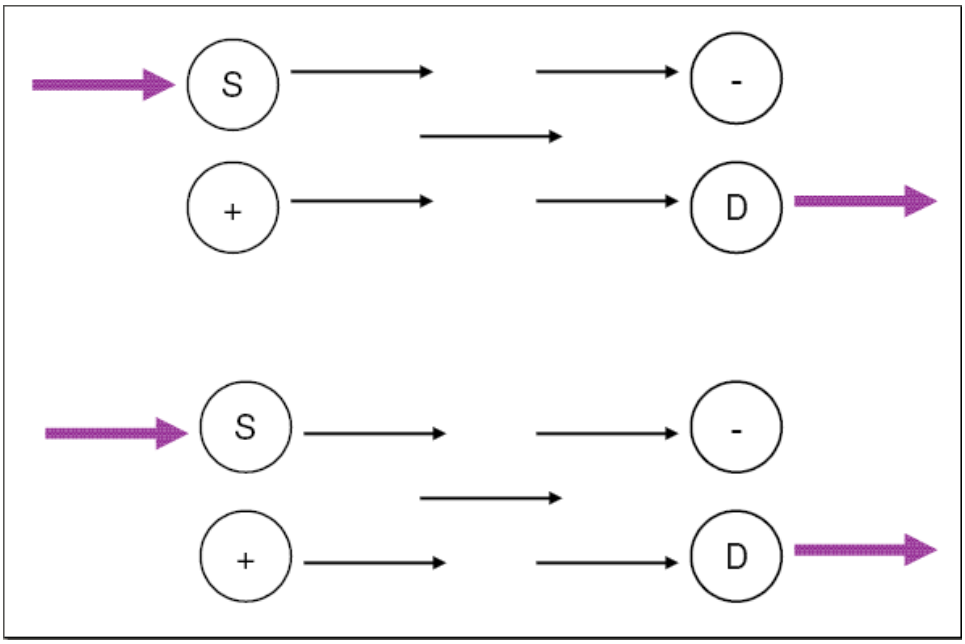
Another new feature in the NETFLOW procedure is that it enables you to specify a network containing both nodes with missing S supply values and nodes with missing D demand values. This feature is a powerful modeling tool, and we show in the later sections how to use it to formulate and solve maximum flow problems and network models with range constraints on supply and demand.

Whenever a network is detected to have both nodes with missing S supply values and nodes with missing D demand values, a special value of the EXCESS= option is assigned internally by the procedure; any value you specify for the EXCESS= option is overridden. The procedure solves the problem in the following manner:

- Nodes with positive (negative) supdem values supply (demand) the exact amount of flow specified.
- Nodes with missing S supply (missing D demand) values supply (demand) flow quantities that are determined by optimization.

Figure 6.28 displays how the slack variables are set up by the procedure internally. These variables are neither a part of the input data set nor displayed in any output SAS data set or printed output.

Figure 6.28 A Network with Both Missing S Supply and Missing D Demand Nodes



Maximum Flow Problems

The maximum flow problem (MFP) can be stated as follows: Given a directed graph $G = (N, A)$ with capacity $u_{ij} \geq 0$ on each arc $(i, j) \in A$, a source node s and a sink node t, find the maximum flow that can go from s to t, while obeying the flow conservation constraints at each node. You can solve such problems using the MAXFLOW option in the call to PROC NETFLOW.

Ordinarily many, if not all, arcs in an MFP network have capacities, and it is common that these arcs have zero costs. However, the NETFLOW procedure enables you to have nonzero costs to influence the optimal solution in cases where multiple maximum flow patterns are known to exist.

The following two subsections explain the role of the EXCESS= option in solving pure and generalized maximum flow problems.

The EXCESS=ARCS Option

Consider a maximum flow problem involving a pure network. Assume that you do not explicitly specify the EXCESS= option (the EXCESS=ARCS option is used by the procedure by default). The NETFLOW procedure sets up the problem in the following manner:

1. The source node is assigned a supdem value equal to INFINITY−1.
2. The sink node is assigned a supdem value equal to −(INFINITY−1).

3. If there is no existing arc between the source node and the sink node, an arc called the *bypass arc* directed from the source node to the sink node is added.

4. If there is an existing arc between the source node and the sink node, a dummy node is used to break up what would have been a single bypass arc: *source* —> *sink* gets transformed into two arcs, *source* —> *dummy* —> *sink*.

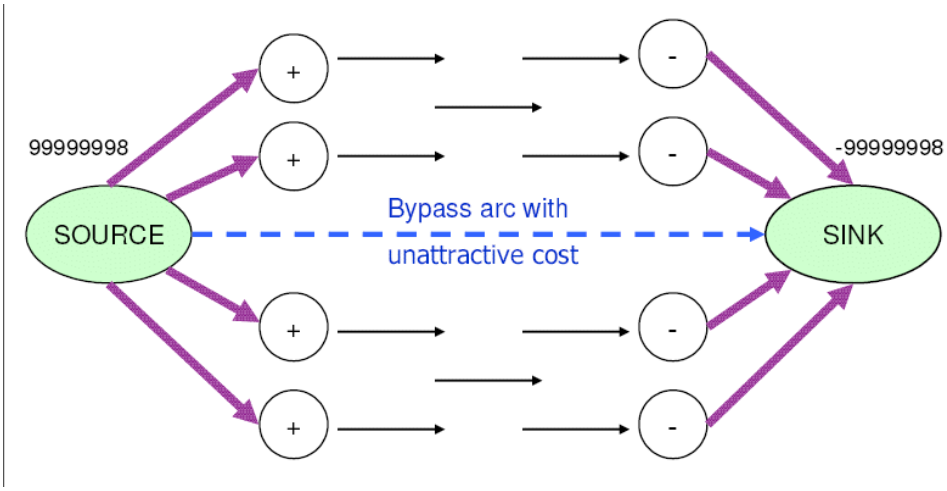
5. If you are maximizing, then the cost of the bypass arc(s) is equal to −1 if all other arcs have zero costs; otherwise the cost of the bypass arc(s) is equal to −(INFINITY / BYPASSDIV).

6. If you are minimizing, then the cost of the bypass arc(s) is equal to 1 if all other arcs have zero costs; otherwise the cost of the bypass arc(s) is equal to INFINITY / BYPASSDIV.

You can specify the value of the INFINITY= option in the procedure statement, or you can use the default value of 99999999. You can also specify the BYPASSDIV= option. The default value for the BYPASSDIV= option is 100.

This scenario is depicted in Figure 6.29. Since the cost of the bypass arc is unattractive, the optimization process minimizes the flow through it, thereby maximizing the flow through the real network. See the first subsection in Example 6.10 for an illustration.

Figure 6.29 Pure Maximum Flow Problem, EXCESS=ARCS Option Specified



This method of setting up a maximum flow problem does come with a drawback. It is likely to produce incorrect results if the following occur:

- the maximum flow is greater than INFINITY−1, or
- the cost of the bypass arc is insufficiently unattractive to ensure that the entire flow traverses the real network and not through the bypass arc

Additionally, numbers of large magnitude can cause problems during optimization, including numerical instability and loss of precision. In the next section, we explain how to overcome these difficulties when solving maximum flow problems.

The EXCESS=SLACKS Option

Assume you have a pure maximum flow problem and you specify the EXCESS=SLACKS option. The NETFLOW procedure sets up the problem in the following manner:

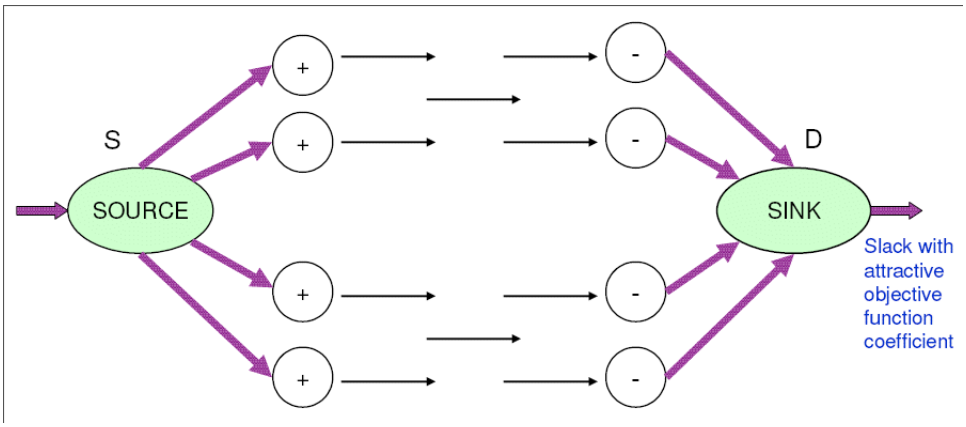
- The source node is assigned a missing S supply value.
- The sink node is assigned a missing D demand value.

Since this network contains a node with a missing S supply value and a node with a missing D demand value, we have a situation similar to the one described in the section "Handling Missing Supply and Demand Simultaneously" on page 456. Both of these nodes have slack variables. Usually, slack variables have zero objective function coefficients, but because the MAXFLOW option is specified, one of the slack variables must be attractive enough to make it worthwhile for flow to traverse the network. Figure 6.30 presents the scenario clearly.

If you are maximizing, then the objective function coefficient of the slack variable associated with the sink node is -1 if all other arcs have zero costs. Otherwise it is $-$(INFINITY / BYPASSDIV). If you are minimizing, then the objective function coefficient of the slack variable associated with the sink node is 1 if all arcs have zero costs. Otherwise it is INFINITY / BYPASSDIV. See the second subsection in Example 6.10 for an illustration of the EXCESS=SLACKS option in pure maximum flow problems.

NOTE: If the MAXFLOW option is not specified, these slack variables assume zero objective function coefficients, and the MFP may not be solved properly.

Figure 6.30 Pure Maximum Flow Problem with EXCESS=SLACKS Option Specified

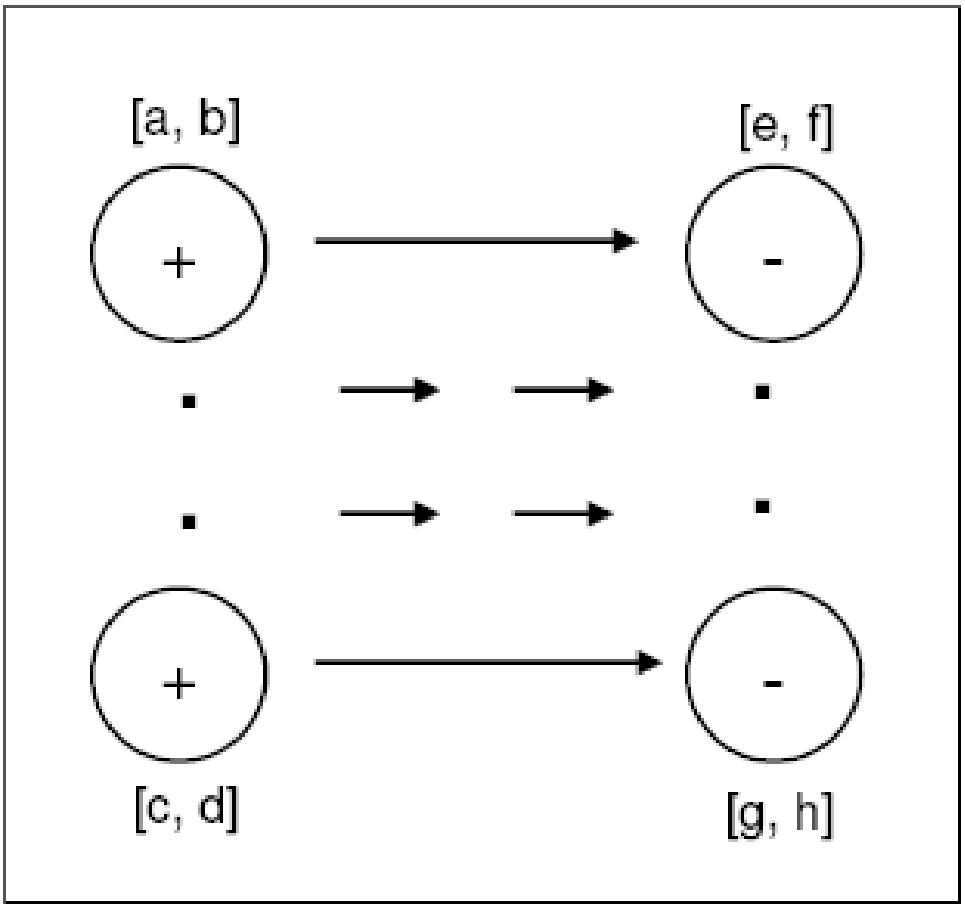


When you use the MAXFLOW option, the procedure sets up the problem in such a way that maximum flow traverses the network. This enables you to transform certain types of problems into maximum flow problems. One such instance is when you have a network where the amount of flow that is supplied or demanded falls within a range of values. The following section describes how to solve such problems.

Handling Supply and Demand Ranges

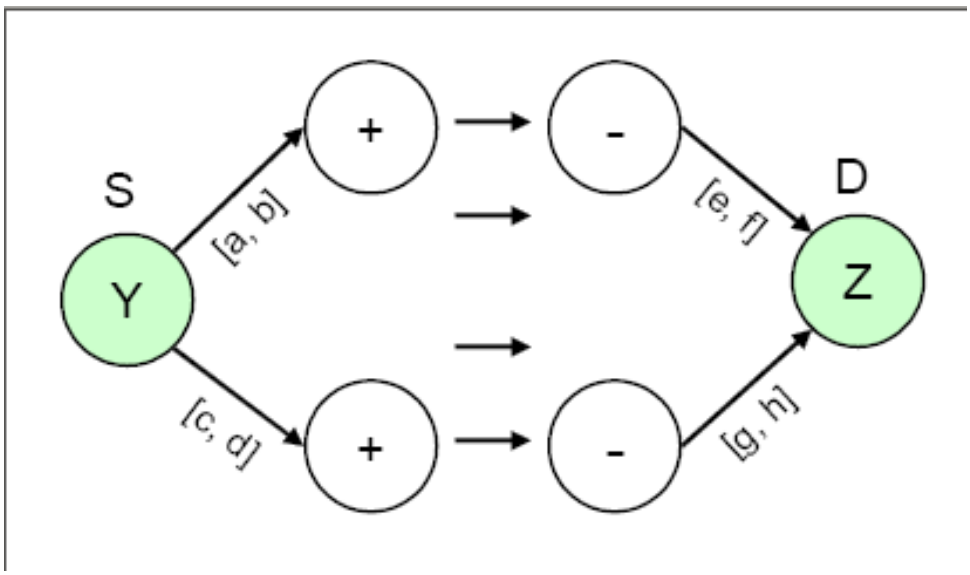
Consider the scenario depicted by Figure 6.31, where the supply and demand nodes have ranges; i.e., the amounts they can supply or demand are constrained to be within certain lower and upper bounds.

Figure 6.31 Network with Ranges on Supplies and Demands



To model this situation, you first need to add a supply node with missing S supply value (the Y node in Figure 6.32) and a demand node with missing D demand value (the Z node in Figure 6.32). The bounds on the supply and demand nodes get transformed into upper/lower bounds on the arcs that connect them to nodes Y and Z, respectively. It might be necessary to have costs for these arcs to make it worthwhile for flow to traverse them, and subsequently to traverse the actual network. In practice, these costs represent procurement costs, profit from sales, etc.

Figure 6.32 Network with Ranges of Supplies and Demands: Transformed Model



You could set up all your network models in this fashion, not only in scenarios in which there are supply and demand ranges. For instance, this modeling technique could be used under the following conditions:

- if there are no ranges
- if the network is generalized, and you do not know whether to specify EXCESS=SUPPLY or EXCESS=DEMAND
- if some of the lower bounds are zero or some capacities are infinite, in which case you simply do not specify the capacity

Using the New EXCESS= Option in Generalized Networks: NETFLOW Procedure

In this section we briefly describe how to use the new EXCESS= option in generalized networks. We provide simple scenarios to enable you to understand what happens internally in the solver when different values for the EXCESS= option are specified.

Total Supply and Total Demand: How Generalized Networks Differ from Pure Networks

For a pure network, it is easy to check for excess supply or excess demand. If the sum of positive supdem values exceeds (is less than) the absolute value of the sum of negative supdem values, then the network has excess supply (demand).

However, in a generalized network you need to specify whether the network should have excess supply or excess demand. To do that you can specify the option EXCESS=SUPPLY or EXCESS=DEMAND, respectively.

Although the total supply and total demand of a generalized network can be determined, you may not know beforehand if excess flow must be added to, removed from, or left unused by the network. For example, consider a simple network, one consisting of two nodes, A and B, connected by a single arc, A —> B. Suppose the supply of node A is 10 and the demand of node B is 30. If this is a pure network, then the network solved must be either _EXCESS_ —> A —> B if the THRUNET option is not specified and the flow through the arc between A and B is 30 units, or A —> B <— _EXCESS_ if the THRUNET option is specified and the flow through the arc from A to B is 10 units. _EXCESS_ is the name of an extra node that is set up by the procedure behind the scenes, and in both cases it would have a supply capacity of 20 units, which is the flow through the excess arc. However, if the network is generalized, and the arc from A to B has a multiplier of 3.0, then the flow through the arc from A to B would be 10 units, and the network would be feasible without any excess node and arcs. Indeed, no excess node and arcs would be created, even though total supply and total demand are unequal. Therefore, once the NETFLOW procedure detects that the network has arc multipliers that are not 1.0, it might not set up the excess node and the excess arcs.

In Example 6.11 we illustrate the use of the EXCESS= option to solve generalized networks that have total supply equal to total demand, but have arcs with varying multipliers.

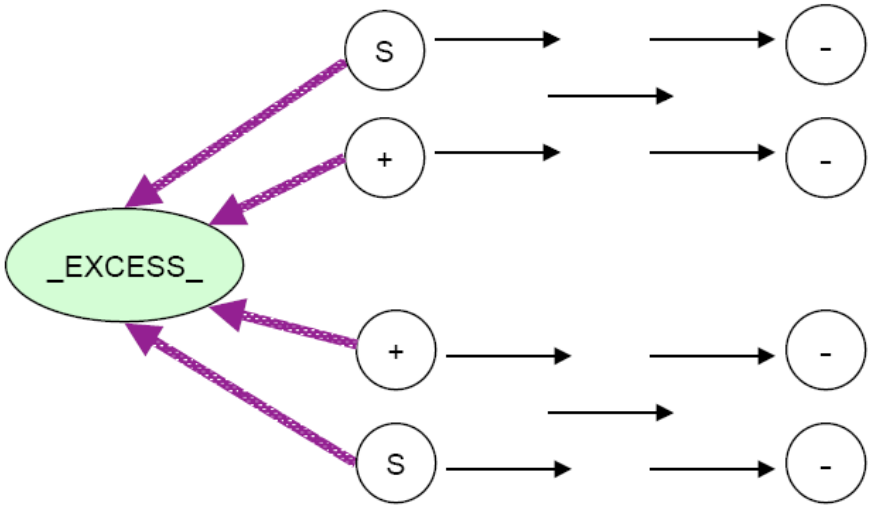
In the section "Handling Missing Supply and Demand Simultaneously" on page 456, we discuss the case where a network has both nodes with missing S supply values and nodes with missing D demand values. In the next two subsections we analyze scenarios where a network has nodes with either missing S supply values or missing D demand values, but not both.

The EXCESS=SUPPLY Option

If you specify the EXCESS=SUPPLY option, then there are three possible scenarios to deal with:

Case 1: No Node with Missing D Demand, THRUNET Not Specified (see Figure 6.33)
Drain the excess supply from all supply nodes.

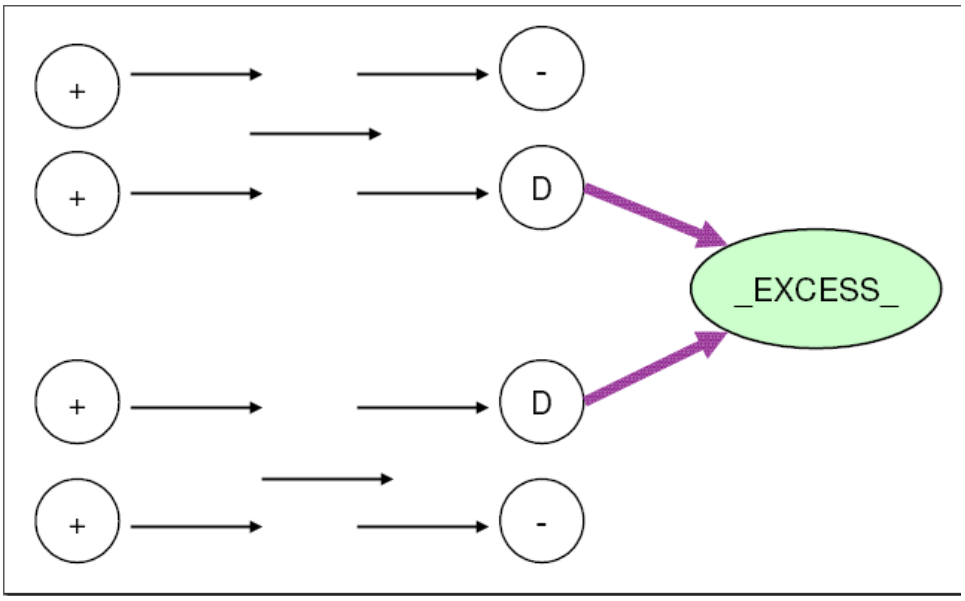
Figure 6.33 Nodes with Missing S Supply, THRUNET Specified



Case 2: Some Nodes with Missing D Demand, THRUNET Not Specified (see Figure 6.34)

Drain the excess supply from nodes that have missing D demand values. If a node has a missing D demand value, then the amount it demands is determined by optimization. For a demand node with negative supdem value, that value negated is equal to the sum of flows on all actual arcs directed toward that node.

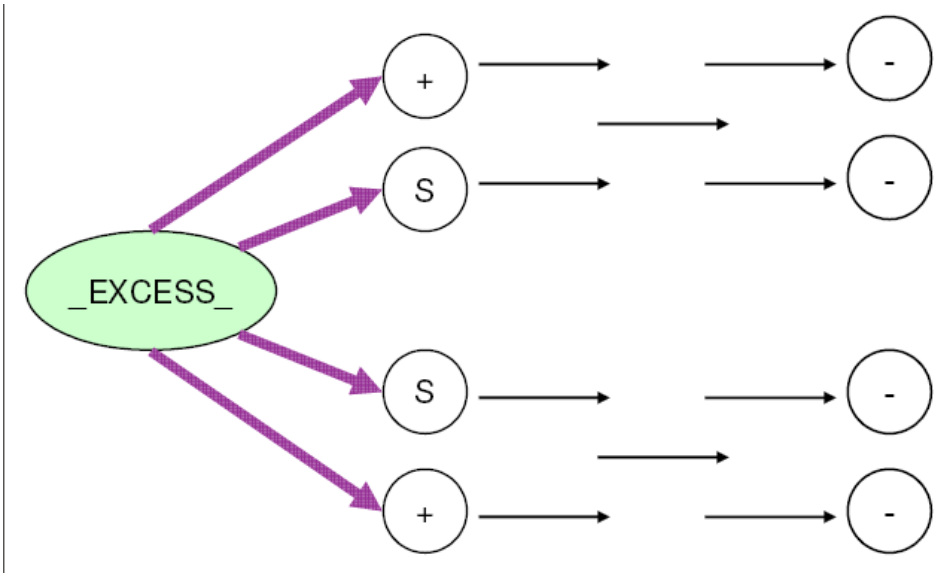
Figure 6.34 Nodes with Missing D Demand



Case 3: THRUNET Specified (see Figure 6.35)

Drain the excess supply from all demand nodes. If a node has a negative supdem value, that value negated is the lower bound on the sum of flows on all actual arcs directed toward that node. If a node has a missing D demand value, then the amount it demands is determined by optimization.

Figure 6.35 Nodes with Missing D Demand, THRUNET Specified

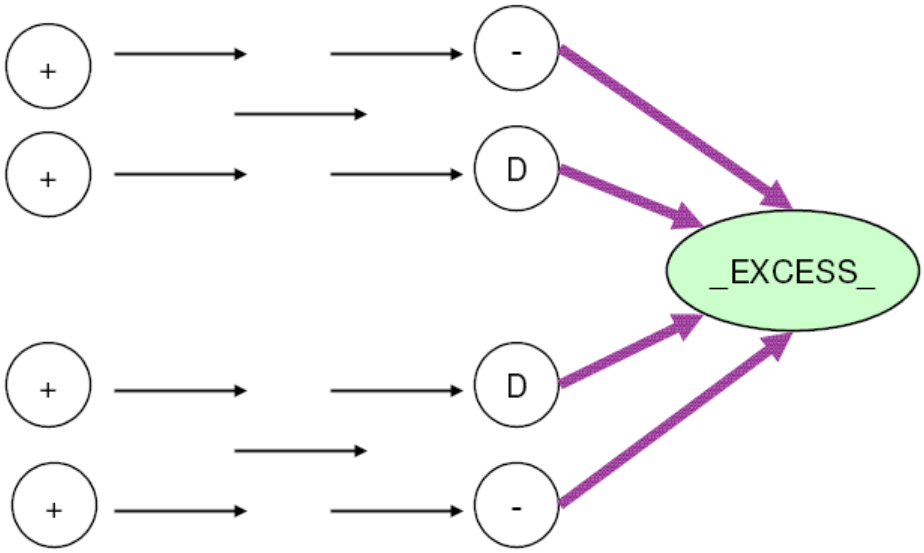


The EXCESS=DEMAND Option

If you specify the EXCESS=DEMAND option, then there are three possible scenarios to deal with:

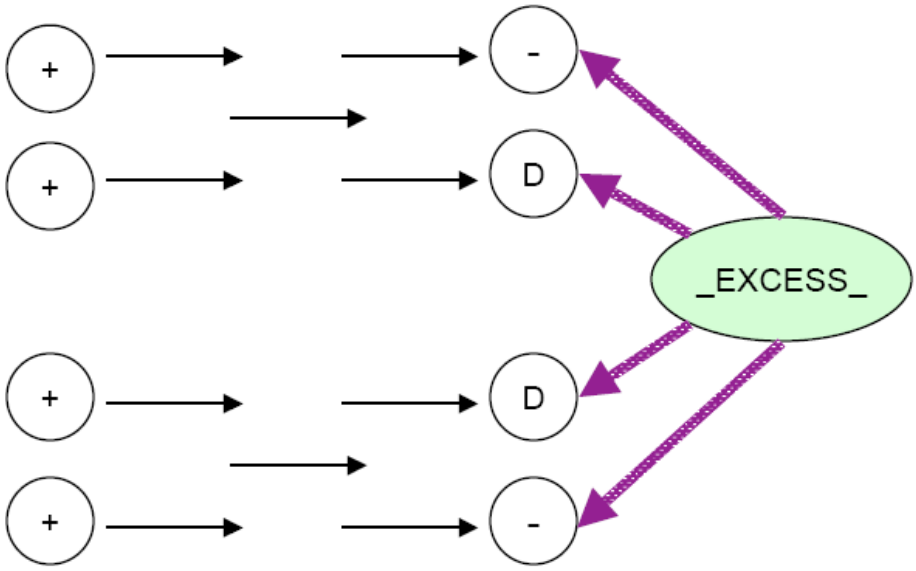
Case 1: No Node with Missing S Supply, THRUNET Not Specified (see Figure 6.36)
Supply the excess demand to all demand nodes directly.

Figure 6.36 Nodes with Missing D Demand



Case 2: Some Nodes with Missing S Supply, THRUNET Not Specified (see Figure 6.37)
Supply the excess demand by the nodes that have a missing S supply value. If a node has a missing S supply value, then the amount it supplies is determined by optimization. For a supply node with a positive supdem value, that value is equal to the sum of flows on all actual arcs directed away from that node.

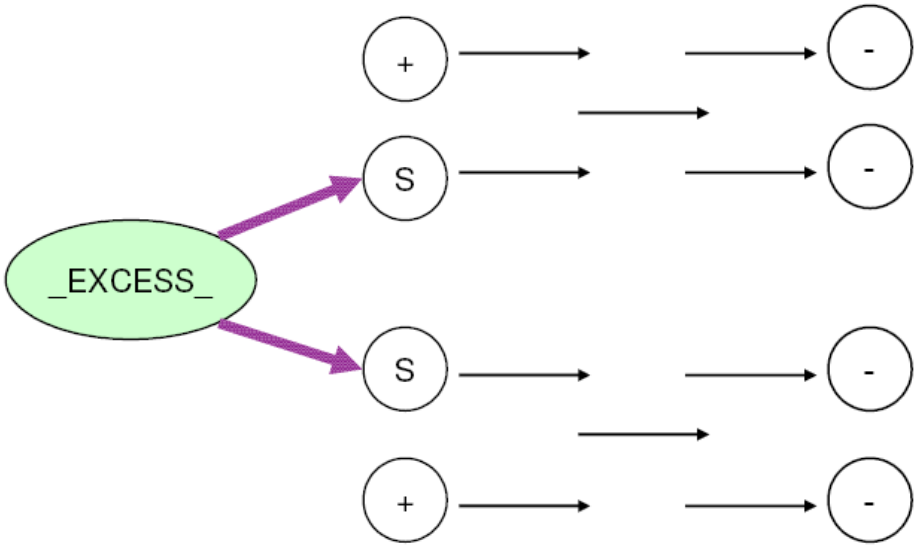
Figure 6.37 Nodes with Missing S Supply



Case 3: THRUNET Specified (see Figure 6.38)

Supply the excess demand by all supply nodes. If a node has a positive supdem value, that value is the lower bound on the sum of flows on all actual arcs directed away from that node. If a node has a missing S supply value, then the amount it supplies is determined by optimization.

Figure 6.38 Nodes with Missing S Supply, THRUNET Specified



Examples: NETFLOW Procedure

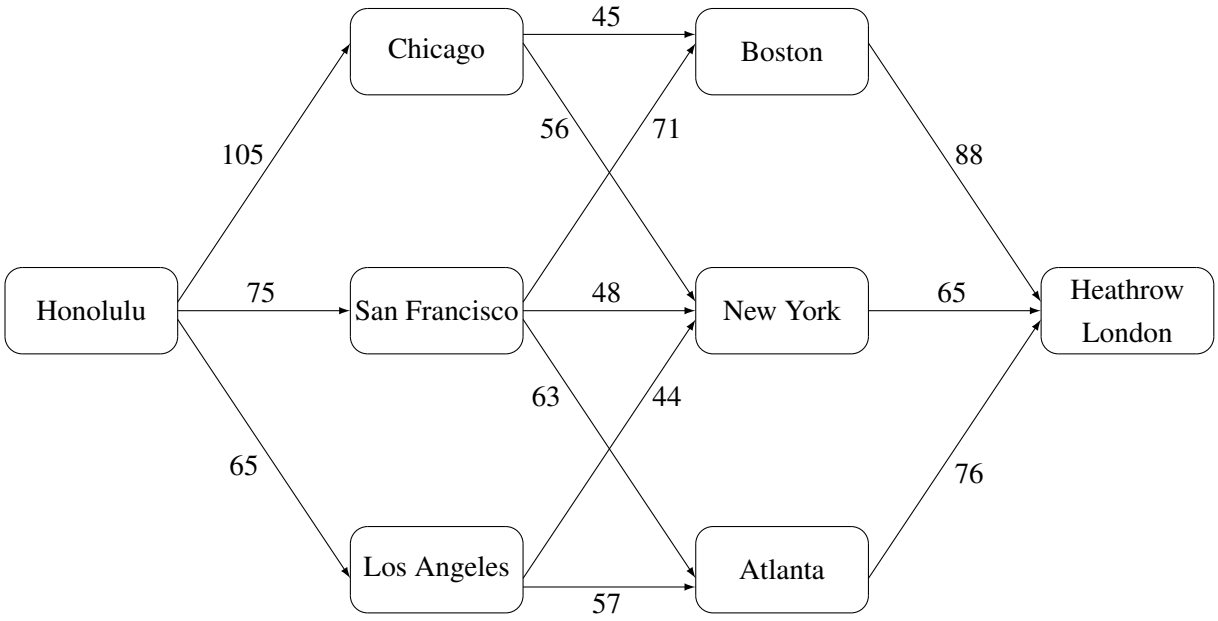
The following examples illustrate some of the capabilities of PROC NETFLOW. These examples, together with the other SAS/OR examples, can be found in the SAS sample library.

Example 6.1: Shortest Path Problem

Whole pineapples are served in a restaurant in London. To ensure freshness, the pineapples are purchased in Hawaii and air freighted from Honolulu to Heathrow in London. The network diagram in Figure 6.39 outlines the different routes that the pineapples could take.

The cost to freight a pineapple is known for each arc. You can use PROC NETFLOW to determine what routes should be used to minimize total shipping cost. The shortest path is the least cost path that all pineapples should use. The SHORTPATH option indicates this type of network problem.

Figure 6.39 Pineapple Routes: Shortest Path Problem



The SINK= option value HEATHROW LONDON is not a valid SAS variable name so it must be enclosed in single quotes. The TAILNODE list variable is FFROM. Because the name of this variable is not _TAIL_ or _FROM_, the TAILNODE list must be specified in the PROC NETFLOW statement. The HEADNODE list must also be explicitly specified because the variable that belongs to this list does not have the name _HEAD_ or _TO_, but is TTO.

```
  title 'Shortest Path Problem';
  title2 'How to get Hawaiian Pineapples to a London Restaurant';
  data aircost1;
     input   ffrom&$13. tto&$15. _cost_ ;
     datalines;
  Honolulu       Chicago          105
  Honolulu       San Francisco     75
  Honolulu       Los Angeles       68
  Chicago        Boston            45
  Chicago        New York          56
  San Francisco  Boston            71
  San Francisco  New York          48
  San Francisco  Atlanta           63
  Los Angeles    New York          44
  Los Angeles    Atlanta           57
  Boston         Heathrow London   88
  New York       Heathrow London   65
  Atlanta        Heathrow London   76
  ;

proc netflow
   shortpath
   sourcenode=Honolulu
   sinknode='Heathrow London'   /* Quotes for embedded blank */
   ARCDATA=aircost1
   arcout=spath;
   tail    ffrom;
   head    tto;
run;

proc print data=spath;
   sum _fcost_;
run;
```

The length at optimality is written to the SAS log as

```
NOTE: Number of nodes= 8 .
NOTE: Number of arcs= 13 .
NOTE: Number of iterations performed (neglecting any constraints)= 5 .
NOTE: Of these, 3 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Shortest path= 177 .
NOTE: The data set WORK.SPATH has 13 observations and 13 variables.
```

The output data set ARCOUT=SPATH in Output 6.1.1 shows that the best route for the pineapples is from Honolulu to Los Angeles to New York to Heathrow London.

Output 6.1.1 ARCOUT=SPATH

Shortest Path Problem
How to get Hawaiian Pineapples to a London Restaurant

Obs	ffrom	tto	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_
1	San Francisco	Atlanta	63	99999999	0	.	.
2	Los Angeles	Atlanta	57	99999999	0	.	.
3	Chicago	Boston	45	99999999	0	.	.
4	San Francisco	Boston	71	99999999	0	.	.
5	Honolulu	Chicago	105	99999999	0	1	.
6	Boston	Heathrow London	88	99999999	0	.	1
7	New York	Heathrow London	65	99999999	0	.	1
8	Atlanta	Heathrow London	76	99999999	0	.	1
9	Honolulu	Los Angeles	68	99999999	0	1	.
10	Chicago	New York	56	99999999	0	.	.
11	San Francisco	New York	48	99999999	0	.	.
12	Los Angeles	New York	44	99999999	0	.	.
13	Honolulu	San Francisco	75	99999999	0	1	.

Obs	_FLOW_	_FCOST_	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	
1	0	0	37	9	3	LOWERBD	NONBASIC
2	0	0	24	10	4	LOWERBD	NONBASIC
3	0	0	49	4	2	LOWERBD	NONBASIC
4	0	0	45	5	3	LOWERBD	NONBASIC
5	0	0	.	1	1	KEY_ARC	BASIC
6	0	0	12	11	5	LOWERBD	NONBASIC
7	1	65	.	12	6	KEY_ARC	BASIC
8	0	0	.	13	7	KEY_ARC	BASIC
9	1	68	.	3	1	KEY_ARC	BASIC
10	0	0	49	6	2	LOWERBD	NONBASIC
11	0	0	11	7	3	LOWERBD	NONBASIC
12	1	44	.	8	4	KEY_ARC	BASIC
13	0	0	.	2	1	KEY_ARC	BASIC
		177					

Example 6.2: Minimum Cost Flow Problem

You can continue to use the pineapple example in Example 6.1 by supposing that the airlines now stipulate that no more than 350 pineapples per week can be handled in any single leg of the journey. The restaurant uses 500 pineapples each week. How many pineapples should take each route between Hawaii and London?

You will probably have more minimum cost flow problems because they are more general than maximal flow and shortest path problems. A shortest path formulation is no longer valid because the sink node does not demand one flow unit.

All arcs have the same capacity of 350 pineapples. Because of this, the DEFCAPACITY= option can be specified in the PROC NETFLOW statement, rather than having a CAPACITY list variable in ARCDATA=aircost1. You can have a CAPACITY list variable, but the value of this variable would be 350 in all observations, so using the DEFCAPACITY= option is more convenient. You would have to use the CAPACITY list variable if arcs had differing capacities. You can use both the DEFCAPACITY= option and a CAPACITY list variable.

There is only one supply node and one demand node. These can be named in the SOURCE= and SINK= options. DEMAND=500 is specified for the restaurant demand. There is no need to specify SUPPLY=500, as this is assumed.

```
title 'Minimum Cost Flow Problem';
title2 'How to get Hawaiian Pineapples to a London Restaurant';
proc netflow
  defcapacity=350
  sourcenode='Honolulu'
  sinknode='Heathrow London'  /* Quotes for embedded blank */
  demand=500
     arcdata=aircost1
     arcout=arcout1
     nodeout=nodeout1;
        tail    ffrom;
        head    tto;
  set future1;
  run;
  quit;

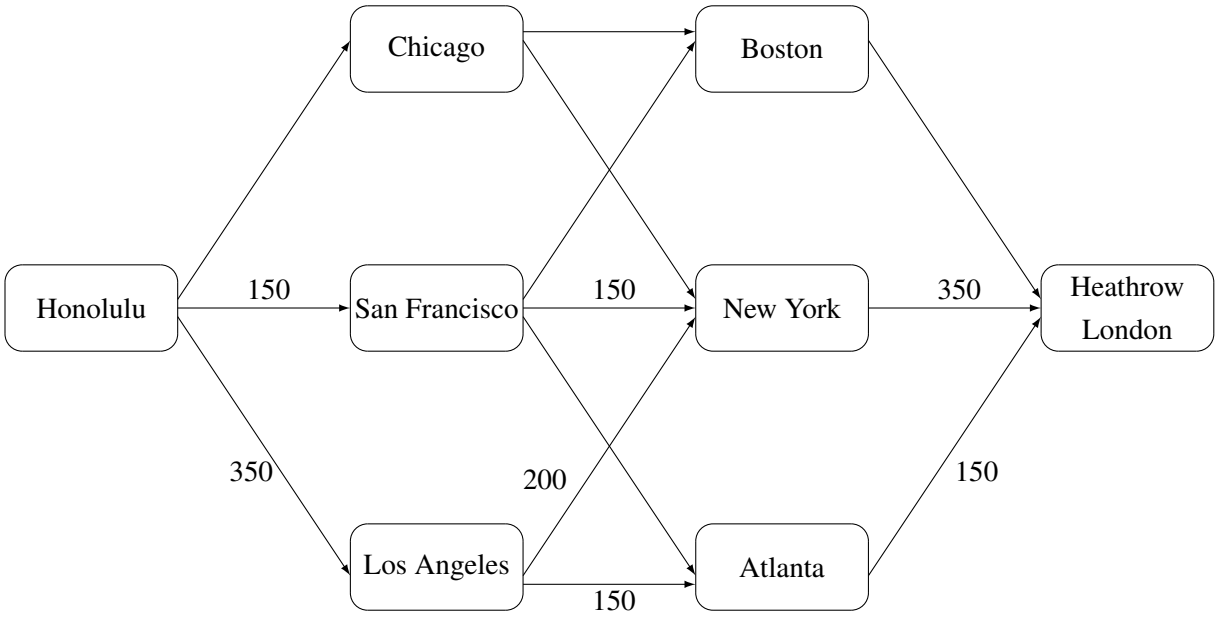

proc print data=arcout1;sum _fcost_;run;


proc print data=nodeout1;
run;
```

The following notes appear on the SAS log:

```
NOTE: SOURCENODE was assigned supply of the total network demand= 500 .
NOTE: Number of nodes= 8 .
NOTE: Number of supply nodes= 1 .
NOTE: Number of demand nodes= 1 .
NOTE: Total supply= 500 , total demand= 500 .
NOTE: Number of arcs= 13 .
NOTE: Number of iterations performed (neglecting any constraints)= 8 .
NOTE: Of these, 4 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 93750 .
NOTE: The data set WORK.ARCOUT1 has 13 observations and 13 variables.
NOTE: The data set WORK.NODEOUT1 has 9 observations and 10 variables.
```

Figure 6.40 Pineapple Routes: Minimum Cost Flow Solution



The routes and numbers of pineapples in each arc can be seen in the output data set ARCOUT=arcout1 in Output 6.2.1. NODEOUT=NODEOUT1 is shown in Output 6.2.2.

Output 6.2.1 ARCOUT=ARCOUT1

Obs	ffrom	tto	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_
1	San Francisco	Atlanta	63	350	0	.	.
2	Los Angeles	Atlanta	57	350	0	.	.
3	Chicago	Boston	45	350	0	.	.
4	San Francisco	Boston	71	350	0	.	.
5	Honolulu	Chicago	105	350	0	500	.
6	Boston	Heathrow London	88	350	0	.	500
7	New York	Heathrow London	65	350	0	.	500
8	Atlanta	Heathrow London	76	350	0	.	500
9	Honolulu	Los Angeles	68	350	0	500	.
10	Chicago	New York	56	350	0	.	.
11	San Francisco	New York	48	350	0	.	.
12	Los Angeles	New York	44	350	0	.	.
13	Honolulu	San Francisco	75	350	0	500	.

Obs	_FLOW_	_FCOST_	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	
1	0	0	2	9	3	LOWERBD	NONBASIC
2	150	8550	.	10	4	KEY_ARC	BASIC
3	0	0	4	4	2	LOWERBD	NONBASIC
4	0	0	.	5	3	KEY_ARC	BASIC
5	0	0	.	1	1	KEY_ARC	BASIC
6	0	0	22	11	5	LOWERBD	NONBASIC
7	350	22750	-24	12	6	UPPERBD	NONBASIC
8	150	11400	.	13	7	KEY_ARC	BASIC
9	350	23800	-11	3	1	UPPERBD	NONBASIC
10	0	0	38	6	2	LOWERBD	NONBASIC
11	150	7200	.	7	3	KEY_ARC	BASIC
12	200	8800	.	8	4	KEY_ARC	BASIC
13	150	11250	.	2	1	KEY_ARC	BASIC
		93750					

Output 6.2.2 NODEOUT=NODEOUT1

Obs	_NODE_	_SUPDEM_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_	_FBQ_
1	_ROOT_	0	0	9	0	1	0	-1	81	-14
2	Atlanta	.	-136	7	4	8	2	10	150	9
3	Boston	.	-146	5	3	2	1	5	0	4
4	Chicago	.	-105	2	1	9	1	1	0	1
5	Heathrow London	-500	-212	8	7	5	1	13	150	11
6	Honolulu	500	0	1	9	3	8	-14	0	-1
7	Los Angeles	.	-79	4	6	7	3	-8	200	3
8	New York	.	-123	6	3	4	4	7	150	6
9	San Francisco	.	-75	3	1	6	6	2	150	2

Example 6.3: Using a Warm Start

Suppose that the airlines state that the freight cost per pineapple in flights that leave Chicago has been reduced by 30. How many pineapples should take each route between Hawaii and London? This example illustrates how PROC NETFLOW uses a warm start.

In Example 6.2, the RESET statement of PROC NETFLOW is used to specify FUTURE1. A NODEOUT= data set is also specified. The warm start information is saved in the arcout1 and nodeout1 data sets.

In the following DATA step, the costs, reduced costs, and flows in the arcout1 data set are saved in variables called oldcost, oldflow, and oldfc. These variables form an implicit ID list in the following PROC NETFLOW run and will appear in ARCOUT=arcout2. Thus, it is easy to compare the previous optimum and the new optimum.

```
title 'Minimum Cost Flow Problem - Warm Start';
title2 'How to get Hawaiian Pineapples to a London Restaurant';
data aircost2;
  set arcout1;
     oldcost=_cost_;
     oldflow=_flow_;
     oldfc=_fcost_;
     if ffrom='Chicago' then _cost_=_cost_-30;

proc netflow
  warm
     arcdata=aircost2
     nodedata=nodeout1
     arcout=arcout2;
        tail    ffrom;
        head    tto;
        run;
        quit;

proc print data=arcout2;
  var ffrom tto _cost_ oldcost _capac_ _lo_
      _flow_ oldflow _fcost_ oldfc;
  sum _fcost_ oldfc;
run;
```

The following notes appear on the SAS log:

```
NOTE: Number of nodes= 8 .
NOTE: Number of supply nodes= 1 .
NOTE: Number of demand nodes= 1 .
NOTE: The greater of total supply and total demand= 500 .
NOTE: Number of iterations performed (neglecting any constraints)= 3 .
NOTE: Of these, 1 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 93150 .
NOTE: The data set WORK.ARCOUT2 has 13 observations and 16 variables.
```

ARCOUT=arcout2 is shown in Output 6.3.1.

Output 6.3.1 ARCOUT=ARCOUT2

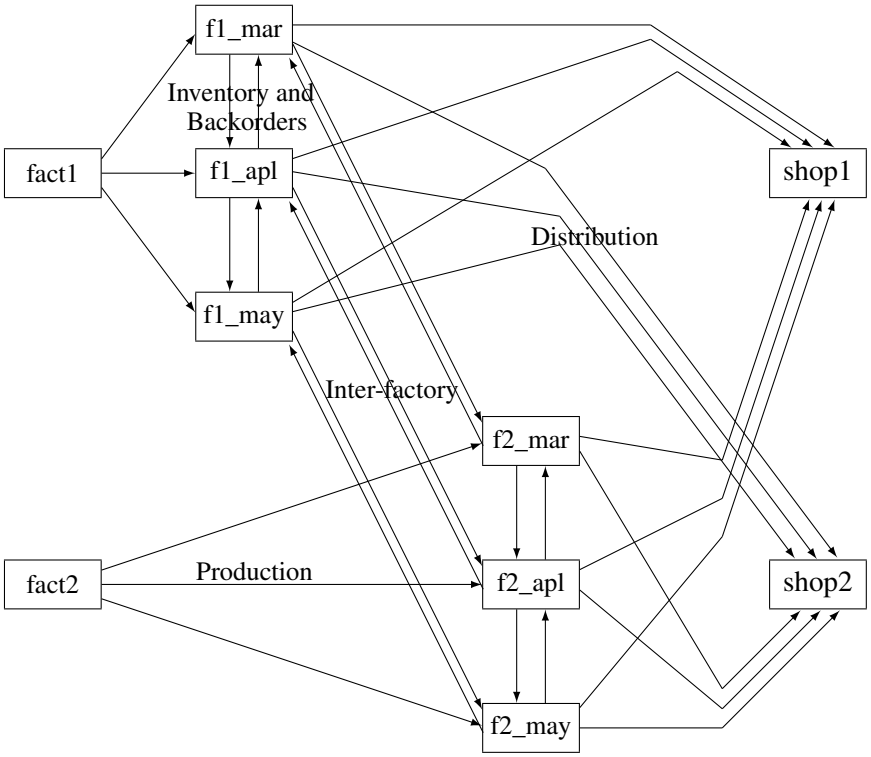
Obs	ffrom	tto	_cost_	oldcost	_CAPAC_	_LO_	_FLOW_	oldflow	_FCOST_	oldfc
1	San Francisco	Atlanta	63	63	350	0	0	0	0	0
2	Los Angeles	Atlanta	57	57	350	0	0	150	0	8550
3	Chicago	Boston	15	45	350	0	150	0	2250	0
4	San Francisco	Boston	71	71	350	0	0	0	0	0
5	Honolulu	Chicago	105	105	350	0	150	0	15750	0
6	Boston	Heathrow London	88	88	350	0	150	0	13200	0
7	New York	Heathrow London	65	65	350	0	350	350	22750	22750
8	Atlanta	Heathrow London	76	76	350	0	0	150	0	11400
9	Honolulu	Los Angeles	68	68	350	0	350	350	23800	23800
10	Chicago	New York	26	56	350	0	0	0	0	0
11	San Francisco	New York	48	48	350	0	0	150	0	7200
12	Los Angeles	New York	44	44	350	0	350	200	15400	8800
13	Honolulu	San Francisco	75	75	350	0	0	150	0	11250
									93150	**93750**

Example 6.4: Production, Inventory, Distribution Problem

Example 6.4 through Example 6.8 use data from a company that produces two sizes of televisions in order to illustrate variations in the use the NETFLOW procedure. The company makes televisions with a diagonal screen measurement of either 19 inches or 25 inches. These televisions are made between March and May at both of the company's two factories. Each factory has a limit on the total number of televisions of each screen dimension that can be made during those months.

The televisions are distributed to one of two shops, stored at the factory where they were made and sold later, or shipped to the other factory. Some sets can be used to fill backorders from the previous months. Each shop demands a number of each type of TV for the months of March through May. The following network in Figure 6.41 illustrates the model. Arc costs can be interpreted as production costs, storage costs, backorder penalty costs, inter-factory transportation costs, and sales profits. The arcs can have capacities and lower flow bounds.

Figure 6.41 TV Problem



There are two similarly structured networks, one for the 19-inch televisions and the other for the 25-inch screen TVs. The minimum cost production, inventory, and distribution plan for both TV types can be determined in the same run of PROC NETFLOW. To ensure that node names are unambiguous, the names of nodes in the 19-inch network have suffix _1, and the node names in the 25-inch network have suffix _2.

The FUTURE1 option is specified because further processing could be required. Information concerning an optimal solution is retained so it can be used to warm start later optimizations. Warm start information is mostly in variables named _NNUMB_, _PRED_, _TRAV_, _SCESS_, _ARCID_, and _FBQ_ and in observations for nodes named _EXCESS_ and _ROOT_, that are in the NODEOUT=NODE2 output data set. (PROC NETFLOW uses similar devices to store warm start information in the DUALOUT= data set when the FUTURE2 option is specified.) Variables _ANUMB_ and _TNUMB_ and observations for arcs directed from or toward a node called _EXCESS_ are present in ARCOUT=arc1. (PROC NETFLOW uses similar devices to store warm start information in the CONOUT= data set when the FUTURE2 option is specified.)

The following code shows how to save the problem data in data sets and solve the model with PROC NETFLOW.

```
title 'Minimum Cost Flow problem';
title2 'Production Planning/Inventory/Distribution';
data node0;
  input _node_ $ _supdem_ ;
  datalines;
fact1_1   1000
fact2_1    850
fact1_2   1000
fact2_2   1500
shop1_1   -900
```

```
   shop2_1   -900
   shop1_2   -900
   shop2_2  -1450
   ;

data arc0;
  input _tail_ $ _head_ $ _cost_ _capac_ _lo_ diagonal factory
        key_id $10. mth_made $ _name_&$17. ;
  datalines;
fact1_1  f1_mar_1  127.9  500 50 19 1 production March prod f1 19 mar
fact1_1  f1_apr_1   78.6  600 50 19 1 production April prod f1 19 apl
fact1_1  f1_may_1   95.1  400 50 19 1 production May   .
f1_mar_1 f1_apr_1   15     50  . 19 1 storage    March .
f1_apr_1 f1_may_1   12     50  . 19 1 storage    April .
f1_apr_1 f1_mar_1   28     20  . 19 1 backorder  April back f1 19 apl
f1_may_1 f1_apr_1   28     20  . 19 1 backorder  May   back f1 19 may
f1_mar_1 f2_mar_1   11      .  . 19 . f1_to_2    March .
f1_apr_1 f2_apr_1   11      .  . 19 . f1_to_2    April .
f1_may_1 f2_may_1   16      .  . 19 . f1_to_2    May   .
f1_mar_1 shop1_1  -327.65 250  . 19 1 sales      March .
f1_apr_1 shop1_1  -300    250  . 19 1 sales      April .
f1_may_1 shop1_1  -285    250  . 19 1 sales      May   .
f1_mar_1 shop2_1  -362.74 250  . 19 1 sales      March .
f1_apr_1 shop2_1  -300    250  . 19 1 sales      April .
f1_may_1 shop2_1  -245    250  . 19 1 sales      May   .
fact2_1  f2_mar_1   88.0  450 35 19 2 production March prod f2 19 mar
fact2_1  f2_apr_1   62.4  480 35 19 2 production April prod f2 19 apl
fact2_1  f2_may_1  133.8  250 35 19 2 production May   .
f2_mar_1 f2_apr_1   18     30  . 19 2 storage    March .
f2_apr_1 f2_may_1   20     30  . 19 2 storage    April .
f2_apr_1 f2_mar_1   17     15  . 19 2 backorder  April back f2 19 apl
f2_may_1 f2_apr_1   25     15  . 19 2 backorder  May   back f2 19 may
f2_mar_1 f1_mar_1   10     40  . 19 . f2_to_1    March .
f2_apr_1 f1_apr_1   11     40  . 19 . f2_to_1    April .
f2_may_1 f1_may_1   13     40  . 19 . f2_to_1    May   .
f2_mar_1 shop1_1  -297.4  250  . 19 2 sales      March .
f2_apr_1 shop1_1  -290    250  . 19 2 sales      April .
f2_may_1 shop1_1  -292    250  . 19 2 sales      May   .
f2_mar_1 shop2_1  -272.7  250  . 19 2 sales      March .
f2_apr_1 shop2_1  -312    250  . 19 2 sales      April .
f2_may_1 shop2_1  -299    250  . 19 2 sales      May   .
fact1_2  f1_mar_2  217.9  400 40 25 1 production March prod f1 25 mar
fact1_2  f1_apr_2  174.5  550 50 25 1 production April prod f1 25 apl
fact1_2  f1_may_2  133.3  350 40 25 1 production May   .
f1_mar_2 f1_apr_2   20     40  . 25 1 storage    March .
f1_apr_2 f1_may_2   18     40  . 25 1 storage    April .
f1_apr_2 f1_mar_2   32     30  . 25 1 backorder  April back f1 25 apl
f1_may_2 f1_apr_2   41     15  . 25 1 backorder  May   back f1 25 may
f1_mar_2 f2_mar_2   23      .  . 25 . f1_to_2    March .
f1_apr_2 f2_apr_2   23      .  . 25 . f1_to_2    April .
f1_may_2 f2_may_2   26      .  . 25 . f1_to_2    May   .
f1_mar_2 shop1_2  -559.76   .  . 25 1 sales      March .
f1_apr_2 shop1_2  -524.28   .  . 25 1 sales      April .
f1_may_2 shop1_2  -475.02   .  . 25 1 sales      May   .
```

```
f1_mar_2 shop2_2  -623.89   .   . 25 1 sales      March .
f1_apr_2 shop2_2  -549.68   .   . 25 1 sales      April .
f1_may_2 shop2_2  -460.00   .   . 25 1 sales      May   .
fact2_2  f2_mar_2  182.0  650 35 25 2 production March prod f2 25 mar
fact2_2  f2_apr_2  196.7  680 35 25 2 production April prod f2 25 apl
fact2_2  f2_may_2  201.4  550 35 25 2 production May   .
f2_mar_2 f2_apr_2   28     50  . 25 2 storage    March .
f2_apr_2 f2_may_2   38     50  . 25 2 storage    April .
f2_apr_2 f2_mar_2   31     15  . 25 2 backorder  April back f2 25 apl
f2_may_2 f2_apr_2   54     15  . 25 2 backorder  May   back f2 25 may
f2_mar_2 f1_mar_2   20     25  . 25 . f2_to_1    March .
f2_apr_2 f1_apr_2   21     25  . 25 . f2_to_1    April .
f2_may_2 f1_may_2   43     25  . 25 . f2_to_1    May   .
f2_mar_2 shop1_2  -567.83 500  . 25 2 sales      March .
f2_apr_2 shop1_2  -542.19 500  . 25 2 sales      April .
f2_may_2 shop1_2  -461.56 500  . 25 2 sales      May   .
f2_mar_2 shop2_2  -542.83 500  . 25 2 sales      March .
f2_apr_2 shop2_2  -559.19 500  . 25 2 sales      April .
f2_may_2 shop2_2  -489.06 500  . 25 2 sales      May   .
;

proc netflow
      nodedata=node0
      arcdata=arc0;
  set future1
      nodeout=node2
      arcout=arc1;
      run;
      quit;

options ls=80 ps = 50;
proc print data=arc1 heading=h width=min;
var _tail_ _head_ _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_ _fcost_;
sum _fcost_;
run;

options ls=80 ps = 50;
proc print data=arc1 heading=h width=min;
var _rcost_ _anumb_ _tnumb_ _status_ diagonal factory key_id mth_made;
run;

proc print data=node2;
run;
```

The following notes appear on the SAS log:

```
NOTE: Number of nodes= 20 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 4 .
NOTE: Total supply= 4350 , total demand= 4150 .
NOTE: Number of arcs= 64 .
NOTE: Number of iterations performed (neglecting any constraints)= 75 .
NOTE: Of these, 1 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= -1281110.35 .
NOTE: The data set WORK.ARC1 has 68 observations and 18 variables.
NOTE: The data set WORK.NODE2 has 22 observations and 10 variables.
```

The solution is given in the NODEOUT=node2 and ARCOUT=arc1 data sets. In the ARCOUT= data set, shown in Output 6.4.1 and Output 6.4.2, the variables diagonal, factory, key_id, and mth_made form an implicit ID list. The diagonal variable has one of two values, 19 or 25. factory also has one of two values, 1 or 2, to denote the factory where either production or storage occurs, from where TVs are either sold to shops or satisfy backorders. PRODUCTION, STORAGE, SALES, and BACKORDER are values of the key_id variable.

Other values of this variable, F1_TO_2 and F2_TO_1, are used when flow through arcs represents the transportation of TVs between factories. The mth_made variable has values MARCH, APRIL, and MAY, the months when TVs that are modeled as flow through an arc were made (assuming that no televisions are stored for more than one month and none manufactured in May are used to fill March backorders).

These ID variables can be used after the PROC NETFLOW run to produce reports and perform analysis on particular parts of the company's operation. For example, reports can be generated for production numbers for each factory; optimal sales figures for each shop; and how many TVs should be stored, used to fill backorders, sent to the other factory, or any combination of these, for TVs with a particular screen, those produced in a particular month, or both.

Output 6.4.1 ARCOUT=ARC1

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	fact1_1	_EXCESS_	0.00	99999999	0		1000	200	5	0.00
2	fact2_1	_EXCESS_	0.00	99999999	0		850	200	45	0.00
3	fact1_2	_EXCESS_	0.00	99999999	0		1000	200	10	0.00
4	fact2_2	_EXCESS_	0.00	99999999	0		1500	200	140	0.00
5	fact1_1	f1_apr_1	78.60	600	50	prod f1 19 apl	1000	.	600	47160.00
6	f1_mar_1	f1_apr_1	15.00	50	0		.	.	0	0.00
7	f1_may_1	f1_apr_1	28.00	20	0	back f1 19 may	.	.	0	0.00
8	f2_apr_1	f1_apr_1	11.00	40	0		.	.	0	0.00
9	fact1_2	f1_apr_2	174.50	550	50	prod f1 25 apl	1000	.	550	95975.00
10	f1_mar_2	f1_apr_2	20.00	40	0		.	.	0	0.00
11	f1_may_2	f1_apr_2	41.00	15	0	back f1 25 may	.	.	15	615.00
12	f2_apr_2	f1_apr_2	21.00	25	0		.	.	0	0.00
13	fact1_1	f1_mar_1	127.90	500	50	prod f1 19 mar	1000	.	345	44125.50
14	f1_apr_1	f1_mar_1	28.00	20	0	back f1 19 apl	.	.	20	560.00
15	f2_mar_1	f1_mar_1	10.00	40	0		.	.	40	400.00
16	fact1_2	f1_mar_2	217.90	400	40	prod f1 25 mar	1000	.	400	87160.00
17	f1_apr_2	f1_mar_2	32.00	30	0	back f1 25 apl	.	.	30	960.00
18	f2_mar_2	f1_mar_2	20.00	25	0		.	.	25	500.00
19	fact1_1	f1_may_1	95.10	400	50		1000	.	50	4755.00
20	f1_apr_1	f1_may_1	12.00	50	0		.	.	50	600.00
21	f2_may_1	f1_may_1	13.00	40	0		.	.	0	0.00
22	fact1_2	f1_may_2	133.30	350	40		1000	.	40	5332.00
23	f1_apr_2	f1_may_2	18.00	40	0		.	.	0	0.00
24	f2_may_2	f1_may_2	43.00	25	0		.	.	0	0.00
25	f1_apr_1	f2_apr_1	11.00	99999999	0		.	.	30	330.00
26	fact2_1	f2_apr_1	62.40	480	35	prod f2 19 apl	850	.	480	29952.00
27	f2_mar_1	f2_apr_1	18.00	30	0		.	.	0	0.00
28	f2_may_1	f2_apr_1	25.00	15	0	back f2 19 may	.	.	0	0.00
29	f1_apr_2	f2_apr_2	23.00	99999999	0		.	.	0	0.00
30	fact2_2	f2_apr_2	196.70	680	35	prod f2 25 apl	1500	.	680	133756.00
31	f2_mar_2	f2_apr_2	28.00	50	0		.	.	0	0.00
32	f2_may_2	f2_apr_2	54.00	15	0	back f2 25 may	.	.	15	810.00
33	f1_mar_1	f2_mar_1	11.00	99999999	0		.	.	0	0.00
34	fact2_1	f2_mar_1	88.00	450	35	prod f2 19 mar	850	.	290	25520.00
35	f2_apr_1	f2_mar_1	17.00	15	0	back f2 19 apl	.	.	0	0.00
36	f1_mar_2	f2_mar_2	23.00	99999999	0		.	.	0	0.00
37	fact2_2	f2_mar_2	182.00	650	35	prod f2 25 mar	1500	.	645	117390.00
38	f2_apr_2	f2_mar_2	31.00	15	0	back f2 25 apl	.	.	0	0.00
39	f1_may_1	f2_may_1	16.00	99999999	0		.	.	100	1600.00
40	fact2_1	f2_may_1	133.80	250	35		850	.	35	4683.00
41	f2_apr_1	f2_may_1	20.00	30	0		.	.	15	300.00
42	f1_may_2	f2_may_2	26.00	99999999	0		.	.	0	0.00
43	fact2_2	f2_may_2	201.40	550	35		1500	.	35	7049.00
44	f2_apr_2	f2_may_2	38.00	50	0		.	.	0	0.00
45	f1_mar_1	shop1_1	-327.65	250	0		.	900	155	-50785.75
46	f1_apr_1	shop1_1	-300.00	250	0		.	900	250	-75000.00
47	f1_may_1	shop1_1	-285.00	250	0		.	900	0	0.00
48	f2_mar_1	shop1_1	-297.40	250	0		.	900	250	-74350.00

Output 6.4.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
49	f2_apr_1	shop1_1	-290.00	250	0		.	900	245	-71050.00
50	f2_may_1	shop1_1	-292.00	250	0		.	900	0	0.00
51	f1_mar_2	shop1_2	-559.76	99999999	0		.	900	0	0.00
52	f1_apr_2	shop1_2	-524.28	99999999	0		.	900	0	0.00
53	f1_may_2	shop1_2	-475.02	99999999	0		.	900	25	-11875.50
54	f2_mar_2	shop1_2	-567.83	500	0		.	900	500	-283915.00
55	f2_apr_2	shop1_2	-542.19	500	0		.	900	375	-203321.25
56	f2_may_2	shop1_2	-461.56	500	0		.	900	0	0.00
57	f1_mar_1	shop2_1	-362.74	250	0		.	900	250	-90685.00
58	f1_apr_1	shop2_1	-300.00	250	0		.	900	250	-75000.00
59	f1_may_1	shop2_1	-245.00	250	0		.	900	0	0.00
60	f2_mar_1	shop2_1	-272.70	250	0		.	900	0	0.00
61	f2_apr_1	shop2_1	-312.00	250	0		.	900	250	-78000.00
62	f2_may_1	shop2_1	-299.00	250	0		.	900	150	-44850.00
63	f1_mar_2	shop2_2	-623.89	99999999	0		.	1450	455	-283869.95
64	f1_apr_2	shop2_2	-549.68	99999999	0		.	1450	535	-294078.80
65	f1_may_2	shop2_2	-460.00	99999999	0		.	1450	0	0.00
66	f2_mar_2	shop2_2	-542.83	500	0		.	1450	120	-65139.60
67	f2_apr_2	shop2_2	-559.19	500	0		.	1450	320	-178940.80
68	f2_may_2	shop2_2	-489.06	500	0		.	1450	20	-9781.20
										-1281110.35

Output 6.4.2 ARCOUT=ARC1 (continued)

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
1	.	65	1	KEY_ARC BASIC	.	.		
2	.	66	10	KEY_ARC BASIC	.	.		
3	.	67	11	KEY_ARC BASIC	.	.		
4	.	68	20	KEY_ARC BASIC	.	.		
5	-0.650	4	1	UPPERBD NONBASIC	19	1	production	April
6	63.650	5	2	LOWERBD NONBASIC	19	1	storage	March
7	43.000	6	4	LOWERBD NONBASIC	19	1	backorder	May
8	22.000	7	6	LOWERBD NONBASIC	19	.	f2_to_1	April
9	-14.350	36	11	UPPERBD NONBASIC	25	1	production	April
10	94.210	37	12	LOWERBD NONBASIC	25	1	storage	March
11	-16.660	38	14	UPPERBD NONBASIC	25	1	backorder	May
12	30.510	39	16	LOWERBD NONBASIC	25	.	f2_to_1	April
13	.	1	1	KEY_ARC BASIC	19	1	production	March
14	-20.650	2	3	UPPERBD NONBASIC	19	1	backorder	April
15	-29.900	3	5	UPPERBD NONBASIC	19	.	f2_to_1	March
16	-45.160	33	11	UPPERBD NONBASIC	25	1	production	March
17	-42.210	34	13	UPPERBD NONBASIC	25	1	backorder	April
18	-61.060	35	15	UPPERBD NONBASIC	25	.	f2_to_1	March
19	0.850	8	1	LOWERBD NONBASIC	19	1	production	May
20	-3.000	9	3	UPPERBD NONBASIC	19	1	storage	April
21	29.000	10	7	LOWERBD NONBASIC	19	.	f2_to_1	May
22	2.110	40	11	LOWERBD NONBASIC	25	1	production	May
23	75.660	41	13	LOWERBD NONBASIC	25	1	storage	April
24	40.040	42	17	LOWERBD NONBASIC	25	.	f2_to_1	May
25	.	14	3	KEY_ARC BASIC	19	.	f1_to_2	April
26	-27.850	15	10	UPPERBD NONBASIC	19	2	production	April
27	15.750	16	5	LOWERBD NONBASIC	19	2	storage	March
28	45.000	17	7	LOWERBD NONBASIC	19	2	backorder	May
29	13.490	46	13	LOWERBD NONBASIC	25	.	f1_to_2	April
30	-1.660	47	20	UPPERBD NONBASIC	25	2	production	April
31	11.640	48	15	LOWERBD NONBASIC	25	2	storage	March
32	-16.130	49	17	UPPERBD NONBASIC	25	2	backorder	May
33	50.900	11	2	LOWERBD NONBASIC	19	.	f1_to_2	March
34	.	12	10	KEY_ARC BASIC	19	2	production	March
35	19.250	13	6	LOWERBD NONBASIC	19	2	backorder	April
36	104.060	43	12	LOWERBD NONBASIC	25	.	f1_to_2	March
37	.	44	20	KEY_ARC BASIC	25	2	production	March
38	47.360	45	16	LOWERBD NONBASIC	25	2	backorder	April
39	.	18	4	KEY_ARC BASIC	19	.	f1_to_2	May
40	23.550	19	10	LOWERBD NONBASIC	19	2	production	May
41	.	20	6	KEY_ARC BASIC	19	2	storage	April
42	28.960	50	14	LOWERBD NONBASIC	25	.	f1_to_2	May
43	73.170	51	20	LOWERBD NONBASIC	25	2	production	May
44	108.130	52	16	LOWERBD NONBASIC	25	2	storage	April
45	.	21	2	KEY_ARC BASIC	19	1	sales	March
46	-21.000	22	3	UPPERBD NONBASIC	19	1	sales	April
47	9.000	23	4	LOWERBD NONBASIC	19	1	sales	May
48	-9.650	24	5	UPPERBD NONBASIC	19	2	sales	March

Output 6.4.2 *continued*

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
49	.	25	6	KEY_ARC BASIC	19	2	sales	April
50	18.000	26	7	LOWERBD NONBASIC	19	2	sales	May
51	47.130	53	12	LOWERBD NONBASIC	25	1	sales	March
52	8.400	54	13	LOWERBD NONBASIC	25	1	sales	April
53	.	55	14	KEY_ARC BASIC	25	1	sales	May
54	-42.000	56	15	UPPERBD NONBASIC	25	2	sales	March
55	.	57	16	KEY_ARC BASIC	25	2	sales	April
56	10.500	58	17	LOWERBD NONBASIC	25	2	sales	May
57	-46.090	27	2	UPPERBD NONBASIC	19	1	sales	March
58	-32.000	28	3	UPPERBD NONBASIC	19	1	sales	April
59	38.000	29	4	LOWERBD NONBASIC	19	1	sales	May
60	4.050	30	5	LOWERBD NONBASIC	19	2	sales	March
61	-33.000	31	6	UPPERBD NONBASIC	19	2	sales	April
62	.	32	7	KEY_ARC BASIC	19	2	sales	May
63	.	59	12	KEY_ARC BASIC	25	1	sales	March
64	.	60	13	KEY_ARC BASIC	25	1	sales	April
65	32.020	61	14	LOWERBD NONBASIC	25	1	sales	May
66	.	62	15	KEY_ARC BASIC	25	2	sales	March
67	.	63	16	KEY_ARC BASIC	25	2	sales	April
68	.	64	17	KEY_ARC BASIC	25	2	sales	May

Output 6.4.3 NODEOUT=NODE2

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_	_FBQ_
1	_ROOT_	238	0.00	22	0	8	0	3	166	-69
2	_EXCESS_	-200	-100000198.75	21	1	11	13	65	5	65
3	f1_apr_1	.	-100000278.00	3	6	7	1	-14	30	4
4	f1_apr_2	.	-100000387.60	13	19	17	1	-60	535	36
5	f1_mar_1	.	-100000326.65	2	8	1	15	-21	155	1
6	f1_mar_2	.	-100000461.81	12	19	13	1	-59	455	33
7	f1_may_1	.	-100000293.00	4	7	2	1	-18	100	8
8	f1_may_2	.	-100000329.94	14	18	12	1	-55	25	40
9	f2_apr_1	.	-100000289.00	6	8	3	5	-25	245	14
10	f2_apr_2	.	-100000397.11	16	19	18	3	-63	320	46
11	f2_mar_1	.	-100000286.75	5	10	22	1	12	255	11
12	f2_mar_2	.	-100000380.75	15	20	19	8	44	610	43
13	f2_may_1	.	-100000309.00	7	6	9	3	20	15	18
14	f2_may_2	.	-100000326.98	17	19	10	1	-64	20	50
15	fact1_1	1000	-100000198.75	1	2	21	14	-1	295	-1
16	fact1_2	1000	-100000198.75	11	21	20	1	-67	10	-33
17	fact2_1	850	-100000198.75	10	21	5	2	-66	45	-33
18	fact2_2	1500	-100000198.75	20	21	15	9	-68	140	-65
19	shop1_1	-900	-99999999.00	8	22	6	21	0	0	21
20	shop1_2	-900	-99999854.92	18	16	14	2	57	375	53
21	shop2_1	-900	-100000010.00	9	7	4	1	32	150	27
22	shop2_2	-1450	-99999837.92	19	15	16	7	62	120	59

Example 6.5: Using an Unconstrained Solution Warm Start

This example examines the effect of changing some of the arc costs. The backorder penalty costs are increased by twenty percent. The sales profit of 25-inch TVs sent to the shops in May is increased by thirty units. The backorder penalty costs of 25-inch TVs manufactured in May for April consumption is decreased by thirty units. The production cost of 19- and 25-inch TVs made in May are decreased by five units and twenty units, respectively. How does the optimal solution of the network after these arc cost alterations compare with the optimum of the original network? If you want to use the warm start facilities of PROC NETFLOW to solve this undefined problem, specify the WARM option. Notice that the FUTURE1 option was specified in the last PROC NETFLOW run.

The following SAS statements produce the new NODEOUT= and ARCOUT= data sets.

```
title 'Minimum Cost Flow problem- Unconstrained Warm Start';
title2 'Production Planning/Inventory/Distribution';
data arc2;
  set arc1;
  oldcost=_cost_;
  oldfc=_fcost_;
  oldflow=_flow_;
  if key_id='backorder'
     then _cost_=_cost_*1.2;
     else if _tail_='f2_may_2' then _cost_=_cost_-30;
  if key_id='production' & mth_made='May' then
     if diagonal=19 then _cost_=_cost_-5;
                    else _cost_=_cost_-20;


proc netflow
  warm future1
  nodedata=node2
  arcdata=arc2
  nodeout=node3
  arcout=arc3;
  run;
  quit;

options ls=80 ps = 50;
proc print data=arc3 heading=h width=min;
sum _fcost_;
var _tail_ _head_ _capac_ _lo_ _supply_ _demand_ _name_ _cost_ _flow_ _fcost_;
run;

options ls=80 ps = 50;
proc print data=arc3 heading=h width=min;
sum oldfc;
var oldcost oldflow oldfc diagonal factory key_id mth_made _anumb_ _tnumb_;
run;

proc print data=node3;
  run;
```

The following notes appear on the SAS log:

```
NOTE: Number of nodes= 21 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 5 .
NOTE: The greater of total supply and total demand= 4350 .
NOTE: Number of iterations performed (neglecting any constraints)= 8 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= -1285086.45 .
NOTE: The data set WORK.ARC3 has 68 observations and 21 variables.
NOTE: The data set WORK.NODE3 has 22 observations and 10 variables.
```

The solution is displayed in Output 6.5.1 and Output 6.5.2. The associated NODEOUT data set is in Output 6.5.3.

Output 6.5.1 ARCOUT=ARC3

Obs	_tail_	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_	_FCOST_
1	fact1_1	_EXCESS_	99999999	0	1000	200		0.00	5	0.00
2	fact2_1	_EXCESS_	99999999	0	850	200		0.00	45	0.00
3	fact1_2	_EXCESS_	99999999	0	1000	200		0.00	0	0.00
4	fact2_2	_EXCESS_	99999999	0	1500	200		0.00	150	0.00
5	fact1_1	f1_apr_1	600	50	1000	.	prod f1 19 apl	78.60	540	42444.00
6	f1_mar_1	f1_apr_1	50	0	.	.		15.00	0	0.00
7	f1_may_1	f1_apr_1	20	0	.	.	back f1 19 may	33.60	0	0.00
8	f2_apr_1	f1_apr_1	40	0	.	.		11.00	0	0.00
9	fact1_2	f1_apr_2	550	50	1000	.	prod f1 25 apl	174.50	250	43625.00
10	f1_mar_2	f1_apr_2	40	0	.	.		20.00	0	0.00
11	f1_may_2	f1_apr_2	15	0	.	.	back f1 25 may	49.20	15	738.00
12	f2_apr_2	f1_apr_2	25	0	.	.		21.00	0	0.00
13	fact1_1	f1_mar_1	500	50	1000	.	prod f1 19 mar	127.90	340	43486.00
14	f1_apr_1	f1_mar_1	20	0	.	.	back f1 19 apl	33.60	20	672.00
15	f2_mar_1	f1_mar_1	40	0	.	.		10.00	40	400.00
16	fact1_2	f1_mar_2	400	40	1000	.	prod f1 25 mar	217.90	400	87160.00
17	f1_apr_2	f1_mar_2	30	0	.	.	back f1 25 apl	38.40	30	1152.00
18	f2_mar_2	f1_mar_2	25	0	.	.		20.00	25	500.00
19	fact1_1	f1_may_1	400	50	1000	.		90.10	115	10361.50
20	f1_apr_1	f1_may_1	50	0	.	.		12.00	0	0.00
21	f2_may_1	f1_may_1	40	0	.	.		13.00	0	0.00
22	fact1_2	f1_may_2	350	40	1000	.		113.30	350	39655.00
23	f1_apr_2	f1_may_2	40	0	.	.		18.00	0	0.00
24	f2_may_2	f1_may_2	25	0	.	.		13.00	0	0.00
25	f1_apr_1	f2_apr_1	99999999	0	.	.		11.00	20	220.00
26	fact2_1	f2_apr_1	480	35	850	.	prod f2 19 apl	62.40	480	29952.00
27	f2_mar_1	f2_apr_1	30	0	.	.		18.00	0	0.00
28	f2_may_1	f2_apr_1	15	0	.	.	back f2 19 may	30.00	0	0.00
29	f1_apr_2	f2_apr_2	99999999	0	.	.		23.00	0	0.00
30	fact2_2	f2_apr_2	680	35	1500	.	prod f2 25 apl	196.70	680	133756.00
31	f2_mar_2	f2_apr_2	50	0	.	.		28.00	0	0.00
32	f2_may_2	f2_apr_2	15	0	.	.	back f2 25 may	64.80	0	0.00
33	f1_mar_1	f2_mar_1	99999999	0	.	.		11.00	0	0.00
34	fact2_1	f2_mar_1	450	35	850	.	prod f2 19 mar	88.00	290	25520.00
35	f2_apr_1	f2_mar_1	15	0	.	.	back f2 19 apl	20.40	0	0.00
36	f1_mar_2	f2_mar_2	99999999	0	.	.		23.00	0	0.00
37	fact2_2	f2_mar_2	650	35	1500	.	prod f2 25 mar	182.00	635	115570.00
38	f2_apr_2	f2_mar_2	15	0	.	.	back f2 25 apl	37.20	0	0.00
39	f1_may_1	f2_may_1	99999999	0	.	.		16.00	115	1840.00
40	fact2_1	f2_may_1	250	35	850	.		128.80	35	4508.00
41	f2_apr_1	f2_may_1	30	0	.	.		20.00	0	0.00
42	f1_may_2	f2_may_2	99999999	0	.	.		26.00	335	8710.00
43	fact2_2	f2_may_2	550	35	1500	.		181.40	35	6349.00
44	f2_apr_2	f2_may_2	50	0	.	.		38.00	0	0.00
45	f1_mar_1	shop1_1	250	0	.	900		-327.65	150	-49147.50
46	f1_apr_1	shop1_1	250	0	.	900		-300.00	250	-75000.00
47	f1_may_1	shop1_1	250	0	.	900		-285.00	0	0.00
48	f2_mar_1	shop1_1	250	0	.	900		-297.40	250	-74350.00

Output 6.5.1 *continued*

Obs	_tail_	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_	_FCOST_
49	f2_apr_1	shop1_1	250	0	.	900		-290.00	250	-72500.00
50	f2_may_1	shop1_1	250	0	.	900		-292.00	0	0.00
51	f1_mar_2	shop1_2	99999999	0	.	900		-559.76	0	0.00
52	f1_apr_2	shop1_2	99999999	0	.	900		-524.28	0	0.00
53	f1_may_2	shop1_2	99999999	0	.	900		-475.02	0	0.00
54	f2_mar_2	shop1_2	500	0	.	900		-567.83	500	-283915.00
55	f2_apr_2	shop1_2	500	0	.	900		-542.19	400	-216876.00
56	f2_may_2	shop1_2	500	0	.	900		-491.56	0	0.00
57	f1_mar_1	shop2_1	250	0	.	900		-362.74	250	-90685.00
58	f1_apr_1	shop2_1	250	0	.	900		-300.00	250	-75000.00
59	f1_may_1	shop2_1	250	0	.	900		-245.00	0	0.00
60	f2_mar_1	shop2_1	250	0	.	900		-272.70	0	0.00
61	f2_apr_1	shop2_1	250	0	.	900		-312.00	250	-78000.00
62	f2_may_1	shop2_1	250	0	.	900		-299.00	150	-44850.00
63	f1_mar_2	shop2_2	99999999	0	.	1450		-623.89	455	-283869.95
64	f1_apr_2	shop2_2	99999999	0	.	1450		-549.68	235	-129174.80
65	f1_may_2	shop2_2	99999999	0	.	1450		-460.00	0	0.00
66	f2_mar_2	shop2_2	500	0	.	1450		-542.83	110	-59711.30
67	f2_apr_2	shop2_2	500	0	.	1450		-559.19	280	-156573.20
68	f2_may_2	shop2_2	500	0	.	1450		-519.06	370	-192052.20
										-1285086.45

Output 6.5.2 ARCOUT=ARC3 (continued)

Obs	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made	_ANUMB_	_TNUMB_
1	0.00	5	0.00	.	.			65	1
2	0.00	45	0.00	.	.			66	10
3	0.00	10	0.00	.	.			67	11
4	0.00	140	0.00	.	.			68	20
5	78.60	600	47160.00	19	1	production	April	4	1
6	15.00	0	0.00	19	1	storage	March	5	2
7	28.00	0	0.00	19	1	backorder	May	6	4
8	11.00	0	0.00	19	.	f2_to_1	April	7	6
9	174.50	550	95975.00	25	1	production	April	36	11
10	20.00	0	0.00	25	1	storage	March	37	12
11	41.00	15	615.00	25	1	backorder	May	38	14
12	21.00	0	0.00	25	.	f2_to_1	April	39	16
13	127.90	345	44125.50	19	1	production	March	1	1
14	28.00	20	560.00	19	1	backorder	April	2	3
15	10.00	40	400.00	19	.	f2_to_1	March	3	5
16	217.90	400	87160.00	25	1	production	March	33	11
17	32.00	30	960.00	25	1	backorder	April	34	13
18	20.00	25	500.00	25	.	f2_to_1	March	35	15
19	95.10	50	4755.00	19	1	production	May	8	1
20	12.00	50	600.00	19	1	storage	April	9	3
21	13.00	0	0.00	19	.	f2_to_1	May	10	7
22	133.30	40	5332.00	25	1	production	May	40	11
23	18.00	0	0.00	25	1	storage	April	41	13
24	43.00	0	0.00	25	.	f2_to_1	May	42	17
25	11.00	30	330.00	19	.	f1_to_2	April	14	3
26	62.40	480	29952.00	19	2	production	April	15	10
27	18.00	0	0.00	19	2	storage	March	16	5
28	25.00	0	0.00	19	2	backorder	May	17	7
29	23.00	0	0.00	25	.	f1_to_2	April	46	13
30	196.70	680	133756.00	25	2	production	April	47	20
31	28.00	0	0.00	25	2	storage	March	48	15
32	54.00	15	810.00	25	2	backorder	May	49	17
33	11.00	0	0.00	19	.	f1_to_2	March	11	2
34	88.00	290	25520.00	19	2	production	March	12	10
35	17.00	0	0.00	19	2	backorder	April	13	6
36	23.00	0	0.00	25	.	f1_to_2	March	43	12
37	182.00	645	117390.00	25	2	production	March	44	20
38	31.00	0	0.00	25	2	backorder	April	45	16
39	16.00	100	1600.00	19	.	f1_to_2	May	18	4
40	133.80	35	4683.00	19	2	production	May	19	10
41	20.00	15	300.00	19	2	storage	April	20	6
42	26.00	0	0.00	25	.	f1_to_2	May	50	14
43	201.40	35	7049.00	25	2	production	May	51	20
44	38.00	0	0.00	25	2	storage	April	52	16
45	-327.65	155	-50785.75	19	1	sales	March	21	2
46	-300.00	250	-75000.00	19	1	sales	April	22	3
47	-285.00	0	0.00	19	1	sales	May	23	4
48	-297.40	250	-74350.00	19	2	sales	March	24	5

Output 6.5.2 *continued*

Obs	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made	_ANUMB_	_TNUMB_
49	-290.00	245	-71050.00	19	2	sales	April	25	6
50	-292.00	0	0.00	19	2	sales	May	26	7
51	-559.76	0	0.00	25	1	sales	March	53	12
52	-524.28	0	0.00	25	1	sales	April	54	13
53	-475.02	25	-11875.50	25	1	sales	May	55	14
54	-567.83	500	-283915.00	25	2	sales	March	56	15
55	-542.19	375	-203321.25	25	2	sales	April	57	16
56	-461.56	0	0.00	25	2	sales	May	58	17
57	-362.74	250	-90685.00	19	1	sales	March	27	2
58	-300.00	250	-75000.00	19	1	sales	April	28	3
59	-245.00	0	0.00	19	1	sales	May	29	4
60	-272.70	0	0.00	19	2	sales	March	30	5
61	-312.00	250	-78000.00	19	2	sales	April	31	6
62	-299.00	150	-44850.00	19	2	sales	May	32	7
63	-623.89	455	-283869.95	25	1	sales	March	59	12
64	-549.68	535	-294078.80	25	1	sales	April	60	13
65	-460.00	0	0.00	25	1	sales	May	61	14
66	-542.83	120	-65139.60	25	2	sales	March	62	15
67	-559.19	320	-178940.80	25	2	sales	April	63	16
68	-489.06	20	-9781.20	25	2	sales	May	64	17
			-1281110.35						

Output 6.5.3 NODEOUT=NODE3

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_	_FBQ_
1	_ROOT_	238	0.00	22	0	8	0	3	166	-69
2	_EXCESS_	-200	-100000198.75	21	1	20	13	65	5	65
3	f1_apr_1	.	-100000277.35	3	1	6	2	4	490	4
4	f1_apr_2	.	-100000387.60	13	19	11	2	-60	235	36
5	f1_mar_1	.	-100000326.65	2	8	1	20	-21	150	1
6	f1_mar_2	.	-100000461.81	12	19	13	1	-59	455	33
7	f1_may_1	.	-100000288.85	4	1	7	3	8	65	8
8	f1_may_2	.	-100000330.98	14	17	10	1	-50	335	40
9	f2_apr_1	.	-100000288.35	6	3	4	1	14	20	14
10	f2_apr_2	.	-100000397.11	16	19	18	2	-63	280	46
11	f2_mar_1	.	-100000286.75	5	10	22	1	12	255	11
12	f2_mar_2	.	-100000380.75	15	20	19	9	44	600	43
13	f2_may_1	.	-100000304.85	7	4	9	2	18	115	18
14	f2_may_2	.	-100000356.98	17	19	14	2	-64	370	50
15	fact1_1	1000	-100000198.75	1	2	3	19	-1	290	-1
16	fact1_2	1000	-100000213.10	11	13	17	1	-36	200	-33
17	fact2_1	850	-100000198.75	10	21	5	2	-66	45	-33
18	fact2_2	1500	-100000198.75	20	21	15	10	-68	150	-65
19	shop1_1	-900	-99999999.00	8	22	2	21	0	0	21
20	shop1_2	-900	-99999854.92	18	16	12	1	57	400	53
21	shop2_1	-900	-100000005.85	9	7	21	1	32	150	27
22	shop2_2	-1450	-99999837.92	19	15	16	8	62	110	59

Example 6.6: Adding Side Constraints, Using a Warm Start

The manufacturer of Gizmo chips, which are parts needed to make televisions, can supply only 2600 chips to factory 1 and 3750 chips to factory 2 in time for production in each of the months of March and April. However, Gizmo chips will not be in short supply in May. Three chips are required to make each 19-inch TV while the 25-inch TVs require four chips each. To limit the production of televisions produced at factory 1 in March so that the TVs have the correct number of chips, a side constraint called FACT1 MAR GIZMO is used. The form of this constraint is

```
3 * prod f1 19 mar   +   4 * prod f1 25 mar   <=   2600
```

"prod f1 19 mar" is the name of the arc directed from the node fact1_1 toward node f1_mar_1 and, in the previous constraint, designates the flow assigned to this arc. The ARCDATA= and ARCOUT= data sets have arc names in a variable called _name_.

The other side constraints (shown below) are called FACT2 MAR GIZMO , FACT1 APL GIZMO, and FACT2 APL GIZMO.

```
3 * prod f2 19 mar   +   4 * prod f2 25 mar   <=   3750
3 * prod f1 19 apl   +   4 * prod f1 25 apl   <=   2600
3 * prod f2 19 apl   +   4 * prod f2 25 apl   <=   3750
```

To maintain customer goodwill, the total number of backorders is not to exceed 50 sets. The side constraint TOTAL BACKORDER that models this restriction is:

```
back f1 19 apl  +  back f1 25 apl  +
back f2 19 apl  +  back f2 25 apl  +
back f1 19 may  +  back f1 25 may  +
back f2 19 may  +  back f2 25 may   <=   50
```

The sparse CONDATA= data set format is used. All side constraints are less than or equal type. Because this is the default type value for the DEFCONTYPE= option, type information is not necessary in the following CONDATA=CON3. Also, DEFCONTYPE= <= does not have to be specified in the PROC NETFLOW statement that follows. Notice that the _column_ variable value CHIP/BO LIMIT indicates that an observation of the CON3 data set contains rhs information. Therefore, specify RHSOBS='CHIP/BO LIMIT'.

```
title 'Adding Side Constraints and Using a Warm Start';
title2 'Production Planning/Inventory/Distribution';
data con3;
  input _column_ &$14. _row_ &$15. _coef_ ;
  datalines;
prod f1 19 mar  FACT1 MAR GIZMO  3
prod f1 25 mar  FACT1 MAR GIZMO  4
CHIP/BO LIMIT   FACT1 MAR GIZMO  2600
prod f2 19 mar  FACT2 MAR GIZMO  3
prod f2 25 mar  FACT2 MAR GIZMO  4
CHIP/BO LIMIT   FACT2 MAR GIZMO  3750
prod f1 19 apl  FACT1 APL GIZMO  3
prod f1 25 apl  FACT1 APL GIZMO  4
```

```
CHIP/BO LIMIT   FACT1 APL GIZMO  2600
prod f2 19 apl  FACT2 APL GIZMO  3
prod f2 25 apl  FACT2 APL GIZMO  4
CHIP/BO LIMIT   FACT2 APL GIZMO  3750
back f1 19 apl  TOTAL BACKORDER  1
back f1 25 apl  TOTAL BACKORDER  1
back f2 19 apl  TOTAL BACKORDER  1
back f2 25 apl  TOTAL BACKORDER  1
back f1 19 may  TOTAL BACKORDER  1
back f1 25 may  TOTAL BACKORDER  1
back f2 19 may  TOTAL BACKORDER  1
back f2 25 may  TOTAL BACKORDER  1
CHIP/BO LIMIT   TOTAL BACKORDER  50
;
```

The four pairs of data sets that follow can be used as ARCDATA= and NODEDATA= data sets in the following PROC NETFLOW run. The set used depends on which cost information the arcs are to have and whether a warm start is to be used.

```
ARCDATA=arc0    NODEDATA=node0
ARCDATA=arc1    NODEDATA=node2
ARCDATA=arc2    NODEDATA=node2
ARCDATA=arc3    NODEDATA=node3
```

arc0, node0, arc1, and node2 were created in Example 6.4. The first two data sets are the original input data sets. arc1 and node2 were the ARCOUT= and NODEOUT= data sets of a PROC NETFLOW run with FUTURE1 specified. Now, if you use arc1 and node2 as the ARCDATA= data set and NODEDATA= data set in a PROC NETFLOW run, you can specify WARM, as these data sets contain additional information describing a warm start.

In Example 6.5, arc2 was created by modifying arc1 to reflect different arc costs. arc2 and node2 can also be used as the ARCDATA= and NODEDATA= data sets in a PROC NETFLOW run. Again, specify WARM, as these data sets contain additional information describing a warm start. This start, however, contains the optimal basis using the original costs.

If you are going to continue optimization using the changed arc costs, it is probably best to use arc3 and node3 as the ARCDATA= and NODEDATA= data sets. These data sets, created in Example 6.6 by PROC NETFLOW when the FUTURE1 option was specified, contain an optimal basis that can be used as a warm start.

PROC NETFLOW is used to find the changed cost network solution that obeys the chip limit and backorder side constraints. The FUTURE2 option is specified in case further processing is required. An explicit ID list has also been specified so that the variables oldcost, oldfc and oldflow do not appear in the subsequent output data sets.

```
proc netflow
  nodedata=node3 arcdata=arc3 warm
  condata=con3 sparsecondata rhsobs='CHIP/BO LIMIT'
  future2 dualout=dual4 conout=con4;
     id diagonal factory key_id mth_made;
     run;
     quit;
```

```
proc print data=con4 heading=h width=min;
sum _fcost_;
var _tail_ _head_ _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_ _fcost_;
run;

proc print data=con4 heading=h width=min;
var _rcost_ _anumb_ _tnumb_ _status_ diagonal factory key_id mth_made;
run;

proc print data=dual4;
  run;
```

The following messages appear on the SAS log:

```
NOTE: The following 3 variables in ARCDATA do not belong to any SAS variable
      list. These will be ignored.
      oldcost
      oldfc
      oldflow
NOTE: Number of nodes= 21 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 5 .
NOTE: The greater of total supply and total demand= 4350 .
NOTE: Number of iterations performed (neglecting any constraints)= 1 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= -1285086.45 .
NOTE: Number of <= side constraints= 5 .
NOTE: Number of == side constraints= 0 .
NOTE: Number of >= side constraints= 0 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 16 .
NOTE: Number of iterations, optimizing with constraints= 14 .
NOTE: Of these, 1 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= -1282708.625 .
NOTE: The data set WORK.CON4 has 68 observations and 18 variables.
NOTE: The data set WORK.DUAL4 has 27 observations and 14 variables.
```

Output 6.6.1 CONOUT=CON4

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	fact1_1	_EXCESS_	0.00	99999999	0		1000	200	5.000	0.00
2	fact2_1	_EXCESS_	0.00	99999999	0		850	200	45.000	0.00
3	fact1_2	_EXCESS_	0.00	99999999	0		1000	200	0.000	0.00
4	fact2_2	_EXCESS_	0.00	99999999	0		1500	200	150.000	0.00
5	fact1_1	f1_apr_1	78.60	600	50	prod f1 19 apl	1000	.	533.333	41920.00
6	f1_mar_1	f1_apr_1	15.00	50	0		.	.	0.000	0.00
7	f1_may_1	f1_apr_1	33.60	20	0	back f1 19 may	.	.	0.000	0.00
8	f2_apr_1	f1_apr_1	11.00	40	0		.	.	0.000	0.00
9	fact1_2	f1_apr_2	174.50	550	50	prod f1 25 apl	1000	.	250.000	43625.00
10	f1_mar_2	f1_apr_2	20.00	40	0		.	.	0.000	0.00
11	f1_may_2	f1_apr_2	49.20	15	0	back f1 25 may	.	.	0.000	0.00
12	f2_apr_2	f1_apr_2	21.00	25	0		.	.	0.000	0.00
13	fact1_1	f1_mar_1	127.90	500	50	prod f1 19 mar	1000	.	333.333	42633.33
14	f1_apr_1	f1_mar_1	33.60	20	0	back f1 19 apl	.	.	20.000	672.00
15	f2_mar_1	f1_mar_1	10.00	40	0		.	.	40.000	400.00
16	fact1_2	f1_mar_2	217.90	400	40	prod f1 25 mar	1000	.	400.000	87160.00
17	f1_apr_2	f1_mar_2	38.40	30	0	back f1 25 apl	.	.	30.000	1152.00
18	f2_mar_2	f1_mar_2	20.00	25	0		.	.	25.000	500.00
19	fact1_1	f1_may_1	90.10	400	50		1000	.	128.333	11562.83
20	f1_apr_1	f1_may_1	12.00	50	0		.	.	0.000	0.00
21	f2_may_1	f1_may_1	13.00	40	0		.	.	0.000	0.00
22	fact1_2	f1_may_2	113.30	350	40		1000	.	350.000	39655.00
23	f1_apr_2	f1_may_2	18.00	40	0		.	.	0.000	0.00
24	f2_may_2	f1_may_2	13.00	25	0		.	.	0.000	0.00
25	f1_apr_1	f2_apr_1	11.00	99999999	0		.	.	13.333	146.67
26	fact2_1	f2_apr_1	62.40	480	35	prod f2 19 apl	850	.	480.000	29952.00
27	f2_mar_1	f2_apr_1	18.00	30	0		.	.	0.000	0.00
28	f2_may_1	f2_apr_1	30.00	15	0	back f2 19 may	.	.	0.000	0.00
29	f1_apr_2	f2_apr_2	23.00	99999999	0		.	.	0.000	0.00
30	fact2_2	f2_apr_2	196.70	680	35	prod f2 25 apl	1500	.	577.500	113594.25
31	f2_mar_2	f2_apr_2	28.00	50	0		.	.	0.000	0.00
32	f2_may_2	f2_apr_2	64.80	15	0	back f2 25 may	.	.	0.000	0.00
33	f1_mar_1	f2_mar_1	11.00	99999999	0		.	.	0.000	0.00
34	fact2_1	f2_mar_1	88.00	450	35	prod f2 19 mar	850	.	290.000	25520.00
35	f2_apr_1	f2_mar_1	20.40	15	0	back f2 19 apl	.	.	0.000	0.00
36	f1_mar_2	f2_mar_2	23.00	99999999	0		.	.	0.000	0.00
37	fact2_2	f2_mar_2	182.00	650	35	prod f2 25 mar	1500	.	650.000	118300.00
38	f2_apr_2	f2_mar_2	37.20	15	0	back f2 25 apl	.	.	0.000	0.00
39	f1_may_1	f2_may_1	16.00	99999999	0		.	.	115.000	1840.00
40	fact2_1	f2_may_1	128.80	250	35		850	.	35.000	4508.00
41	f2_apr_1	f2_may_1	20.00	30	0		.	.	0.000	0.00
42	f1_may_2	f2_may_2	26.00	99999999	0		.	.	350.000	9100.00
43	fact2_2	f2_may_2	181.40	550	35		1500	.	122.500	22221.50
44	f2_apr_2	f2_may_2	38.00	50	0		.	.	0.000	0.00
45	f1_mar_1	shop1_1	-327.65	250	0		.	900	143.333	-46963.17
46	f1_apr_1	shop1_1	-300.00	250	0		.	900	250.000	-75000.00
47	f1_may_1	shop1_1	-285.00	250	0		.	900	13.333	-3800.00
48	f2_mar_1	shop1_1	-297.40	250	0		.	900	250.000	-74350.00

Output 6.6.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
49	f2_apr_1	shop1_1	-290.00	250	0		.	900	243.333	-70566.67
50	f2_may_1	shop1_1	-292.00	250	0		.	900	0.000	0.00
51	f1_mar_2	shop1_2	-559.76	99999999	0		.	900	0.000	0.00
52	f1_apr_2	shop1_2	-524.28	99999999	0		.	900	0.000	0.00
53	f1_may_2	shop1_2	-475.02	99999999	0		.	900	0.000	0.00
54	f2_mar_2	shop1_2	-567.83	500	0		.	900	500.000	-283915.00
55	f2_apr_2	shop1_2	-542.19	500	0		.	900	400.000	-216876.00
56	f2_may_2	shop1_2	-491.56	500	0		.	900	0.000	0.00
57	f1_mar_1	shop2_1	-362.74	250	0		.	900	250.000	-90685.00
58	f1_apr_1	shop2_1	-300.00	250	0		.	900	250.000	-75000.00
59	f1_may_1	shop2_1	-245.00	250	0		.	900	0.000	0.00
60	f2_mar_1	shop2_1	-272.70	250	0		.	900	0.000	0.00
61	f2_apr_1	shop2_1	-312.00	250	0		.	900	250.000	-78000.00
62	f2_may_1	shop2_1	-299.00	250	0		.	900	150.000	-44850.00
63	f1_mar_2	shop2_2	-623.89	99999999	0		.	1450	455.000	-283869.95
64	f1_apr_2	shop2_2	-549.68	99999999	0		.	1450	220.000	-120929.60
65	f1_may_2	shop2_2	-460.00	99999999	0		.	1450	0.000	0.00
66	f2_mar_2	shop2_2	-542.83	500	0		.	1450	125.000	-67853.75
67	f2_apr_2	shop2_2	-559.19	500	0		.	1450	177.500	-99256.23
68	f2_may_2	shop2_2	-519.06	500	0		.	1450	472.500	-245255.85
										-1282708.63

Output 6.6.2 CONOUT=CON4 (continued)

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
1	.	65	1	KEY_ARC BASIC	.	.		
2	.	66	10	KEY_ARC BASIC	.	.		
3	30.187	67	11	LOWERBD NONBASIC	.	.		
4	.	68	20	KEY_ARC BASIC	.	.		
5	.	4	1	KEY_ARC BASIC	19	1	production	April
6	63.650	5	2	LOWERBD NONBASIC	19	1	storage	March
7	54.650	6	4	LOWERBD NONBASIC	19	1	backorder	May
8	22.000	7	6	LOWERBD NONBASIC	19	.	f2_to_1	April
9	.	36	11	KEY_ARC BASIC	25	1	production	April
10	94.210	37	12	LOWERBD NONBASIC	25	1	storage	March
11	7.630	38	14	LOWERBD NONBASIC	25	1	backorder	May
12	30.510	39	16	LOWERBD NONBASIC	25	.	f2_to_1	April
13	.	1	1	KEY_ARC BASIC	19	1	production	March
14	.	2	3	NONKEY ARC BASIC	19	1	backorder	April
15	-34.750	3	5	UPPERBD NONBASIC	19	.	f2_to_1	March
16	-31.677	33	11	UPPERBD NONBASIC	25	1	production	March
17	-20.760	34	13	UPPERBD NONBASIC	25	1	backorder	April
18	-61.060	35	15	UPPERBD NONBASIC	25	.	f2_to_1	March
19	.	8	1	KEY_ARC BASIC	19	1	production	May
20	6.000	9	3	LOWERBD NONBASIC	19	1	storage	April
21	29.000	10	7	LOWERBD NONBASIC	19	.	f2_to_1	May
22	-11.913	40	11	UPPERBD NONBASIC	25	1	production	May
23	74.620	41	13	LOWERBD NONBASIC	25	1	storage	April
24	39.000	42	17	LOWERBD NONBASIC	25	.	f2_to_1	May
25	.	14	3	KEY_ARC BASIC	19	.	f1_to_2	April
26	-14.077	15	10	UPPERBD NONBASIC	19	2	production	April
27	10.900	16	5	LOWERBD NONBASIC	19	2	storage	March
28	56.050	17	7	LOWERBD NONBASIC	19	2	backorder	May
29	13.490	46	13	LOWERBD NONBASIC	25	.	f1_to_2	April
30	.	47	20	KEY_ARC BASIC	25	2	production	April
31	11.640	48	15	LOWERBD NONBASIC	25	2	storage	March
32	39.720	49	17	LOWERBD NONBASIC	25	2	backorder	May
33	55.750	11	2	LOWERBD NONBASIC	19	.	f1_to_2	March
34	.	12	10	KEY_ARC BASIC	19	2	production	March
35	42.550	13	6	LOWERBD NONBASIC	19	2	backorder	April
36	104.060	43	12	LOWERBD NONBASIC	25	.	f1_to_2	March
37	-23.170	44	20	UPPERBD NONBASIC	25	2	production	March
38	68.610	45	16	LOWERBD NONBASIC	25	2	backorder	April
39	.	18	4	KEY_ARC BASIC	19	.	f1_to_2	May
40	22.700	19	10	LOWERBD NONBASIC	19	2	production	May
41	9.000	20	6	LOWERBD NONBASIC	19	2	storage	April
42	.	50	14	KEY_ARC BASIC	25	.	f1_to_2	May
43	.	51	20	NONKEY ARC BASIC	25	2	production	May
44	78.130	52	16	LOWERBD NONBASIC	25	2	storage	April
45	.	21	2	KEY_ARC BASIC	19	1	sales	March
46	-21.000	22	3	UPPERBD NONBASIC	19	1	sales	April
47	.	23	4	NONKEY ARC BASIC	19	1	sales	May
48	-14.500	24	5	UPPERBD NONBASIC	19	2	sales	March

Output 6.6.2 *continued*

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
49	.	25	6	NONKEY ARC BASIC	19	2	sales	April
50	9.000	26	7	LOWERBD NONBASIC	19	2	sales	May
51	47.130	53	12	LOWERBD NONBASIC	25	1	sales	March
52	8.400	54	13	LOWERBD NONBASIC	25	1	sales	April
53	1.040	55	14	LOWERBD NONBASIC	25	1	sales	May
54	-42.000	56	15	UPPERBD NONBASIC	25	2	sales	March
55	.	57	16	KEY_ARC BASIC	25	2	sales	April
56	10.500	58	17	LOWERBD NONBASIC	25	2	sales	May
57	-37.090	27	2	UPPERBD NONBASIC	19	1	sales	March
58	-23.000	28	3	UPPERBD NONBASIC	19	1	sales	April
59	38.000	29	4	LOWERBD NONBASIC	19	1	sales	May
60	8.200	30	5	LOWERBD NONBASIC	19	2	sales	March
61	-24.000	31	6	UPPERBD NONBASIC	19	2	sales	April
62	.	32	7	KEY_ARC BASIC	19	2	sales	May
63	.	59	12	KEY_ARC BASIC	25	1	sales	March
64	.	60	13	KEY_ARC BASIC	25	1	sales	April
65	33.060	61	14	LOWERBD NONBASIC	25	1	sales	May
66	.	62	15	KEY_ARC BASIC	25	2	sales	March
67	.	63	16	KEY_ARC BASIC	25	2	sales	April
68	.	64	17	KEY_ARC BASIC	25	2	sales	May

Output 6.6.3 DUALOUT=DUAL4

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_
1	_ROOT_	238	0.00	22	0	8	5	3
2	_EXCESS_	-200	-100000193.90	21	1	20	13	65
3	f1_apr_1	.	-100000278.00	3	1	6	2	4
4	f1_apr_2	.	-100000405.92	13	19	11	2	-60
5	f1_mar_1	.	-100000326.65	2	8	1	20	-21
6	f1_mar_2	.	-100000480.13	12	19	13	1	-59
7	f1_may_1	.	-100000284.00	4	1	7	3	8
8	f1_may_2	.	-100000349.30	14	17	15	1	-50
9	f2_apr_1	.	-100000289.00	6	3	4	1	14
10	f2_apr_2	.	-100000415.43	16	20	18	9	47
11	f2_mar_1	.	-100000281.90	5	10	3	1	12
12	f2_mar_2	.	-100000399.07	15	19	10	1	-62
13	f2_may_1	.	-100000300.00	7	4	9	2	18
14	f2_may_2	.	-100000375.30	17	19	14	2	-64
15	fact1_1	1000	-100000193.90	1	2	21	19	-1
16	fact1_2	1000	-100000224.09	11	13	17	1	-36
17	fact2_1	850	-100000193.90	10	21	5	2	-66
18	fact2_2	1500	-100000193.90	20	21	16	10	-68
19	shop1_1	-900	-99999999.00	8	22	2	21	0
20	shop1_2	-900	-99999873.24	18	16	19	1	57
21	shop2_1	-900	-100000001.00	9	7	22	1	32
22	shop2_2	-1450	-99999856.24	19	16	12	7	63
23		.	-1.83	2	8	.	.	25

Obs	_FLOW_	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
1	166.000	-69	0	75		
2	5.000	65	.	.		
3	483.333	4	.	.		
4	220.000	36	.	.		
5	143.333	1	.	.		
6	455.000	33	.	.		
7	78.333	8	.	.		
8	350.000	40	.	.		
9	13.333	14	.	.		
10	542.500	46	.	.		
11	255.000	11	.	.		
12	125.000	43	.	.		
13	115.000	18	.	.		
14	472.500	50	.	.		
15	283.333	-1	.	.		
16	200.000	-33	.	.		
17	45.000	-33	.	.		
18	150.000	-65	.	.		
19	0.000	21	.	.		
20	400.000	53	.	.		
21	150.000	27	.	.		
22	177.500	59	.	.		
23	243.333	.	2600	2600	LE	FACT1 APL GIZMO

Output 6.6.3 *continued*

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_
24		.	-1.62	0	8	.	.	23
25		.	-6.21	3	17	.	.	51
26		.	0.00	1	1	.	1	.
27		.	-15.05	4	2	.	.	2

Obs	_FLOW_	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
24	13.333	.	2600	2600	LE	FACT1 MAR GIZMO
25	87.500	.	3750	3750	LE	FACT2 APL GIZMO
26	280.000	.	3470	3750	LE	FACT2 MAR GIZMO
27	20.000	.	50	50	LE	TOTAL BACKORDER

Example 6.7: Using a Constrained Solution Warm Start

Suppose the 25-inch screen TVs produced at factory 1 in May can be sold at either shop with an increased profit of 40 dollars each. What is the new optimal solution? Because only arc costs have been changed, information about the present solution in DUALOUT=dual4 and CONOUT=con4 can be used as a warm start in the following PROC NETFLOW run. It is still necessary to specify CONDATA=con3 SPARSECONDATA RHSOBS='CHIP/BO LIMIT', since the CONDATA= data set is always read.

```
  title 'Using a Constrained Solution Warm Start';
  title2 'Production Planning/Inventory/Distribution';
  data new_con4;
     set con4;
     oldcost=_cost_;
     oldflow=_flow_;
     oldfc=_fcost_;
     if _tail_='f1_may_2'
        & (_head_='shop1_2' | _head_='shop2_2')
        then _cost_=_cost_-40;
     run;

proc netflow
   warm
   arcdata=new_con4
   dualin=dual4
   condata=con3
   sparsecondata
   rhsobs='CHIP/BO LIMIT'
   dualout=dual5
   conout=con5;
run;
 quit;

proc print data=con5 heading=h width=min;
sum _fcost_;
var _tail_ _head_ _capac_ _lo_ _supply_ _demand_ _name_ _cost_ _flow_ _fcost_;
run;
```

```
proc print data=con5 heading=h width=min;
   sum oldfc;
   var oldcost oldflow oldfc diagonal factory key_id mth_made _anumb_ _tnumb_;
run;

proc print data=dual5;
run;
```

The following messages appear on the SAS log:

```
NOTE: The following 1 variables in NODEDATA do not belong to any SAS variable
      list. These will be ignored.
      _VALUE_
NOTE: Number of nodes= 21 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 5 .
NOTE: The greater of total supply and total demand= 4350 .
NOTE: Number of <= side constraints= 5 .
NOTE: Number of == side constraints= 0 .
NOTE: Number of >= side constraints= 0 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 16 .
NOTE: Number of iterations, optimizing with constraints= 6 .
NOTE: Of these, 0 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= -1295661.8 .
NOTE: The data set WORK.CON5 has 68 observations and 21 variables.
NOTE: The data set WORK.DUAL5 has 25 observations and 14 variables.
```

Output 6.7.1 CONOUT=CON5

Obs	_tail_	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_	_FCOST_
1	fact1_1	_EXCESS_	99999999	0	1000	200		0.00	5.000	0.00
2	fact2_1	_EXCESS_	99999999	0	850	200		0.00	45.000	0.00
3	fact1_2	_EXCESS_	99999999	0	1000	200		0.00	0.000	0.00
4	fact2_2	_EXCESS_	99999999	0	1500	200		0.00	150.000	0.00
5	fact1_1	f1_apr_1	600	50	1000	.	prod f1 19 apl	78.60	533.333	41920.00
6	f1_mar_1	f1_apr_1	50	0	.	.		15.00	0.000	0.00
7	f1_may_1	f1_apr_1	20	0	.	.	back f1 19 may	33.60	0.000	0.00
8	f2_apr_1	f1_apr_1	40	0	.	.		11.00	0.000	0.00
9	fact1_2	f1_apr_2	550	50	1000	.	prod f1 25 apl	174.50	250.000	43625.00
10	f1_mar_2	f1_apr_2	40	0	.	.		20.00	0.000	0.00
11	f1_may_2	f1_apr_2	15	0	.	.	back f1 25 may	49.20	0.000	0.00
12	f2_apr_2	f1_apr_2	25	0	.	.		21.00	0.000	0.00
13	fact1_1	f1_mar_1	500	50	1000	.	prod f1 19 mar	127.90	333.333	42633.33
14	f1_apr_1	f1_mar_1	20	0	.	.	back f1 19 apl	33.60	20.000	672.00
15	f2_mar_1	f1_mar_1	40	0	.	.		10.00	40.000	400.00
16	fact1_2	f1_mar_2	400	40	1000	.	prod f1 25 mar	217.90	400.000	87160.00
17	f1_apr_2	f1_mar_2	30	0	.	.	back f1 25 apl	38.40	30.000	1152.00
18	f2_mar_2	f1_mar_2	25	0	.	.		20.00	25.000	500.00
19	fact1_1	f1_may_1	400	50	1000	.		90.10	128.333	11562.83
20	f1_apr_1	f1_may_1	50	0	.	.		12.00	0.000	0.00
21	f2_may_1	f1_may_1	40	0	.	.		13.00	0.000	0.00
22	fact1_2	f1_may_2	350	40	1000	.		113.30	350.000	39655.00
23	f1_apr_2	f1_may_2	40	0	.	.		18.00	0.000	0.00
24	f2_may_2	f1_may_2	25	0	.	.		13.00	0.000	0.00
25	f1_apr_1	f2_apr_1	99999999	0	.	.		11.00	13.333	146.67
26	fact2_1	f2_apr_1	480	35	850	.	prod f2 19 apl	62.40	480.000	29952.00
27	f2_mar_1	f2_apr_1	30	0	.	.		18.00	0.000	0.00
28	f2_may_1	f2_apr_1	15	0	.	.	back f2 19 may	30.00	0.000	0.00
29	f1_apr_2	f2_apr_2	99999999	0	.	.		23.00	0.000	0.00
30	fact2_2	f2_apr_2	680	35	1500	.	prod f2 25 apl	196.70	550.000	108185.00
31	f2_mar_2	f2_apr_2	50	0	.	.		28.00	0.000	0.00
32	f2_may_2	f2_apr_2	15	0	.	.	back f2 25 may	64.80	0.000	0.00
33	f1_mar_1	f2_mar_1	99999999	0	.	.		11.00	0.000	0.00
34	fact2_1	f2_mar_1	450	35	850	.	prod f2 19 mar	88.00	290.000	25520.00
35	f2_apr_1	f2_mar_1	15	0	.	.	back f2 19 apl	20.40	0.000	0.00
36	f1_mar_2	f2_mar_2	99999999	0	.	.		23.00	0.000	0.00
37	fact2_2	f2_mar_2	650	35	1500	.	prod f2 25 mar	182.00	650.000	118300.00
38	f2_apr_2	f2_mar_2	15	0	.	.	back f2 25 apl	37.20	0.000	0.00
39	f1_may_1	f2_may_1	99999999	0	.	.		16.00	115.000	1840.00
40	fact2_1	f2_may_1	250	35	850	.		128.80	35.000	4508.00
41	f2_apr_1	f2_may_1	30	0	.	.		20.00	0.000	0.00
42	f1_may_2	f2_may_2	99999999	0	.	.		26.00	0.000	0.00
43	fact2_2	f2_may_2	550	35	1500	.		181.40	150.000	27210.00
44	f2_apr_2	f2_may_2	50	0	.	.		38.00	0.000	0.00
45	f1_mar_1	shop1_1	250	0	.	900		-327.65	143.333	-46963.17
46	f1_apr_1	shop1_1	250	0	.	900		-300.00	250.000	-75000.00
47	f1_may_1	shop1_1	250	0	.	900		-285.00	13.333	-3800.00
48	f2_mar_1	shop1_1	250	0	.	900		-297.40	250.000	-74350.00

Output 6.7.1 *continued*

Obs	_tail_	_head_	_capac_	_lo_	_SUPPLY_	_DEMAND_	_name_	_cost_	_FLOW_	_FCOST_
49	f2_apr_1	shop1_1	250	0	.	900		-290.00	243.333	-70566.67
50	f2_may_1	shop1_1	250	0	.	900		-292.00	0.000	0.00
51	f1_mar_2	shop1_2	99999999	0	.	900		-559.76	0.000	0.00
52	f1_apr_2	shop1_2	99999999	0	.	900		-524.28	0.000	0.00
53	f1_may_2	shop1_2	99999999	0	.	900		-515.02	350.000	-180257.00
54	f2_mar_2	shop1_2	500	0	.	900		-567.83	500.000	-283915.00
55	f2_apr_2	shop1_2	500	0	.	900		-542.19	50.000	-27109.50
56	f2_may_2	shop1_2	500	0	.	900		-491.56	0.000	0.00
57	f1_mar_1	shop2_1	250	0	.	900		-362.74	250.000	-90685.00
58	f1_apr_1	shop2_1	250	0	.	900		-300.00	250.000	-75000.00
59	f1_may_1	shop2_1	250	0	.	900		-245.00	0.000	0.00
60	f2_mar_1	shop2_1	250	0	.	900		-272.70	0.000	0.00
61	f2_apr_1	shop2_1	250	0	.	900		-312.00	250.000	-78000.00
62	f2_may_1	shop2_1	250	0	.	900		-299.00	150.000	-44850.00
63	f1_mar_2	shop2_2	99999999	0	.	1450		-623.89	455.000	-283869.95
64	f1_apr_2	shop2_2	99999999	0	.	1450		-549.68	220.000	-120929.60
65	f1_may_2	shop2_2	99999999	0	.	1450		-500.00	0.000	0.00
66	f2_mar_2	shop2_2	500	0	.	1450		-542.83	125.000	-67853.75
67	f2_apr_2	shop2_2	500	0	.	1450		-559.19	500.000	-279595.00
68	f2_may_2	shop2_2	500	0	.	1450		-519.06	150.000	-77859.00
										-1295661.80

Output 6.7.2 CONOUT=CON5 (continued)

Obs	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made	_ANUMB_	_TNUMB_
1	0.00	5.000	0.00	.	.			65	1
2	0.00	45.000	0.00	.	.			66	10
3	0.00	0.000	0.00	.	.			67	11
4	0.00	150.000	0.00	.	.			68	20
5	78.60	533.333	41920.00	19	1	production	April	4	1
6	15.00	0.000	0.00	19	1	storage	March	5	2
7	33.60	0.000	0.00	19	1	backorder	May	6	4
8	11.00	0.000	0.00	19	.	f2_to_1	April	7	6
9	174.50	250.000	43625.00	25	1	production	April	36	11
10	20.00	0.000	0.00	25	1	storage	March	37	12
11	49.20	0.000	0.00	25	1	backorder	May	38	14
12	21.00	0.000	0.00	25	.	f2_to_1	April	39	16
13	127.90	333.333	42633.33	19	1	production	March	1	1
14	33.60	20.000	672.00	19	1	backorder	April	2	3
15	10.00	40.000	400.00	19	.	f2_to_1	March	3	5
16	217.90	400.000	87160.00	25	1	production	March	33	11
17	38.40	30.000	1152.00	25	1	backorder	April	34	13
18	20.00	25.000	500.00	25	.	f2_to_1	March	35	15
19	90.10	128.333	11562.83	19	1	production	May	8	1
20	12.00	0.000	0.00	19	1	storage	April	9	3
21	13.00	0.000	0.00	19	.	f2_to_1	May	10	7
22	113.30	350.000	39655.00	25	1	production	May	40	11
23	18.00	0.000	0.00	25	1	storage	April	41	13
24	13.00	0.000	0.00	25	.	f2_to_1	May	42	17
25	11.00	13.333	146.67	19	.	f1_to_2	April	14	3
26	62.40	480.000	29952.00	19	2	production	April	15	10
27	18.00	0.000	0.00	19	2	storage	March	16	5
28	30.00	0.000	0.00	19	2	backorder	May	17	7
29	23.00	0.000	0.00	25	.	f1_to_2	April	46	13
30	196.70	577.500	113594.25	25	2	production	April	47	20
31	28.00	0.000	0.00	25	2	storage	March	48	15
32	64.80	0.000	0.00	25	2	backorder	May	49	17
33	11.00	0.000	0.00	19	.	f1_to_2	March	11	2
34	88.00	290.000	25520.00	19	2	production	March	12	10
35	20.40	0.000	0.00	19	2	backorder	April	13	6
36	23.00	0.000	0.00	25	.	f1_to_2	March	43	12
37	182.00	650.000	118300.00	25	2	production	March	44	20
38	37.20	0.000	0.00	25	2	backorder	April	45	16
39	16.00	115.000	1840.00	19	.	f1_to_2	May	18	4
40	128.80	35.000	4508.00	19	2	production	May	19	10
41	20.00	0.000	0.00	19	2	storage	April	20	6
42	26.00	350.000	9100.00	25	.	f1_to_2	May	50	14
43	181.40	122.500	22221.50	25	2	production	May	51	20
44	38.00	0.000	0.00	25	2	storage	April	52	16
45	-327.65	143.333	-46963.17	19	1	sales	March	21	2
46	-300.00	250.000	-75000.00	19	1	sales	April	22	3
47	-285.00	13.333	-3800.00	19	1	sales	May	23	4
48	-297.40	250.000	-74350.00	19	2	sales	March	24	5

Output 6.7.2 *continued*

Obs	oldcost	oldflow	oldfc	diagonal	factory	key_id	mth_made	_ANUMB_	_TNUMB_
49	-290.00	243.333	-70566.67	19	2	sales	April	25	6
50	-292.00	0.000	0.00	19	2	sales	May	26	7
51	-559.76	0.000	0.00	25	1	sales	March	53	12
52	-524.28	0.000	0.00	25	1	sales	April	54	13
53	-475.02	0.000	0.00	25	1	sales	May	55	14
54	-567.83	500.000	-283915.00	25	2	sales	March	56	15
55	-542.19	400.000	-216876.00	25	2	sales	April	57	16
56	-491.56	0.000	0.00	25	2	sales	May	58	17
57	-362.74	250.000	-90685.00	19	1	sales	March	27	2
58	-300.00	250.000	-75000.00	19	1	sales	April	28	3
59	-245.00	0.000	0.00	19	1	sales	May	29	4
60	-272.70	0.000	0.00	19	2	sales	March	30	5
61	-312.00	250.000	-78000.00	19	2	sales	April	31	6
62	-299.00	150.000	-44850.00	19	2	sales	May	32	7
63	-623.89	455.000	-283869.95	25	1	sales	March	59	12
64	-549.68	220.000	-120929.60	25	1	sales	April	60	13
65	-460.00	0.000	0.00	25	1	sales	May	61	14
66	-542.83	125.000	-67853.75	25	2	sales	March	62	15
67	-559.19	177.500	-99256.23	25	2	sales	April	63	16
68	-519.06	472.500	-245255.85	25	2	sales	May	64	17
			-1282708.63						

Output 6.7.3 DUALOUT=DUAL5

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_
1	f1_apr_1	.	-100000278.00	3	1	6	2	4	483.333
2	f1_apr_2	.	-100000405.92	13	19	11	2	-60	220.000
3	f1_mar_1	.	-100000326.65	2	8	1	20	-21	143.333
4	f1_mar_2	.	-100000480.13	12	19	13	1	-59	455.000
5	f1_may_1	.	-100000284.00	4	1	7	3	8	78.333
6	f1_may_2	.	-100000363.43	14	18	10	1	-55	350.000
7	f2_apr_1	.	-100000289.00	6	3	4	1	14	13.333
8	f2_apr_2	.	-100000390.60	16	20	18	3	47	515.000
9	f2_mar_1	.	-100000281.90	5	10	3	1	12	255.000
10	f2_mar_2	.	-100000399.07	15	19	16	1	-62	125.000
11	f2_may_1	.	-100000300.00	7	4	9	2	18	115.000
12	f2_may_2	.	-100000375.30	17	20	19	6	51	115.000
13	fact1_1	1000	-100000193.90	1	2	21	19	-1	283.333
14	fact1_2	1000	-100000224.09	11	13	15	1	-36	200.000
15	fact2_1	850	-100000193.90	10	21	5	2	-66	45.000
16	fact2_2	1500	-100000193.90	20	21	17	10	-68	150.000
17	shop1_1	-900	-99999999.00	8	22	2	21	0	0.000
18	shop1_2	-900	-99999848.41	18	16	14	2	57	50.000
19	shop2_1	-900	-100000001.00	9	7	22	1	32	150.000
20	shop2_2	-1450	-99999856.24	19	17	12	5	64	150.000
21		.	-1.83	2	8	.	.	25	243.333
22		.	-1.62	0	8	.	.	23	13.333
23		.	0.00	3	3	.	3	.	110.000

Obs	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
1	4	.	.		
2	36	.	.		
3	1	.	.		
4	33	.	.		
5	8	.	.		
6	40	.	.		
7	14	.	.		
8	46	.	.		
9	11	.	.		
10	43	.	.		
11	18	.	.		
12	50	.	.		
13	-1	.	.		
14	-33	.	.		
15	-33	.	.		
16	-65	.	.		
17	21	.	.		
18	53	.	.		
19	27	.	.		
20	59	.	.		
21	.	2600	2600	LE	FACT1 APL GIZMO
22	.	2600	2600	LE	FACT1 MAR GIZMO
23	.	3640	3750	LE	FACT2 APL GIZMO

Output 6.7.3 *continued*

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_
24		.	0.00	1	1	.	1	.	280.000
25		.	-15.05	4	2	.	.	2	20.000

Obs	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
24	.	3470	3750	LE	FACT2 MAR GIZMO
25	.	50	50	LE	TOTAL BACKORDER

Example 6.8: Nonarc Variables in the Side Constraints

Notice in DUALOUT=dual5 from Example 6.7 the FACT2 MAR GIZMO constraint (observation 24) has a _VALUE_ of 3470, which is not equal to the _RHS_ of this constraint. Not all of the 3750 chips that can be supplied to factory 2 for March production are used. It is suggested that all the possible chips be obtained in March and those not used be saved for April production. Because chips must be kept in an air-controlled environment, it costs 1 dollar to store each chip purchased in March until April. The maximum number of chips that can be stored in this environment at each factory is 150. In addition, a search of the parts inventory at factory 1 turned up 15 chips available for their March production.

Nonarc variables are used in the side constraints that handle the limitations of supply of Gizmo chips. A nonarc variable called “f1 unused chips” has as a value the number of chips that are not used at factory 1 in March. Another nonarc variable, “f2 unused chips”, has as a value the number of chips that are not used at factory 2 in March. “f1 chips from mar” has as a value the number of chips left over from March used for production at factory 1 in April. Similarly, “f2 chips from mar” has as a value the number of chips left over from March used for April production at factory 2 in April. The last two nonarc variables have objective function coefficients of 1 and upper bounds of 150. The Gizmo side constraints are

```
3*prod f1 19 mar + 4*prod f1 25 mar + f1 unused chips = 2615
3*prod f2 19 apl + 4*prod f2 25 apl + f2 unused chips = 3750
3*prod f1 19 apl + 4*prod f1 25 apl - f1 chips from mar = 2600
3*prod f2 19 apl + 4*prod f2 25 apl - f2 chips from mar = 3750
f1 unused chips + f2 unused chips -
f1 chips from mar - f2 chips from mar >= 0
```

The last side constraint states that the number of chips not used in March is not less than the number of chips left over from March and used in April. Here, this constraint is called CHIP LEFTOVER.

The following SAS code creates a new data set containing constraint data. It seems that most of the constraints are now equalities, so you specify DEFCONTYPE=EQ in the PROC NETFLOW statements from now on and provide constraint type data for constraints that are not “equal to” type, using the default TYPEOBS value _TYPE_ as the _COLUMN_ variable value to indicate observations that contain constraint type data. Also, from now on, the default RHSOBS value is used.

```
title 'Nonarc Variables in the Side Constraints';
title2 'Production Planning/Inventory/Distribution';
data con6;
  input _column_ &$17. _row_ &$15. _coef_ ;
  datalines;
prod f1 19 mar     FACT1 MAR GIZMO  3
```

```
prod f1 25 mar     FACT1 MAR GIZMO  4
f1 unused chips    FACT1 MAR GIZMO  1
_RHS_              FACT1 MAR GIZMO  2615
prod f2 19 mar     FACT2 MAR GIZMO  3
prod f2 25 mar     FACT2 MAR GIZMO  4
f2 unused chips    FACT2 MAR GIZMO  1
_RHS_              FACT2 MAR GIZMO  3750
prod f1 19 apl     FACT1 APL GIZMO  3
prod f1 25 apl     FACT1 APL GIZMO  4
f1 chips from mar  FACT1 APL GIZMO  -1
_RHS_              FACT1 APL GIZMO  2600
prod f2 19 apl     FACT2 APL GIZMO  3
prod f2 25 apl     FACT2 APL GIZMO  4
f2 chips from mar  FACT2 APL GIZMO  -1
_RHS_              FACT2 APL GIZMO  3750
f1 unused chips    CHIP LEFTOVER    1
f2 unused chips    CHIP LEFTOVER    1
f1 chips from mar  CHIP LEFTOVER    -1
f2 chips from mar  CHIP LEFTOVER    -1
_TYPE_             CHIP LEFTOVER    1
back f1 19 apl     TOTAL BACKORDER  1
back f1 25 apl     TOTAL BACKORDER  1
back f2 19 apl     TOTAL BACKORDER  1
back f2 25 apl     TOTAL BACKORDER  1
back f1 19 may     TOTAL BACKORDER  1
back f1 25 may     TOTAL BACKORDER  1
back f2 19 may     TOTAL BACKORDER  1
back f2 25 may     TOTAL BACKORDER  1
_TYPE_             TOTAL BACKORDER  -1
_RHS_              TOTAL BACKORDER  50
;
```

The nonarc variables “f1 chips from mar” and “f2 chips from mar” have objective function coefficients of 1 and upper bounds of 150. There are various ways in which this information can be furnished to PROC NETFLOW. If there were a TYPE list variable in the CONDATA= data set, observations could be in the form

```
_COLUMN_           _TYPE_  _ROW_ _COEF_
f1 chips from mar  objfn     .       1
f1 chips from mar  upperbd   .     150
f2 chips from mar  objfn     .       1
f2 chips from mar  upperbd   .     150
```

It is desirable to assign ID list variable values to all the nonarc variables:

```
data arc6;
  set con5;
  drop oldcost oldfc oldflow _flow_ _fcost_ _status_ _rcost_;
data arc6_b;
  length key_id $10;
  input _name_ &$17. _cost_ _capac_ factory key_id $ ;
  datalines;
f1 unused chips    .    . 1 chips
f2 unused chips    .    . 2 chips
f1 chips from mar  1 150 1 chips
f2 chips from mar  1 150 2 chips
;

proc append force nowarn
   base=arc6 data=arc6_b;
run;

proc netflow
   nodedata=node0 arcdata=arc6
   condata=con6 defcontype=eq sparsecondata
   dualout=dual7 conout=con7;
run;

print nonarcs/short;
```

The following messages appear on the SAS log:

```
NOTE: Number of nodes= 21 .
NOTE: Number of supply nodes= 4 .
NOTE: Number of demand nodes= 5 .
NOTE: Total supply= 4350 , total demand= 4350 .
NOTE: Number of arcs= 68 .
NOTE: Number of nonarc variables= 4 .
NOTE: Number of iterations performed (neglecting any constraints)= 69 .
NOTE: Of these, 1 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= -1295730.8 .
NOTE: Number of <= side constraints= 1 .
NOTE: Number of == side constraints= 4 .
NOTE: Number of >= side constraints= 1 .
NOTE: Number of arc and nonarc variable side constraint coefficients= 24 .
NOTE: Number of iterations, optimizing with constraints= 13 .
NOTE: Of these, 2 were degenerate.
NOTE: Optimum reached.
NOTE: Minimal total cost= -1295542.742 .
NOTE: The data set WORK.CON7 has 68 observations and 18 variables.
NOTE: The data set WORK.DUAL7 has 26 observations and 14 variables.
```

The output in Output 6.8.1 is produced by

print *nonarcs/short* **;**

Output 6.8.1 Output of PRINT NONARCS/SHORT;

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

The NETFLOW Procedure

_N_	_name_	_cost_	_capac_	_lo_	_VALUE_
1	f1 chips from mar	1	150	0	20
2	f1 unused chips	0	99999999	0	0
3	f2 chips from mar	1	150	0	0
4	f2 unused chips	0	99999999	0	280

The optimal solution data sets, CONOUT=CON7 in Output 6.8.2 and Output 6.8.3 and DUALOUT=DUAL7 in Output 6.8.4 follow.

```
proc print data=con7;
   sum _fcost_;
proc print data=dual7;
```

Output 6.8.2 CONOUT=CON7

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	fact1_1	f1_apr_1	78.60	600	50	prod f1 19 apl	1000	.	540.000	42444.00
2	f1_mar_1	f1_apr_1	15.00	50	0		.	.	0.000	0.00
3	f1_may_1	f1_apr_1	33.60	20	0	back f1 19 may	.	.	0.000	0.00
4	f2_apr_1	f1_apr_1	11.00	40	0		.	.	0.000	0.00
5	fact1_2	f1_apr_2	174.50	550	50	prod f1 25 apl	1000	.	250.000	43625.00
6	f1_mar_2	f1_apr_2	20.00	40	0		.	.	0.000	0.00
7	f1_may_2	f1_apr_2	49.20	15	0	back f1 25 may	.	.	0.000	0.00
8	f2_apr_2	f1_apr_2	21.00	25	0		.	.	25.000	525.00
9	fact1_1	f1_mar_1	127.90	500	50	prod f1 19 mar	1000	.	338.333	43272.83
10	f1_apr_1	f1_mar_1	33.60	20	0	back f1 19 apl	.	.	20.000	672.00
11	f2_mar_1	f1_mar_1	10.00	40	0		.	.	40.000	400.00
12	fact1_2	f1_mar_2	217.90	400	40	prod f1 25 mar	1000	.	400.000	87160.00
13	f1_apr_2	f1_mar_2	38.40	30	0	back f1 25 apl	.	.	30.000	1152.00
14	f2_mar_2	f1_mar_2	20.00	25	0		.	.	25.000	500.00
15	fact1_1	f1_may_1	90.10	400	50		1000	.	116.667	10511.67
16	f1_apr_1	f1_may_1	12.00	50	0		.	.	0.000	0.00
17	f2_may_1	f1_may_1	13.00	40	0		.	.	0.000	0.00
18	fact1_2	f1_may_2	113.30	350	40		1000	.	350.000	39655.00
19	f1_apr_2	f1_may_2	18.00	40	0		.	.	0.000	0.00
20	f2_may_2	f1_may_2	13.00	25	0		.	.	0.000	0.00
21	f1_apr_1	f2_apr_1	11.00	99999999	0		.	.	20.000	220.00
22	fact2_1	f2_apr_1	62.40	480	35	prod f2 19 apl	850	.	480.000	29952.00
23	f2_mar_1	f2_apr_1	18.00	30	0		.	.	0.000	0.00
24	f2_may_1	f2_apr_1	30.00	15	0	back f2 19 may	.	.	0.000	0.00
25	f1_apr_2	f2_apr_2	23.00	99999999	0		.	.	0.000	0.00
26	fact2_2	f2_apr_2	196.70	680	35	prod f2 25 apl	1500	.	577.500	113594.25
27	f2_mar_2	f2_apr_2	28.00	50	0		.	.	0.000	0.00
28	f2_may_2	f2_apr_2	64.80	15	0	back f2 25 may	.	.	0.000	0.00
29	f1_mar_1	f2_mar_1	11.00	99999999	0		.	.	0.000	0.00
30	fact2_1	f2_mar_1	88.00	450	35	prod f2 19 mar	850	.	290.000	25520.00
31	f2_apr_1	f2_mar_1	20.40	15	0	back f2 19 apl	.	.	0.000	0.00
32	f1_mar_2	f2_mar_2	23.00	99999999	0		.	.	0.000	0.00
33	fact2_2	f2_mar_2	182.00	650	35	prod f2 25 mar	1500	.	650.000	118300.00
34	f2_apr_2	f2_mar_2	37.20	15	0	back f2 25 apl	.	.	0.000	0.00
35	f1_may_1	f2_may_1	16.00	99999999	0		.	.	115.000	1840.00
36	fact2_1	f2_may_1	128.80	250	35		850	.	35.000	4508.00
37	f2_apr_1	f2_may_1	20.00	30	0		.	.	0.000	0.00
38	f1_may_2	f2_may_2	26.00	99999999	0		.	.	0.000	0.00
39	fact2_2	f2_may_2	181.40	550	35		1500	.	122.500	22221.50
40	f2_apr_2	f2_may_2	38.00	50	0		.	.	0.000	0.00
41	f1_mar_1	shop1_1	-327.65	250	0		.	900	148.333	-48601.42
42	f1_apr_1	shop1_1	-300.00	250	0		.	900	250.000	-75000.00
43	f1_may_1	shop1_1	-285.00	250	0		.	900	1.667	-475.00
44	f2_mar_1	shop1_1	-297.40	250	0		.	900	250.000	-74350.00
45	f2_apr_1	shop1_1	-290.00	250	0		.	900	250.000	-72500.00
46	f2_may_1	shop1_1	-292.00	250	0		.	900	0.000	0.00

Output 6.8.2 *continued*

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
47	f1_mar_2	shop1_2	-559.76	99999999	0		.	900	0.000	0.00
48	f1_apr_2	shop1_2	-524.28	99999999	0		.	900	0.000	0.00
49	f1_may_2	shop1_2	-515.02	99999999	0		.	900	347.500	-178969.45
50	f2_mar_2	shop1_2	-567.83	500	0		.	900	500.000	-283915.00
51	f2_apr_2	shop1_2	-542.19	500	0		.	900	52.500	-28464.98
52	f2_may_2	shop1_2	-491.56	500	0		.	900	0.000	0.00
53	f1_mar_1	shop2_1	-362.74	250	0		.	900	250.000	-90685.00
54	f1_apr_1	shop2_1	-300.00	250	0		.	900	250.000	-75000.00
55	f1_may_1	shop2_1	-245.00	250	0		.	900	0.000	0.00
56	f2_mar_1	shop2_1	-272.70	250	0		.	900	0.000	0.00
57	f2_apr_1	shop2_1	-312.00	250	0		.	900	250.000	-78000.00
58	f2_may_1	shop2_1	-299.00	250	0		.	900	150.000	-44850.00
59	f1_mar_2	shop2_2	-623.89	99999999	0		.	1450	455.000	-283869.95
60	f1_apr_2	shop2_2	-549.68	99999999	0		.	1450	245.000	-134671.60
61	f1_may_2	shop2_2	-500.00	99999999	0		.	1450	2.500	-1250.00
62	f2_mar_2	shop2_2	-542.83	500	0		.	1450	125.000	-67853.75
63	f2_apr_2	shop2_2	-559.19	500	0		.	1450	500.000	-279595.00
64	f2_may_2	shop2_2	-519.06	500	0		.	1450	122.500	-63584.85
65			1.00	150	0	f1 chips from mar	.	.	20.000	20.00
66			0.00	99999999	0	f1 unused chips	.	.	0.000	0.00
67			1.00	150	0	f2 chips from mar	.	.	0.000	0.00
68			0.00	99999999	0	f2 unused chips	.	.	280.000	0.00
										-1295542.74

Output 6.8.3 CONOUT=CON7 (continued)

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
1	.	5	1	KEY_ARC BASIC	19	1	production	April
2	66.150	6	7	LOWERBD NONBASIC	19	1	storage	March
3	42.580	7	8	LOWERBD NONBASIC	19	1	backorder	May
4	22.000	8	9	LOWERBD NONBASIC	19	.	f2_to_1	April
5	.	19	4	KEY_ARC BASIC	25	1	production	April
6	94.210	20	11	LOWERBD NONBASIC	25	1	storage	March
7	.	21	12	NONKEY ARC BASIC	25	1	backorder	May
8	-1.510	22	13	UPPERBD NONBASIC	25	.	f2_to_1	April
9	.	9	1	KEY_ARC BASIC	19	1	production	March
10	-17.070	10	6	UPPERBD NONBASIC	19	1	backorder	April
11	-34.750	11	14	UPPERBD NONBASIC	19	.	f2_to_1	March
12	-28.343	23	4	UPPERBD NONBASIC	25	1	production	March
13	-35.330	24	10	UPPERBD NONBASIC	25	1	backorder	April
14	-61.060	25	15	UPPERBD NONBASIC	25	.	f2_to_1	March
15	.	12	1	KEY_ARC BASIC	19	1	production	May
16	3.500	13	6	LOWERBD NONBASIC	19	1	storage	April
17	29.000	14	16	LOWERBD NONBASIC	19	.	f2_to_1	May
18	-15.520	26	4	UPPERBD NONBASIC	25	1	production	May
19	67.680	27	10	LOWERBD NONBASIC	25	1	storage	April
20	32.060	28	17	LOWERBD NONBASIC	25	.	f2_to_1	May
21	.	15	6	KEY_ARC BASIC	19	.	f1_to_2	April
22	-35.592	16	3	UPPERBD NONBASIC	19	2	production	April
23	13.400	17	14	LOWERBD NONBASIC	19	2	storage	March
24	43.980	18	16	LOWERBD NONBASIC	19	2	backorder	May
25	45.510	29	10	LOWERBD NONBASIC	25	.	f1_to_2	April
26	.	30	5	KEY_ARC BASIC	25	2	production	April
27	43.660	31	15	LOWERBD NONBASIC	25	2	storage	March
28	57.170	32	17	LOWERBD NONBASIC	25	2	backorder	May
29	55.750	33	7	LOWERBD NONBASIC	19	.	f1_to_2	March
30	.	34	3	KEY_ARC BASIC	19	2	production	March
31	25.480	35	9	LOWERBD NONBASIC	19	2	backorder	April
32	104.060	36	11	LOWERBD NONBASIC	25	.	f1_to_2	March
33	-23.170	37	5	UPPERBD NONBASIC	25	2	production	March
34	22.020	38	13	LOWERBD NONBASIC	25	2	backorder	April
35	.	39	8	KEY_ARC BASIC	19	.	f1_to_2	May
36	22.700	40	3	LOWERBD NONBASIC	19	2	production	May
37	6.500	41	9	LOWERBD NONBASIC	19	2	storage	April
38	6.940	42	12	LOWERBD NONBASIC	25	.	f1_to_2	May
39	.	43	5	KEY_ARC BASIC	25	2	production	May
40	46.110	44	13	LOWERBD NONBASIC	25	2	storage	April
41	.	45	7	KEY_ARC BASIC	19	1	sales	March
42	-23.500	46	6	UPPERBD NONBASIC	19	1	sales	April
43	.	47	8	NONKEY ARC BASIC	19	1	sales	May
44	-14.500	48	14	UPPERBD NONBASIC	19	2	sales	March
45	-2.500	49	9	UPPERBD NONBASIC	19	2	sales	April
46	9.000	50	16	LOWERBD NONBASIC	19	2	sales	May

Output 6.8.3 *continued*

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_RCOST_	_ANUMB_	_TNUMB_	_STATUS_	diagonal	factory	key_id	mth_made
47	79.150	51	11	LOWERBD NONBASIC	25	1	sales	March
48	40.420	52	10	LOWERBD NONBASIC	25	1	sales	April
49	.	53	12	KEY_ARC BASIC	25	1	sales	May
50	-9.980	54	15	UPPERBD NONBASIC	25	2	sales	March
51	.	55	13	KEY_ARC BASIC	25	2	sales	April
52	42.520	56	17	LOWERBD NONBASIC	25	2	sales	May
53	-37.090	57	7	UPPERBD NONBASIC	19	1	sales	March
54	-25.500	58	6	UPPERBD NONBASIC	19	1	sales	April
55	38.000	59	8	LOWERBD NONBASIC	19	1	sales	May
56	8.200	60	14	LOWERBD NONBASIC	19	2	sales	March
57	-26.500	61	9	UPPERBD NONBASIC	19	2	sales	April
58	.	62	16	KEY_ARC BASIC	19	2	sales	May
59	.	63	11	KEY_ARC BASIC	25	1	sales	March
60	.	64	10	KEY_ARC BASIC	25	1	sales	April
61	.	65	12	NONKEY ARC BASIC	25	1	sales	May
62	.	66	15	KEY_ARC BASIC	25	2	sales	March
63	-32.020	67	13	UPPERBD NONBASIC	25	2	sales	April
64	.	68	17	KEY_ARC BASIC	25	2	sales	May
65	.	-2	.	NONKEY BASIC	.	1	chips	
66	1.617	0	.	LOWERBD NONBASIC	.	1	chips	
67	2.797	-3	.	LOWERBD NONBASIC	.	2	chips	
68	.	-1	.	NONKEY BASIC	.	2	chips	

Output 6.8.4 DUALOUT=DUAL7

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_
1	f1_apr_1	.	-100000275.50	6	1	9	2	5	490.000
2	f1_apr_2	.	-100000405.92	10	21	4	2	-64	245.000
3	f1_mar_1	.	-100000326.65	7	18	1	20	-45	148.333
4	f1_mar_2	.	-100000480.13	11	21	15	1	-63	455.000
5	f1_may_1	.	-100000284.00	8	1	16	3	12	66.667
6	f1_may_2	.	-100000356.24	12	19	6	1	-53	347.500
7	f2_apr_1	.	-100000286.50	9	6	8	1	15	20.000
8	f2_apr_2	.	-100000383.41	13	5	19	3	30	542.500
9	f2_mar_1	.	-100000281.90	14	3	5	1	34	255.000
10	f2_mar_2	.	-100000399.07	15	21	10	1	-66	125.000
11	f2_may_1	.	-100000300.00	16	8	20	2	39	115.000
12	f2_may_2	.	-100000375.30	17	5	21	6	43	87.500
13	fact1_1	1000	-100000193.90	1	7	2	19	-9	288.333
14	fact1_2	1000	-100000227.42	4	10	13	1	-19	200.000
15	fact2_1	850	-100000193.90	3	2	14	2	-2	45.000
16	fact2_2	1500	-100000193.90	5	2	17	10	-4	150.000
17	shop1_1	-900	-99999999.00	18	22	7	21	0	0.000
18	shop1_2	-900	-99999841.22	19	13	12	2	55	52.500
19	shop2_1	-900	-100000001.00	20	16	22	1	62	150.000
20	shop2_2	-1450	-99999856.24	21	17	11	5	68	122.500
21		.	0.00	4	4	.	4	.	260.000
22		.	-1.00	2	2	.	.	-2	20.000

Obs	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
1	5	.	.		
2	19	.	.		
3	9	.	.		
4	23	.	.		
5	12	.	.		
6	26	.	.		
7	15	.	.		
8	29	.	.		
9	33	.	.		
10	36	.	.		
11	39	.	.		
12	42	.	.		
13	-1	.	.		
14	-5	.	.		
15	-5	.	.		
16	-5	.	.		
17	45	.	.		
18	51	.	.		
19	57	.	.		
20	63	.	.		
21	.	260	0	GE	CHIP LEFTOVER
22	.	2600	2600	EQ	FACT1 APL GIZMO

Output 6.8.4 *continued*

Nonarc Variables in the Side Constraints
Production Planning/Inventory/Distribution

Obs	_node_	_supdem_	_DUAL_	_NNUMB_	_PRED_	_TRAV_	_SCESS_	_ARCID_	_FLOW_
23		.	-1.62	0	18	.	.	47	1.667
24		.	1.80	3	21	.	.	65	2.500
25		.	0.00	1	1	.	.	-1	280.000
26		.	-0.48	5	10	.	.	21	0.000

Obs	_FBQ_	_VALUE_	_RHS_	_TYPE_	_row_
23	.	2615	2615	EQ	FACT1 MAR GIZMO
24	.	3750	3750	EQ	FACT2 APL GIZMO
25	.	3750	3750	EQ	FACT2 MAR GIZMO
26	.	50	50	LE	TOTAL BACKORDER

The optimal value of the nonarc variable "f2 unused chips" is 280. This means that although there are 3750 chips that can be used at factory 2 in March, only 3470 are used. As the optimal value of "f1 unused chips" is zero, all chips available for production in March at factory 1 are used. The nonarc variable "f2 chips from mar" also has zero optimal value. This means that the April production at factory 2 does not need any chips that could have been held in inventory since March. However, the nonarc variable "f1 chips from mar" has value of 20. Thus, 3490 chips should be ordered for factory 2 in March. Twenty of these chips should be held in inventory until April, then sent to factory 1.

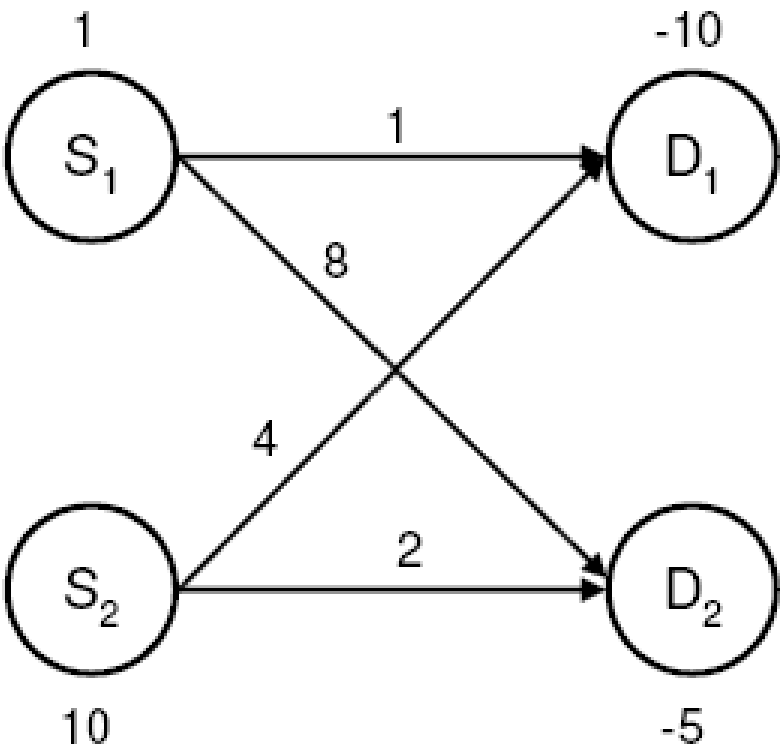
Example 6.9: Pure Networks: Using the EXCESS= Option

In this example we illustrate the use of the EXCESS= option for various scenarios in pure networks. Consider a simple network as shown in Output 6.9.1. The positive numbers on the nodes correspond to supply and the negative numbers correspond to demand. The numbers on the arcs indicate costs.

Transportation Problem, Total Supply < Total Demand

We first analyze a simple transportation problem where total demand exceeds total supply, as seen in Output 6.9.1. The EXCESS=SLACKS option is illustrated first.

Output 6.9.1 Transportation Problem



The following SAS code creates the input data sets.

```
data parcs;
   input _from_ $ _to_ $ _cost_;
datalines;
s1 d1 1
s1 d2 8
s2 d1 4
s2 d2 2
;

data SleD;
   input _node_ $ _sd_;
datalines;
s1   1
s2  10
d1 -10
d2  -5
;
```

You can solve the problem using the following call to PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   excess   = slacks
   arcdata  = parcs
   nodedata = SleD
   conout   = solex1;
run;
```

Since the EXCESS=SLACKS option is specified, the interior point method is used for optimization. Accordingly, the CONOUT= data set is specified. The optimal solution is displayed in Output 6.9.2.

Output 6.9.2 Supply < Demand

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	1	10	1	1
2	s2	d1	4	99999999	0	10	10	5	20
3	s1	d2	8	99999999	0	1	5	0	0
4	s2	d2	2	99999999	0	10	5	5	10

The solution with the THRUNET option specified is displayed in Output 6.9.3.

```
title1 'The NETFLOW Procedure';
proc netflow
   thrunet
   excess   = slacks
   arcdata  = parcs
   nodedata = SleD
   conout   = solex1t;
run;
```

Output 6.9.3 Supply < Demand, THRUNET Specified

The NETFLOW Procedure

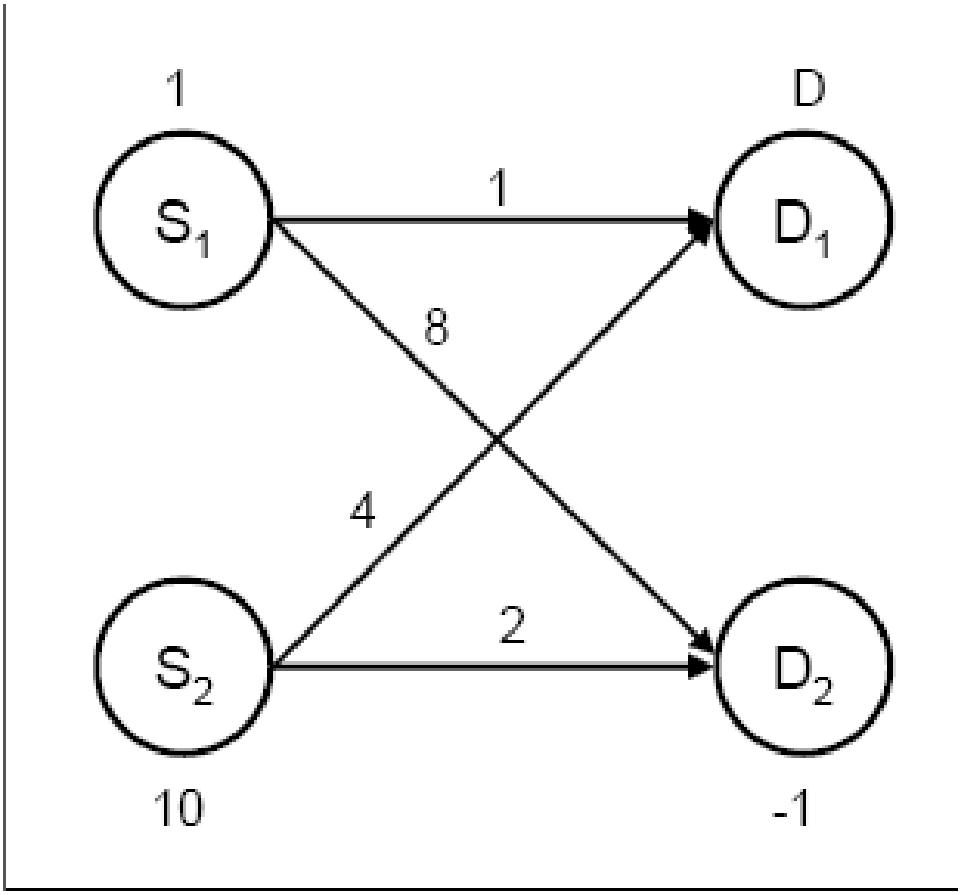
Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	1	10	5	5
2	s2	d1	4	99999999	0	10	10	5	20
3	s1	d2	8	99999999	0	1	5	0	0
4	s2	d2	2	99999999	0	10	5	5	10

NOTE: If you want to use the network simplex solver instead, you need to specify the EXCESS=ARCS option and, accordingly, the ARCOUT= data set.

Missing D Demand

As shown in Output 6.9.4, node D_1 has a missing D demand value.

Output 6.9.4 Missing D Demand



The following code creates the node data set:

```
data node_missingD1;
   input _node_ $ _sd_;
   missing D;
datalines;
s1   1
s2  10
d1   D
d2  -1
;
```

You can use the following call to PROC NETFLOW to solve the problem:

```
title1 'The NETFLOW Procedure';
proc netflow
   excess   = slacks
   arcdata  = parcs
   nodedata = node_missingD1
   conout   = solex1b;
run;
```

The optimal solution is displayed in Output 6.9.5. As you can see, the flow balance at nodes with nonmissing supdem values is maintained. In other words, if a node has a nonmissing supply (demand) value, then the sum of flows out of (into) that node is equal to its supdem value.

Output 6.9.5 THRUNET Not Specified

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	1	D	1	1
2	s2	d1	4	99999999	0	10	D	9	36
3	s1	d2	8	99999999	0	1	1	0	0
4	s2	d2	2	99999999	0	10	1	1	2

Missing D Demand, THRUNET Specified

Consider the previous example, but with the THRUNET option specified.

```
title1 'The NETFLOW Procedure';
proc netflow
   thrunet
   excess   = slacks
   arcdata  = parcs
   nodedata = node_missingD1
   conout   = solex1c;
run;
```

The optimal solution is displayed in Output 6.9.6. By specifying the THRUNET option, we have actually obtained the minimum-cost flow through the network, while maintaining flow balance at the nodes with nonmissing supply values.

Output 6.9.6 Missing D Demand, THRUNET Specified

The NETFLOW Procedure

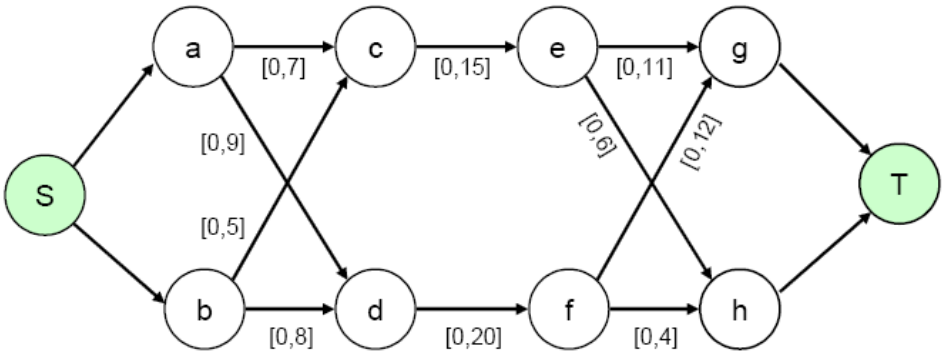
Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	1	D	1	1
2	s2	d1	4	99999999	0	10	D	0	0
3	s1	d2	8	99999999	0	1	1	0	0
4	s2	d2	2	99999999	0	10	1	10	20

NOTE: The case with missing S supply values is similar to the case with missing D demand values.

Example 6.10: Maximum Flow Problem

Consider the maximum flow problem depicted in Output 6.10.1. The maximum flow between nodes S and T is to be determined. The minimum arc flow and arc capacities are specified as lower and upper bounds in square brackets, respectively.

Output 6.10.1 Maximum Flow Problem Example



You can solve the problem using either EXCESS=ARCS or EXCESS=SLACKS. Consider using the EXCESS=ARCS option first. You can use the following SAS code to create the input data set:

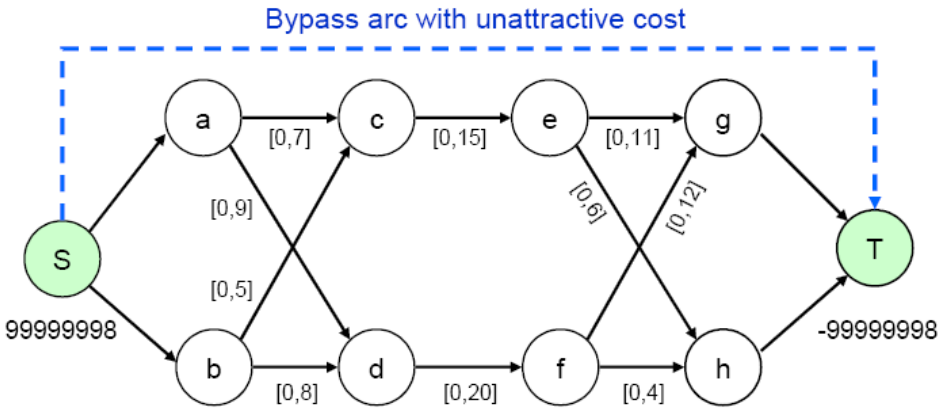
```
data arcs;
   input _from_ $ _to_ $ _cost_ _capac_;
datalines;
S a  .  .
S b  .  .
a c  1  7
b c  2  9
a d  3  5
b d  4  8
c e  5 15
d f  6 20
e g  7 11
f g  8  6
e h  9 12
f h 10  4
g T  .  .
h T  .  .
;
```

You can use the following call to PROC NETFLOW to solve the problem:

```
title1 'The NETFLOW Procedure';
proc netflow
   intpoint
   maxflow
   excess = arcs
   arcdata = arcs
   source  = S    sink = T
   conout  = gout3;
run;
```

With the EXCESS=ARCS option specified, the problem gets transformed internally to the one depicted in Output 6.10.2. Note that there is an additional arc from the source node to the sink node.

Output 6.10.2 Maximum Flow Problem, EXCESS=ARCS Option Specified



The output SAS data set is displayed in Output 6.10.3.

Output 6.10.3 Maximum Flow Problem, EXCESS=ARCS Option Specified

The NETFLOW Procedure

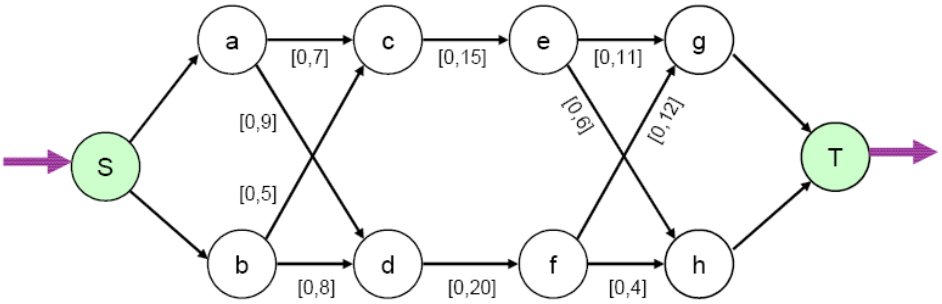
Obs	_from_	_to_	_cost_	_capac_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	g	T	0	99999999	0	.	99999998	16.9996	0.0000
2	h	T	0	99999999	0	.	99999998	8.0004	0.0000
3	S	a	0	99999999	0	99999998	.	11.9951	0.0000
4	S	b	0	99999999	0	99999998	.	13.0049	0.0000
5	a	c	1	7	0	.	.	6.9952	6.9952
6	b	c	2	9	0	.	.	8.0048	16.0097
7	a	d	3	5	0	.	.	4.9999	14.9998
8	b	d	4	8	0	.	.	5.0001	20.0002
9	c	e	5	15	0	.	.	15.0000	75.0000
10	d	f	6	20	0	.	.	10.0000	60.0000
11	e	g	7	11	0	.	.	10.9996	76.9975
12	f	g	8	6	0	.	.	6.0000	48.0000
13	e	h	9	12	0	.	.	4.0004	36.0032
14	f	h	10	4	0	.	.	4.0000	40.0000

You can solve the same maximum flow problem, but this time with EXCESS=SLACKS specified. The SAS code is as follows:

```
title1 'The NETFLOW Procedure';
proc netflow
   intpoint
   excess  = slacks
   arcdata = arcs
   source  = S    sink = T
   maxflow
   conout  = gout3b;
run;
```

With the EXCESS=SLACKS option specified, the problem gets transformed internally to the one depicted in Figure 6.10.4. Note that the source node and sink node each have a single-ended "excess" arc attached to them.

Output 6.10.4 Maximum Flow Problem, EXCESS=SLACKS Option Specified



The solution, as displayed in Output 6.10.5, is the same as before. Note that the _SUPPLY_ value of the source node Y has changed from 99999998 to missing S, and the _DEMAND_value of the sink node Z has changed from −99999998 to missing D.

Output 6.10.5 Maximal Flow Problem

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_capac_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	g	T	0	99999999	0	.	D	16.9993	0.0000
2	h	T	0	99999999	0	.	D	8.0007	0.0000
3	S	a	0	99999999	0	S	.	11.9867	0.0000
4	S	b	0	99999999	0	S	.	13.0133	0.0000
5	a	c	1	7	0	.	.	6.9868	6.9868
6	b	c	2	9	0	.	.	8.0132	16.0264
7	a	d	3	5	0	.	.	4.9999	14.9998
8	b	d	4	8	0	.	.	5.0001	20.0002
9	c	e	5	15	0	.	.	15.0000	75.0000
10	d	f	6	20	0	.	.	10.0000	60.0000
11	e	g	7	11	0	.	.	10.9993	76.9953
12	f	g	8	6	0	.	.	6.0000	48.0000
13	e	h	9	12	0	.	.	4.0007	36.0061
14	f	h	10	4	0	.	.	4.0000	40.0000

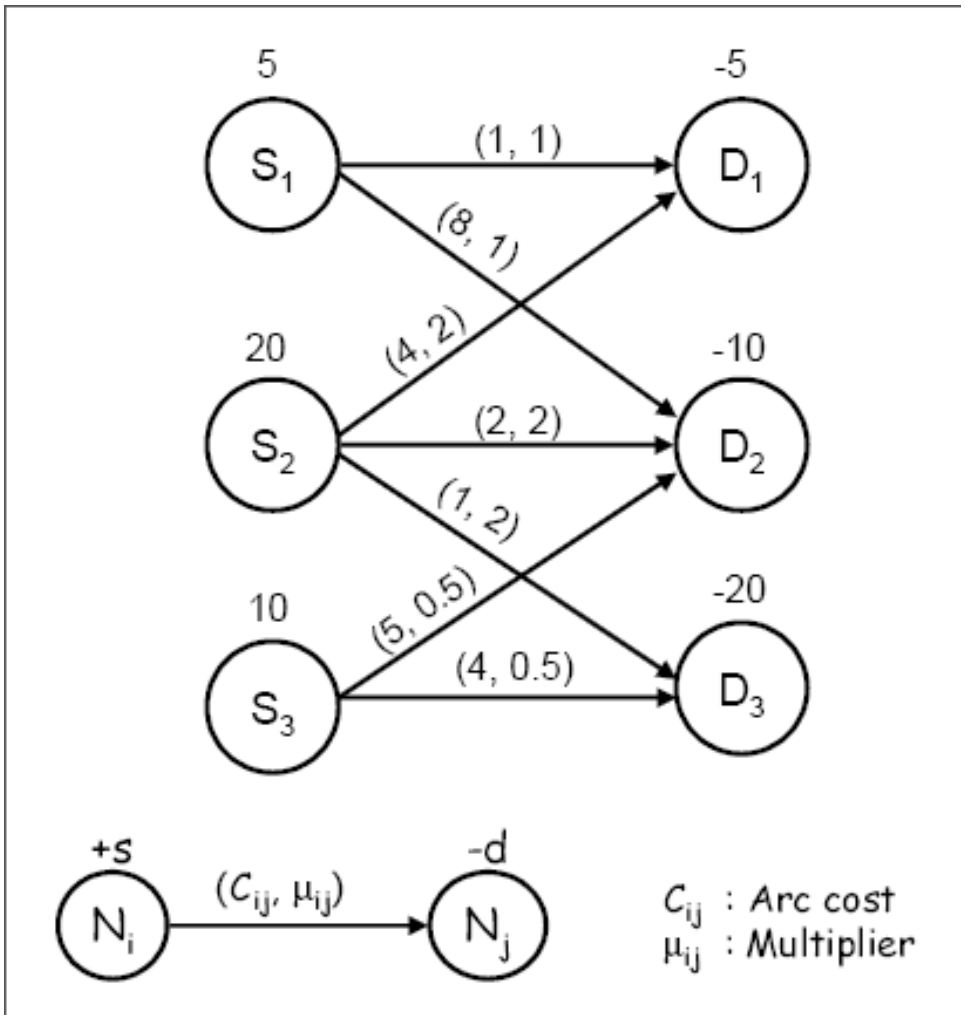
Example 6.11: Generalized Networks: Using the EXCESS= Option

For generalized networks you can specify either EXCESS=SUPPLY or EXCESS=DEMAND to indicate which nodal flow conservation constraints have slack variables associated with them. The default option is EXCESS=NONE.

Using the EXCESS=SUPPLY Option

Consider the simple network shown in Output 6.11.1. As you can see, the sum of positive supdem values (35) is equal to the absolute sum of the negative ones. However, the arcs connecting the supply and demand nodes have varying arc multipliers. Let us now solve the problem using the EXCESS=SUPPLY option.

Output 6.11.1 Generalized Network: Supply = Demand



You can use the following SAS code to create the input data sets:

```
data garcs;
   input _from_ $ _to_ $ _cost_ _mult_;
datalines;
s1 d1 1 .
s1 d2 8 .
s2 d1 4 2
s2 d2 2 2
s2 d3 1 2
s3 d2 5 0.5
s3 d3 4 0.5
;

data gnodes;
   input _node_ $ _sd_ ;
datalines;
s1  5
s2  20
s3  10
d1  -5
d2  -10
d3  -20
;
```

To solve the problem, use the following call to PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   arcdata  = garcs
   nodedata = gnodes
   excess   = supply
   conout   = gnetout;
run;
```

The optimal solution is displayed in Output 6.11.2.

Output 6.11.2 Optimal Solution Obtained Using the EXCESS=SUPPLY Option

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	1.0	5	5	5	5
2	s2	d1	4	99999999	0	2.0	20	5	0	0
3	s1	d2	8	99999999	0	1.0	5	10	0	0
4	s2	d2	2	99999999	0	2.0	20	10	5	10
5	s3	d2	5	99999999	0	0.5	10	10	0	0
6	s2	d3	1	99999999	0	2.0	20	20	10	10
7	s3	d3	4	99999999	0	0.5	10	20	0	0

NOTE: If you do not specify the EXCESS= option, or if you specify the EXCESS=DEMAND option, the procedure will declare the problem infeasible. Therefore, in case of real-life problems, you would need to have a little more detail about how the arc multipliers end up affecting the network — whether they tend to create excess demand or excess supply.

Using the EXCESS=DEMAND Option

Consider the previous example but with a slight modification: the arcs out of node S_1 have multipliers of 0.5, and the arcs out of node S_2 have multipliers of 1. You can use the following SAS code to create the input arc data set:

```
data garcs1;
   input _from_ $ _to_ $ _cost_ _mult_;
datalines;
s1 d1 1 0.5
s1 d2 8 0.5
s2 d1 4 .
s2 d2 2 .
s2 d3 1 .
s3 d2 5 0.5
s3 d3 4 0.5
;
```

Note that the node data set remains unchanged. You can use the following call to PROC NETFLOW to solve the problem:

```
title1 'The NETFLOW Procedure';
proc netflow
   arcdata  = garcs1
   nodedata = gnodes
   excess = demand
   conout   = gnetout1;
run;
```

The optimal solution is displayed in Output 6.11.3.

Output 6.11.3 Optimal Solution Obtained Using the EXCESS=DEMAND Option

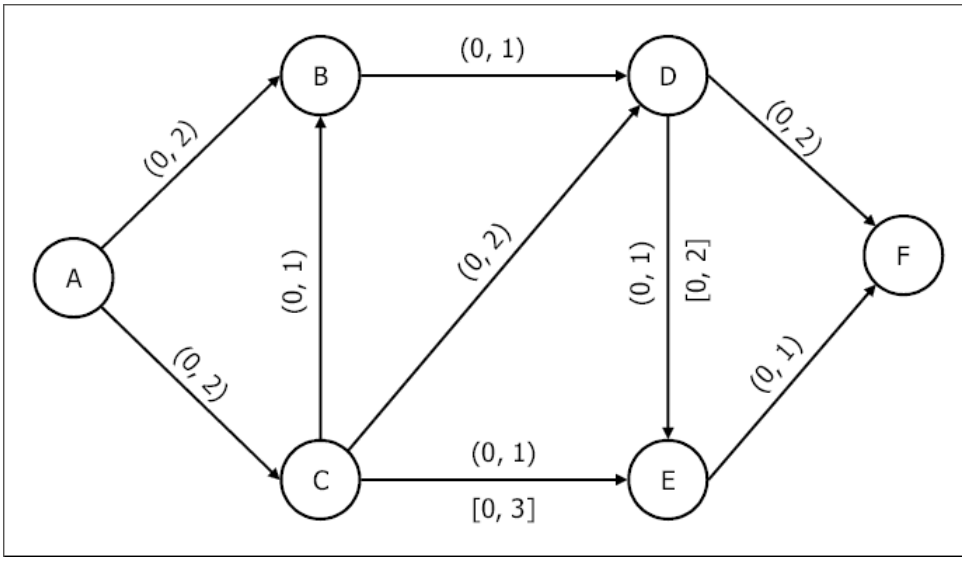
The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	s1	d1	1	99999999	0	0.5	5	5	5.0000	5.0000
2	s2	d1	4	99999999	0	1.0	20	5	0.0000	0.0000
3	s1	d2	8	99999999	0	0.5	5	10	0.0000	0.0000
4	s2	d2	2	99999999	0	1.0	20	10	5.0000	10.0000
5	s3	d2	5	99999999	0	0.5	10	10	0.0000	0.0000
6	s2	d3	1	99999999	0	1.0	20	20	15.0000	15.0000
7	s3	d3	4	99999999	0	0.5	10	20	10.0000	40.0000

Example 6.12: Generalized Networks: Maximum Flow Problem

Consider the generalized network displayed in Output 6.12.1. Lower and upper bounds of the flow are displayed in parentheses above the arc, and cost and multiplier, where applicable, are indicated in square brackets below the arc.

Output 6.12.1 Generalized Maximum Flow Problem



You can enter the data for the problem using the following SAS code:

```
data garcsM;
   input _from_ $ _to_ $ _upper_ _mult_;
datalines;
A B 2 .
A C 2 .
C B 1 .
B D 1 .
C D 2 .
C E 1 3
D E 1 2
E F 5 .
D F 2 .
;
```

Use the following call to PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   arcdata = garcsM
   maxflow
   source  = A  sink = F
   conout  = gmfpout;
run;
```

The optimal solution is displayed in Output 6.12.2.

Output 6.12.2 Generalized Maximum Flow Problem: Optimal Solution

The NETFLOW Procedure

Obs	_from_	_to_	_COST_	_upper_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	A	B	0	2	0	1	S	.	1.00000	0
2	C	B	0	1	0	1	.	.	0.00000	0
3	A	C	0	2	0	1	S	.	2.00000	0
4	B	D	0	1	0	1	.	.	1.00000	0
5	C	D	0	2	0	1	.	.	1.00000	0
6	C	E	0	1	0	3	.	.	1.00000	0
7	D	E	0	1	0	2	.	.	1.00000	0
8	E	F	0	5	0	1	.	D	5.00000	0
9	D	F	0	2	0	1	.	D	1.00000	0

Example 6.13: Machine Loading Problem

Machine loading problems arise in a variety of applications. Consider a simple instance as described in Ahuja, Magnanti, and Orlin (1993). Assume you need to schedule the production of three products, $P1 - P3$, on four machines, $M1 - M4$. Suppose that machine 1 and machine 2 are each available for 40 hours and machine 3 and machine 4 are each available for 50 hours. Also, any of the machines can produce any product. The per-unit processing time and production cost for each product on each machine are indicated in Table 6.14.

Table 6.14 Processing Times and Production Costs

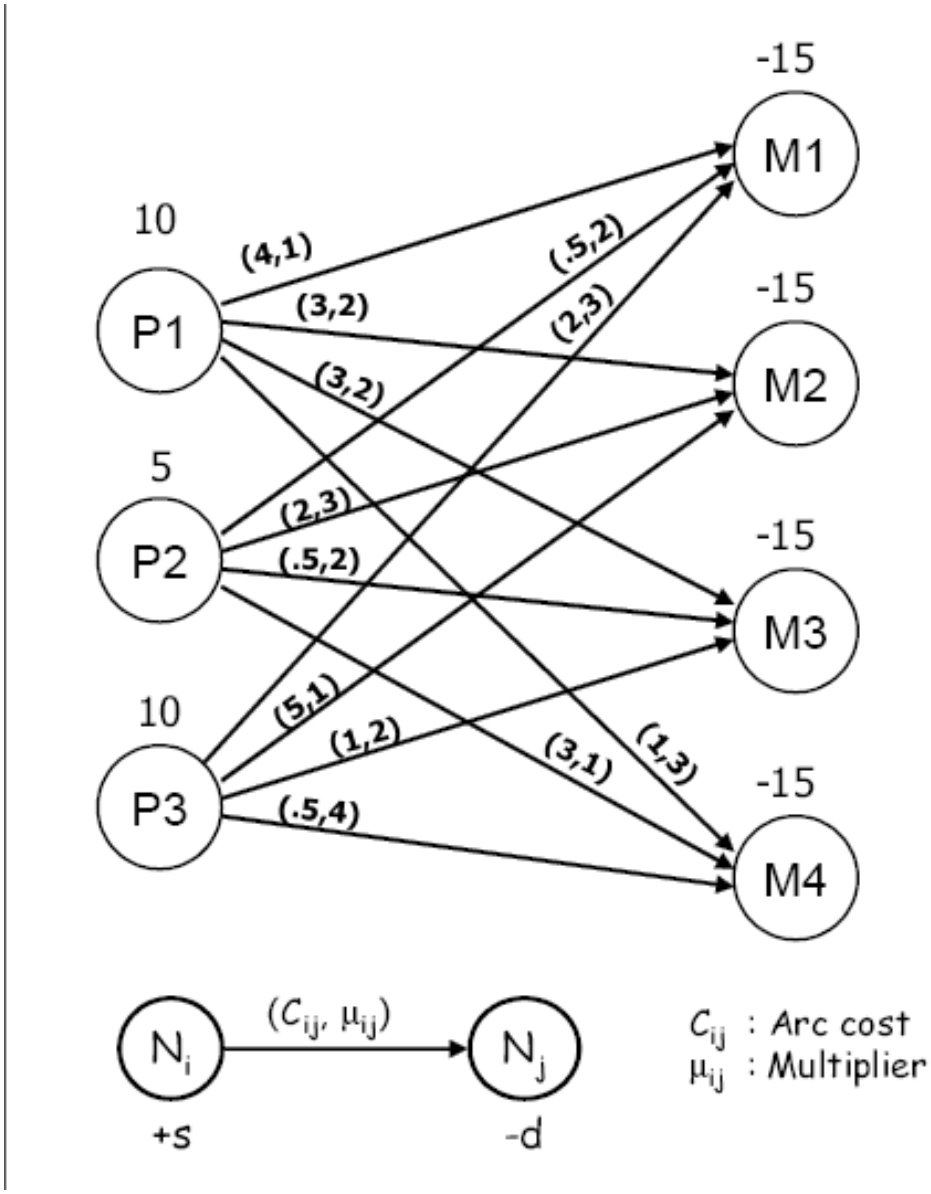
	M1	M2	M3	M4
P1	1	2	2	3
P2	2	3	2	1
P3	3	1	2	4

	M1	M2	M3	M4
P1	4	3	3	1
P2	0.5	2	0.5	3
P3	2	5	1	5

The problem is to satisfy the demands for the three products at minimum cost.

You can model this problem as a generalized network as shown in Output 6.13.1. The network has three product nodes with demands indicated by positive supdem values and four machine nodes with availabilities (in hours) indicated by negative supdem values. The multiplier on an arc between a machine and a product indicates the hours of machine capacity needed to produce one unit of the product.

Output 6.13.1 Machine Loading Problem



You can create the input data sets with the following SAS code:

```
data mlarcs;
   input _from_ $ _to_ $ _cost_ _mult_;
datalines;
P1  M1   4   .
P1  M2   3   2
P1  M3   3   2
P1  M4   1   3
P2  M1  .5   2
P2  M2   2   3
P2  M3  .5   2
P2  M4   3   1
P3  M1   2   3
```

```
P3  M2   5   .
P3  M3   1   2
P3  M4  .5   4
;

data mlnodes;
   input _node_ $ _sd_;
datalines;
P1   10
P2   5
P3   10
M1  -40
M2  -40
M3  -50
M4  -50
;
```

You can solve the problem using the following call to PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   excess   = demand
   arcdata  = mlarcs
   nodedata = mlnodes
   conout   = mlsol;
run;
```

The optimal solution, as displayed in Output 6.13.2, can be interpreted as follows:

- Product 1: 10 units on machine 4
- Product 2: 3 units on machine 1, and 2 units on machine 3
- Product 3: 5 units on machine 3, and 5 units on machine 4

Output 6.13.2 Optimum Solution to the Machine Loading Problem

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	P1	M1	4.0	99999999	0	1	10	40	0.00000	0.00000
2	P2	M1	0.5	99999999	0	2	5	40	3.67856	1.83928
3	P3	M1	2.0	99999999	0	3	10	40	0.00000	0.00000
4	P1	M2	3.0	99999999	0	2	10	40	0.00000	0.00000
5	P2	M2	2.0	99999999	0	3	5	40	0.00000	0.00000
6	P3	M2	5.0	99999999	0	1	10	40	0.00000	0.00000
7	P1	M3	3.0	99999999	0	2	10	50	0.00000	0.00000
8	P2	M3	0.5	99999999	0	2	5	50	1.32144	0.66072
9	P3	M3	1.0	99999999	0	2	10	50	5.00000	5.00000
10	P1	M4	1.0	99999999	0	3	10	50	10.0000	10.0000
11	P2	M4	3.0	99999999	0	1	5	50	0.00000	0.00000
12	P3	M4	0.5	99999999	0	4	10	50	5.00000	2.50000

Example 6.14: Generalized Networks: Distribution Problem

Consider a distribution problem (from Jensen and Bard 2003) with three supply plants ($S_1 - S_3$) and five demand points ($D_1 - D_5$). Further information about the problem is as follows:

S_1 To be closed. Entire inventory must be shipped or sold to scrap. The scrap value is \$5 per unit.

S_2 Maximum production of 300 units with manufacturing cost of \$10 per unit.

S_3 The production in regular time amounts to 200 units and must be shipped. An additional 100 units can be produced using overtime at \$14 per unit.

D_1 Fixed demand of 200 units must be met.

D_2 Contracted demand of 300 units. An additional 100 units can be sold at \$20 per unit.

D_3 Minimum demand of 200 units. An additional 100 units can be sold at \$20 per unit. Additional units can be procured from D_4 at \$4 per unit. There is a 5% "shipping loss" on the arc connecting these two nodes.

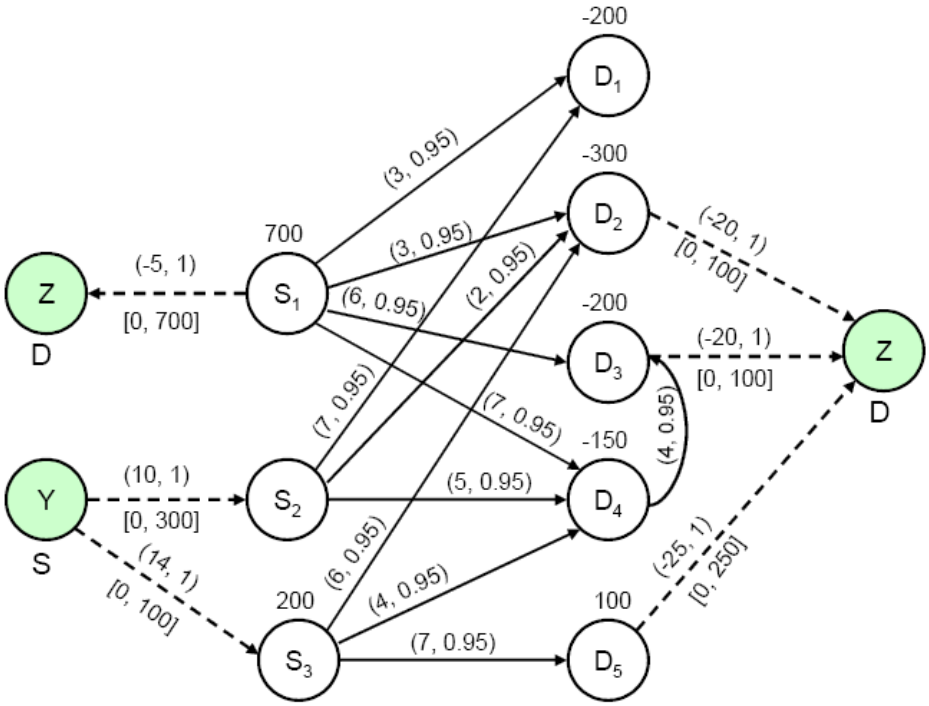
D_4 Fixed demand of 150 units must be met.

D_5 100 units left over from previous shipments. No firm demand, but up to 250 units can be sold at \$25 per unit.

Additionally, there is a 5% "shipping loss" on each of the arcs between supply and demand nodes.

You can model this scenario as a generalized network. Since there are both fixed and varying supply and demand supdem values, you can transform this to a case where you need to address missing supply and demand simultaneously. As seen from Output 6.14.1, we have added two artificial nodes, Y and Z, with missing S supply value and missing D demand value, respectively. The extra production capability is depicted by arcs from node Y to the corresponding supply nodes, and the extra revenue generation capability of the demand points (and scrap revenue for S_1) is depicted by arcs to node Z.

Output 6.14.1 Distribution Problem



The following SAS data set has the complete information about the arc costs, multipliers, and node supdem values:

```
data dnodes;
   input _node_ $ _sd_ ;
   missing S D;
datalines;
S1   700
S2     0
S3   200
D1  -200
D2  -300
D3  -200
D4  -150
D5   100
Y      S
Z      D
;

data darcs;
   input _from_ $ _to_ $ _cost_ _capac_ _mult_;
datalines;
S1  D1    3   200  0.95
S1  D2    3   200  0.95
S1  D3    6   200  0.95
S1  D4    7   200  0.95
S2  D1    7   200  0.95
S2  D2    2   200  0.95
```

```
S2  D4    5   200  0.95
S3  D2    6   200  0.95
S3  D4    4   200  0.95
S3  D5    7   200  0.95
D4  D3    4   200  0.95
Y   S2   10   300  .
Y   S3   14   100  .
S1  Z    -5   700  .
D2  Z   -20   100  .
D3  Z   -20   100  .
D5  Z   -25   250  .
;
```

You can solve this problem by using the following call to PROC NETFLOW:

```
title1 'The NETFLOW Procedure';
proc netflow
   nodedata = dnodes
   arcdata  = darcs
   conout   = dsol;
run;
```

The optimal solution is displayed in Output 6.14.2.

Output 6.14.2 Distribution Problem: Optimal Solution

The NETFLOW Procedure

Obs	_from_	_to_	_cost_	_capac_	_LO_	_mult_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_
1	S1	D1	3	200	0	0.95	700	200	200.000	600.00
2	S2	D1	7	200	0	0.95	.	200	10.526	73.68
3	S1	D2	3	200	0	0.95	700	300	200.000	600.00
4	S2	D2	2	200	0	0.95	.	300	200.000	400.00
5	S3	D2	6	200	0	0.95	200	300	21.053	126.32
6	S1	D3	6	200	0	0.95	700	200	200.000	1200.00
7	D4	D3	4	200	0	0.95	.	200	10.526	42.11
8	S1	D4	7	200	0	0.95	700	150	100.000	700.00
9	S2	D4	5	200	0	0.95	.	150	47.922	239.61
10	S3	D4	4	200	0	0.95	200	150	21.053	84.21
11	S3	D5	7	200	0	0.95	200	.	157.895	1105.26
12	Y	S2	10	300	0	1.00	S	.	258.449	2584.49
13	Y	S3	14	100	0	1.00	S	.	0.000	0.00
14	S1	Z	-5	700	0	1.00	700	D	0.000	0.00
15	D2	Z	-20	100	0	1.00	.	D	100.000	-2000.00
16	D3	Z	-20	100	0	1.00	.	D	0.000	0.00
17	D5	Z	-25	250	0	1.00	100	D	250.000	-6250.00
										-494.32

Example 6.15: Converting to an MPS-Format SAS Data Set

This example demonstrates the use of the MPSOUT= option to convert a problem data set in PROC NETFLOW input format into an MPS-format SAS data set for use with the OPTLP procedure.

Suppose you want to solve a linear program with the following formulation:

$$
\begin{array}{rrcrcrcr}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & \\
\text{subject to} & & - & 2x_2 & - & 3x_3 & \geq & -5 \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 \\
& x_1 & + & 2x_2 & + & 3x_3 & \geq & 7 \\
& & 0 & \leq & x_1 & \leq & 10 & \\
& & 0 & \leq & x_2 & \leq & 15 & \\
& & 0 & \leq & x_3 & \leq & 20 &
\end{array}
$$

You can save the LP in dense format by using the following DATA step:

```
data exdata;
   input x1 x2 x3 _type_ $ _rhs_;
datalines;
2 -3 -4 min      .
. -2 -3 >=      -5
1  1  2 <=       6
1  2  3 >=       7
10 15 20 upperbd  .
;
```

If you decide to solve the problem by using the OPTLP procedure, you need to convert the data set exdata from dense format to MPS format. You can accomplish this by using the following statements:

```
/* convert to MPS format */

proc netflow condata=exdata mpsout=mpsdata bytes=100000;
run;
```

The MPS-format SAS data set mpsdata is shown in Output 6.15.1.

Output 6.15.1 Data Set mpsdata

The NETFLOW Procedure

Obs	field1	field2	field3	field4	field5	field6
1	NAME		modname	.		.
2	ROWS			.		.
3	MIN	objfn		.		.
4	G	_OBS2_		.		.
5	L	_OBS3_		.		.
6	G	_OBS4_		.		.
7	COLUMNS			.		.
8		x1	objfn	2	_OBS3_	1
9		x1	_OBS4_	1		.
10		x2	objfn	-3	_OBS2_	-2
11		x2	_OBS3_	1	_OBS4_	2
12		x3	objfn	-4	_OBS2_	-3
13		x3	_OBS3_	2	_OBS4_	3
14	RHS			.		.
15			_OBS2_	-5	_OBS3_	6
16			_OBS4_	7		.
17	BOUNDS			.		.
18	UP	bdsvect	x1	10		.
19	UP	bdsvect	x2	15		.
20	UP	bdsvect	x3	20		.
21	ENDATA			.		.

The constraint names _OBS2_, _OBS3_, and _OBS4_ are generated by the NETFLOW procedure. If you want to provide your own constraint names, use the ROW list variable in the CONOUT= data set. If you specify the problem data in sparse format instead of dense format, the MPSOUT= option produces the same MPS-format SAS data set shown in the preceding output.

Now that the problem data are in MPS format, you can solve the problem by using the OPTLP procedure. For more information, see Chapter 12, "The OPTLP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

Example 6.16: Migration to OPTMODEL: Generalized Networks

The following example shows how to solve Example 6.11 using PROC OPTMODEL. The input data sets are the same as in that example.

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GNETOUT:

```
proc optmodel;
   set <str> NODES;
   num _sd_ {NODES} init 0;
   read data gnodes into NODES=[_node_] _sd_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num _mult_ {ARCS} init 1;
   read data garcs nomiss into ARCS=[_from_ _to_] _cost_ _mult_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];

   num infinity = constant('BIG');
   /* change equality constraint to le constraint for supply nodes */
   for {i in NODES: _sd_[i] > 0} balance[i].lb = -infinity;

   solve;

   num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
   num _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gnetout from [_from_ _to_]
      _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a generalized network flow problem, the usual balance constraint is altered to include the arc multiplier "_mult_[j,i]" in the second sum. The balance constraint is initially declared as an equality, but to mimic the PROC NETFLOW EXCESS=SUPPLY option, the sense of this constraint is changed to "$\leq$" by relaxing the constraint's lower bound for supply nodes. The output data set contains the same optimal solution as Output 6.11.2. The log is displayed in Output 6.16.1.

Output 6.16.1 OPTMODEL Log

```
NOTE: There were 6 observations read from the data set WORK.GNODES.
NOTE: There were 7 observations read from the data set WORK.GARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7 variables (0 free, 0 fixed).
NOTE: The problem has 6 linear constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 14 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 2 variables and 2 constraints.
NOTE: The LP presolver removed 4 constraint coefficients.
NOTE: The presolved problem has 5 variables, 4 constraints, and 10 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                            Objective
      Phase Iteration         Value         Time
       D 1          1    0.000000E+00          0
       D 2          2    1.500000E+01          0
       D 2          3    2.500000E+01          0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GNETOUT has 7 observations and 10 variables.
```

Now consider the previous example but with a slight modification to the arc multipliers, as in Example 6.11.

```
data garcs1;
   input _from_ $ _to_ $ _cost_ _mult_;
   datalines;
s1 d1 1 0.5
s1 d2 8 0.5
s2 d1 4 .
s2 d2 2 .
s2 d3 1 .
s3 d2 5 0.5
s3 d3 4 0.5
;
```

The following PROC OPTMODEL statements are identical to the preceding statements, except for the balance constraint. It is still initially declared as an equality, but to mimic the PROC NETFLOW EXCESS=DEMAND option, the sense of this constraint is changed to "≥" by relaxing the constraint's upper bound for demand nodes.

```
proc optmodel;
   set <str> NODES;
   num _sd_ {NODES} init 0;
   read data gnodes into NODES=[_node_] _sd_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num _mult_ {ARCS} init 1;
   read data garcs1 nomiss into ARCS=[_from_ _to_] _cost_ _mult_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];

   num infinity = constant('BIG');
   /* change equality constraint to ge constraint */
   for {i in NODES: _sd_[i] < 0} balance[i].ub = infinity;

   solve;

   num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
   num _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gnetout1 from [_from_ _to_]
      _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

The output data set contains the same optimal solution as Output 6.11.3. The log is displayed in Output 6.16.2.

Output 6.16.2 OPTMODEL Log

```
NOTE: There were 6 observations read from the data set WORK.GNODES.
NOTE: There were 7 observations read from the data set WORK.GARCS1.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7 variables (0 free, 0 fixed).
NOTE: The problem has 6 linear constraints (0 LE, 3 EQ, 3 GE, 0 range).
NOTE: The problem has 14 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 2 variables and 2 constraints.
NOTE: The LP presolver removed 4 constraint coefficients.
NOTE: The presolved problem has 5 variables, 4 constraints, and 10 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration       Value         Time
       D 1          1    0.000000E+00         0
       D 2          2    4.997000E+01         0
       D 2          4    7.000000E+01         0
NOTE: Optimal.
NOTE: Objective = 70.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GNETOUT1 has 7 observations and 10 variables.
```

Example 6.17: Migration to OPTMODEL: Maximum Flow

The following example shows how to solve Example 6.10 using PROC OPTMODEL. The input data set is the same as in that example.

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GOUT3:

```
proc optmodel;
   str source = 'S';
   str sink = 'T';

   set <str> NODES;
   num _supdem_ {i in NODES} = (if i in {source, sink} then . else 0);

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS} init 0;
   read data arcs nomiss into ARCS=[_from_ _to_] _cost_ _capac_;
   NODES = (union {<i,j> in ARCS} {i,j});
```

```
   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   max obj = sum {<i,j> in ARCS: j = sink} Flow[i,j];
   con balance {i in NODES diff {source, sink}}:
      sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gout3 from [_from_ _to_]
      _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a maximum flow problem, you solve a network flow problem that has a zero supply or demand at all nodes other than the source and sink nodes, as specified in the declaration of the _SUPDEM_ numeric parameter and the balance constraint. The objective declaration uses the logical condition J = SINK to maximize the flow into the sink node. The output data set contains the same optimal solution as Output 6.10.3. The log is displayed in Output 6.17.1.

Output 6.17.1 OPTMODEL Log

```
NOTE: There were 14 observations read from the data set WORK.ARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 14 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 24 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is
      recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 6 constraints.
NOTE: The LP presolver removed 20 constraint coefficients.
NOTE: The presolved problem has 4 variables, 2 constraints, and 4 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                            Objective
      Phase Iteration         Value         Time
       D 1          1    0.000000E+00          0
       D 2          2    2.500000E+01          0
       P 2          5    2.500000E+01          0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GOUT3 has 14 observations and 9 variables.
```

Example 6.18: Migration to OPTMODEL: Production, Inventory, Distribution

The following example shows how to solve Example 6.4 using PROC OPTMODEL. The input data sets are the same as in that example.

The following PROC OPTMODEL code read the data sets, build the linear programming model, solve the model, and output the optimal solution to SAS data sets called ARC1 and NODE2:

```
proc optmodel;
   set <str> NODES;
   num _supdem_ {NODES} init 0;
   read data node0 into NODES=[_node_] _supdem_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num diagonal {ARCS};
   num factory {ARCS};
   str key_id {ARCS};
   str mth_made {ARCS};
   str _name_ {ARCS};
   read data arc0 nomiss into ARCS=[_tail_ _head_] _lo_ _capac_ _cost_
      diagonal factory key_id mth_made _name_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}:
      sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   num infinity = constant('BIG');
   num excess = sum {i in NODES} _supdem_[i];
   if (excess > 0) then do;
      /* change equality constraint to le constraint for supply nodes */
      for {i in NODES: _supdem_[i] > 0} balance[i].lb = -infinity;
   end;
   else if (excess < 0) then do;
      /* change equality constraint to ge constraint for demand nodes */
      for {i in NODES: _supdem_[i] < 0} balance[i].ub = infinity;
   end;

   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
```

```
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data arc1 from [_tail_ _head_]
      _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_=Flow _fcost_
      _rcost_ =
         (if Flow[_tail_,_head_].rc ne 0 then Flow[_tail_,_head_].rc else .)
      _status_ = Flow.status diagonal factory key_id mth_made;
   create data node2 from [_node_]
      _supdem_ = (if _supdem_[_node_] ne 0 then _supdem_[_node_] else .)
      _dual_ = balance.dual;
quit;
```

The statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The _SUPDEM_, _LO_, and _CAPAC_ parameters are given initial values, and the NOMISS option in the READ DATA statement tells OPTMODEL to read only the nonmissing values from the input data set. The balance constraint is initially declared as an equality, but depending on the total supply or demand, the sense of this constraint is changed to "≤" or "≥" by relaxing the constraint's lower or upper bound, respectively. The ARC1 output data set contains most of the same information as in Example 6.4, including reduced cost, basis status, and dual values. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to Example 6.4.

```
options ls=80 ps=54;
proc print data=arc1 heading=h width=min;
   var _tail_ _head_ _cost_ _capac_ _lo_ _name_
   _supply_ _demand_ _flow_ _fcost_;
   sum _fcost_;
run;
proc print data=arc1 heading=h width=min;
   var _rcost_ _status_ diagonal factory key_id mth_made;
run;
proc print data=node2;
run;
```

The output data sets contain the same optimal solution as Output 6.4.1, Output 6.4.2, and Output 6.4.3. The log is displayed in Output 6.18.1.

Output 6.18.1 OPTMODEL Log

```
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 64 observations read from the data set WORK.ARC0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 64 variables (0 free, 0 fixed).
NOTE: The problem has 20 linear constraints (4 LE, 16 EQ, 0 GE, 0 range).
NOTE: The problem has 128 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 64 variables, 20 constraints, and 128
      constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration       Value         Time
       D 1          1    0.000000E+00         0
       D 2          2   -4.020320E+06         0
       D 2         32   -1.281110E+06         0
NOTE: Optimal.
NOTE: Objective = -1281110.35.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.ARC1 has 64 observations and 16 variables.
NOTE: The data set WORK.NODE2 has 20 observations and 3 variables.
```

Example 6.19: Migration to OPTMODEL: Shortest Path

The following example shows how to solve Example 6.1 using PROC OPTMODEL. The input data set is the same as in that example.

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SPATH:

```
proc optmodel;
   str sourcenode = 'Honolulu';
   str sinknode = 'Heathrow London';

   set <str> NODES;
   num _supdem_ {i in NODES} = (if i = sourcenode then 1
      else if i = sinknode then -1 else 0);

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   read data aircost1 into ARCS=[ffrom tto] _cost_;
   NODES = (union {<i,j> in ARCS} {i,j});
```

```
   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];
   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data spath from [ffrom tto]
      _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_
      _rcost_ = (if Flow[ffrom,tto].rc ne 0 then Flow[ffrom,tto].rc else .)
      _status_ = Flow.status;
quit;
```

The statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets. The ARCS data set is populated from the ffrom and tto data set variables in the READ DATA statement. To solve a shortest path problem, you solve a minimum-cost network-flow problem that has a supply of one unit at the source node, a demand of one unit at the sink node, and zero supply or demand at all other nodes, as specified in the declaration of the _SUPDEM_ numeric parameter. The SPATH output data set contains most of the same information as in Example 6.1, including reduced cost and basis status. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to Example 6.1.

```
proc print data=spath;
   sum _fcost_;
run;
```

The output data set contains the same optimal solution as Output 6.1.1. The log is displayed in Output 6.19.1.

Output 6.19.1 OPTMODEL Log

```
NOTE: There were 13 observations read from the data set WORK.AIRCOST1.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 13 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 26 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is
      recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 177.
NOTE: The data set WORK.SPATH has 13 observations and 11 variables.
```

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993), *Network Flows: Theory, Algorithms, and Applications*, Englewood Cliffs, NJ: Prentice-Hall.

Bland, R. G. (1977), "New Finite Pivoting Rules for the Simplex Method," *Mathematics of Operations Research*, 2, 103–107.

George, J. A., Liu, J. W., and Ng, E. (2001), "Computer Solution of Positive Definite Systems," Unpublished manuscript obtainable from authors.

Jensen, P. A. and Bard, J. F. (2003), *Operations Research Models and Methods*, Hoboken, NJ: John Wiley & Sons.

Kearney, T. D. (1990), "A Tutorial on the NETFLOW Procedure in SAS/OR," in *Proceedings of the Fifteenth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc.

Kennington, J. L. and Helgason, R. V. (1980), *Algorithms for Networking Programming*, New York: Wiley-Interscience.

Lustig, I. J., Marsten, R. E., and Shanno, D. F. (1992), "On Implementing Mehrotra's Predictor-Corrector Interior-Point Method for Linear Programming," *SIAM Journal on Optimization*, 2, 435–449.

Reid, J. K. (1975), *A Sparsity-Exploiting Variant of the Bartels-Golub Decomposition for Linear Programming Bases*, Technical Report Harwell CSS 20, Atomic Energy Research Establishment, Harwell, UK.

Roos, C., Terlaky, T., and Vial, J. (1997), *Theory and Algorithms for Linear Optimization*, Chichester, UK: John Wiley & Sons.

Ryan, D. M. and Osborne, M. R. (1988), "On the Solution of Highly Degenerate Linear Programmes," *Mathematical Programming*, 41, 385–392.

Wright, S. J. (1997), *Primal-Dual Interior-Point Methods*, Philadelphia: SIAM Publications.

Ye, Y. (1996), *Interior Point Algorithms: Theory and Analysis*, New York: John Wiley & Sons.

Chapter 7
The NLP Procedure

Contents

Overview: NLP Procedure **544**
Getting Started: NLP Procedure **546**
Introductory Examples 546
Syntax: NLP Procedure **556**
Functional Summary 556
PROC NLP Statement 559
ARRAY Statement 577
BOUNDS Statement 578
BY Statement 578
CRPJAC Statement 579
DECVAR Statement 580
GRADIENT Statement 580
HESSIAN Statement 581
INCLUDE Statement 581
JACNLC Statement 582
JACOBIAN Statement 582
LABEL Statement 583
LINCON Statement 584
MATRIX Statement 584
MIN, MAX, and LSQ Statements 586
MINQUAD and MAXQUAD Statements 586
NLINCON Statement 588
PROFILE Statement 589
Program Statements 590
Details: NLP Procedure **594**
Criteria for Optimality 594
Optimization Algorithms 597
Finite-Difference Approximations of Derivatives 607
Hessian and CRP Jacobian Scaling 609
Testing the Gradient Specification 609
Termination Criteria 610
Active Set Methods 611
Feasible Starting Point 613
Line-Search Methods 613
Restricting the Step Length 614
Computational Problems 615

Covariance Matrix . . . 618
Input and Output Data Sets . . . 621
Displayed Output . . . 629
Missing Values . . . 631
Computational Resources . . . 632
Memory Limit . . . 634
Rewriting NLP Models for PROC OPTMODEL . . . 634
Examples: NLP Procedure . . . **642**
Example 7.1: Using the DATA= Option . . . 642
Example 7.2: Using the INQUAD= Option . . . 644
Example 7.3: Using the INEST=Option . . . 645
Example 7.4: Restarting an Optimization . . . 647
Example 7.5: Approximate Standard Errors . . . 648
Example 7.6: Maximum Likelihood Weibull Estimation . . . 653
Example 7.7: Simple Pooling Problem . . . 660
Example 7.8: Chemical Equilibrium . . . 669
Example 7.9: Minimize Total Delay in a Network . . . 675
References . . . **680**

Overview: NLP Procedure

The NLP (nonlinear programming) procedure offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function $f(x)$ of n decision variables, $x = (x_1, \ldots, x_n)^T$ with lower and upper bound, linear and nonlinear, equality and inequality constraints. This can be expressed as solving

$$\begin{array}{ll} \min_{x \in \mathcal{R}^n} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i = 1, \ldots, m_e \\ & c_i(x) \geq 0, \quad i = m_e + 1, \ldots, m \\ & l_i \leq x_i \leq u_i, \quad i = 1, \ldots, n \end{array}$$

where f is the objective function, the c_i's are the nonlinear functions, and the l_i's and u_i's are the lower and upper bounds. Problems of this type are found in many settings ranging from optimal control to maximum likelihood estimation.

The NLP procedure provides a number of algorithms for solving this problem that take advantage of a special structure on the objective function and constraints. One example is the quadratic programming problem,

$$\begin{array}{ll} \min\ (\max) & f(x) = \frac{1}{2} x^T G x + g^T x + b \\ \text{subject to} & c_i(x) = 0, \quad i = 1, \ldots, m_e \end{array}$$

where G is an $n \times n$ symmetric matrix, $g = (g_1, \ldots, g_n)^T$ is a vector, b is a scalar, and the $c_i(x)$'s are linear functions.

Another example is the least squares problem:

$$\begin{array}{ll} \min & f(x) = \frac{1}{2}\{f_1^2(x) + \cdots + f_l^2(x)\} \\ \text{subject to} & c_i(x) = 0, \quad i = 1, \ldots, m_e \end{array}$$

where the $c_i(x)$'s are linear functions, and $f_1(x), \ldots, f_l(x)$ are nonlinear functions of x.

The following problems are handled by PROC NLP:

- quadratic programming with an option for sparse problems
- unconstrained minimization/maximization
- constrained minimization/maximization
- linear complementarity problem

The following optimization techniques are supported in PROC NLP:

- Quadratic Active Set Technique
- Trust Region Method
- Newton-Raphson Method with Line Search
- Newton-Raphson Method with Ridging
- Quasi-Newton Methods
- Double Dogleg Method
- Conjugate Gradient Methods
- Nelder-Mead Simplex Method
- Levenberg-Marquardt Method
- Hybrid Quasi-Newton Methods

These optimization techniques require a continuous objective function *f*, and all but one (NMSIMP) require continuous first-order derivatives of the objective function *f*. Some of the techniques also require continuous second-order derivatives. There are three ways to compute derivatives in PROC NLP:

- analytically (using a special derivative compiler), the default method
- via finite-difference approximations
- via user-supplied exact or approximate numerical functions

Nonlinear programs can be input into the procedure in various ways. The objective, constraint, and derivative functions are specified using the programming statements of PROC NLP. In addition, information in SAS data sets can be used to define the structure of objectives and constraints as well as specify constants used in objectives, constraints and derivatives.

PROC NLP uses data sets to input various pieces of information:

- The DATA= data set enables you to specify data shared by all functions involved in a least squares problem.
- The INQUAD= data set contains the arrays appearing in a quadratic programming problem.
- The INEST= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, and simple boundary and general linear constraints.
- The MODEL= data set specifies a model (functions, constraints, derivatives) saved at a previous execution of the NLP procedure.

PROC NLP uses data sets to output various results:

- The OUTEST= data set saves the values of the decision variables, the derivatives, the solution, and the covariance matrix at the solution.
- The OUT= output data set contains variables generated in the program statements defining the objective function as well as selected variables of the DATA= input data set, if available.
- The OUTMODEL= data set saves the programming statements. It can be used to input a model in the MODEL= input data set.

Getting Started: NLP Procedure

The NLP procedure solves general nonlinear programs. It has several optimizers that are tuned to best perform on a particular class of problems. Guidelines for choosing a particular optimizer for a problem can be found in the section "Optimization Algorithms" on page 597.

Regardless of the selected optimizer, it is necessary to specify an objective function and constraints that the optimal solution must satisfy. In PROC NLP, the objective function and the constraints are specified using SAS programming statements that are similar to those used in the SAS DATA step. Some of the differences are discussed in the section "Program Statements" on page 590 and in the section "ARRAY Statement" on page 577. As with any programming language, there are many different ways to specify the same problem. Some are more economical than others.

Introductory Examples

The following introductory examples illustrate how to get started using the NLP procedure.

An Unconstrained Problem

Consider the simple example of minimizing the Rosenbrock function (Rosenbrock 1960):

$$\begin{aligned} f(x) &= \frac{1}{2}\{100(x_2 - x_1^2)^2 + (1 - x_1)^2\} \\ &= \frac{1}{2}\{f_1^2(x) + f_2^2(x)\}, \quad x = (x_1, x_2) \end{aligned}$$

The minimum function value is $f(x^*) = 0$ at $x^* = (1, 1)$. This problem does not have any constraints.

The following statements can be used to solve this problem:

```
proc nlp;
   min f;
   decvar x1 x2;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
   f  = .5 * (f1 * f1 + f2 * f2);
run;
```

The MIN statement identifies the symbol `f` that characterizes the objective function in terms of `f1` and `f2`, and the DECVAR statement names the decision variables `x1` and `x2`. Because there is no explicit optimizing algorithm option specified (TECH=), PROC NLP uses the Newton-Raphson method with ridging, the default algorithm when there are no constraints.

A better way to solve this problem is to take advantage of the fact that f is a sum of squares of f_1 and f_2 and to treat it as a least squares problem. Using the LSQ statement instead of the MIN statement tells the procedure that this is a least squares problem, which results in the use of one of the specialized algorithms for solving least squares problems (for example, Levenberg-Marquardt).

```
proc nlp;
   lsq f1 f2;
   decvar x1 x2;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
run;
```

The LSQ statement results in the minimization of a function that is the sum of squares of functions that appear in the LSQ statement. The least squares specification is preferred because it enables the procedure to exploit the structure in the problem for numerical stability and performance.

PROC NLP displays the iteration history and the solution to this least squares problem as shown in Figure 7.1. It shows that the solution has $x_1 = 1$ and $x_2 = 1$. As expected in an unconstrained problem, the gradient at the solution is very close to 0.

Figure 7.1 Least Squares Minimization

PROC NLP: Least Squares Minimization

Levenberg-Marquardt Optimization

Scaling Update of More (1978)

Parameter Estimates	2
Functions (Observations)	2

Optimization Start			
Active Constraints	0	**Objective Function**	0.5545849354
Max Abs Gradient Element	16.982372536	**Radius**	299.60285345

Figure 7.1 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	0	2	0	0.04596	0.5086	6.0635	0	0.917
2	0	3	0	4.1662E-30	0.0460	2.89E-15	0	1.000

Optimization Results			
Iterations	2	Function Calls	4
Jacobian Calls	3	Active Constraints	0
Objective Function	4.166172E-30	Max Abs Gradient Element	2.88658E-15
Lambda	0	Actual Over Pred Change	1
Radius	0.6063486947		

ABSGCONV convergence criterion satisfied.

PROC NLP: Least Squares Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	x1	1.000000	-2.88658E-15
2	x2	1.000000	0

Value of Objective Function = 4.166172E-30

Boundary Constraints on the Decision Variables

Bounds on the decision variables can be used. Suppose, for example, that it is necessary to constrain the decision variables in the previous example to be less than 0.5. That can be done by adding a BOUNDS statement.

```
proc nlp;
   lsq f1 f2;
   decvar x1 x2;
   bounds x1-x2 <= .5;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
run;
```

The solution in Figure 7.2 shows that the decision variables meet the constraint bounds.

Figure 7.2 Least Squares with Bounds Solution

PROC NLP: Least Squares Minimization

Levenberg-Marquardt Optimization

PROC NLP: Least Squares Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	x1	0.500000	-0.500000	Upper BC
2	x2	0.250000	0	

Linear Constraints on the Decision Variables

More general linear equality or inequality constraints of the form

$$\sum_{j=1}^{n} a_{ij} x_j \; \{\leq \mid = \mid \geq\} \; b_i \quad \text{for } i = 1, \ldots, m$$

can be specified in a LINCON statement. For example, suppose that in addition to the bounds constraints on the decision variables it is necessary to guarantee that the sum $x_1 + x_2$ is less than or equal to 0.6. That can be achieved by adding a LINCON statement:

```
proc nlp;
   lsq f1 f2;
   decvar x1 x2;
   bounds x1-x2 <= .5;
   lincon x1 + x2 <= .6;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
run;
```

The output in Figure 7.3 displays the iteration history and the convergence criterion.

Figure 7.3 Least Squares with Bounds and Linear Constraints Iteration History

PROC NLP: Least Squares Minimization

Value of Objective Function = 0.3453874109

PROC NLP: Least Squares Minimization

Levenberg-Marquardt Optimization

Parameter Estimates	2
Functions (Observations)	2
Lower Bounds	0
Upper Bounds	2
Linear Constraints	1

Figure 7.3 *continued*

Optimization Start			
Active Constraints (+)	0	**Objective Function**	0.3453874109
Max Abs Gradient Element	5.6534063515	**Radius**	69.030770145

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	0	5	0	0.16789	0.1775	0.4576	166.9	0.522
2	1	7	1	0.16672	0.00117	0.2190	0.00471	0.0117
3	1	8	1	0.16658	0.000140	0.000508	0	0.998
4	1	9	1	0.16658	7.52E-10	9.253E-7	0	0.998

Optimization Results			
Iterations	4	**Function Calls**	10
Jacobian Calls	6	**Active Constraints**	1
Objective Function	0.1665792899	**Max Abs Gradient Element**	9.2529401E-7
Lambda	0	**Actual Over Pred Change**	0.9981767757
Radius	0.0000776394		

GCONV convergence criterion satisfied.

Figure 7.4 shows that the solution satisfies the linear constraint. Note that the procedure displays the active constraints (the constraints that are tight) at optimality.

Figure 7.4 Least Squares with Bounds and Linear Constraints Solution

PROC NLP: Least Squares Minimization

Scaling Update of More (1978)

PROC NLP: Least Squares Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	**x1**	0.423645	-0.312000
2	**x2**	0.176355	-0.312000

Linear Constraints Evaluated at Solution

1 ACT 8.3267E-17 = 0.6000 - 1.0000 * x1 - 1.0000 * x2

Nonlinear Constraints on the Decision Variables

More general nonlinear equality or inequality constraints can be specified using an NLINCON statement. Consider the least squares problem with the additional constraint

$$x_1^2 - 2x_2 \geq 0$$

This constraint is specified by a new function `c1` constrained to be greater than or equal to 0 in the NLINCON statement. The function `c1` is defined in the programming statements.

```
proc nlp tech=QUANEW;
   min f;
   decvar x1 x2;
   bounds x1-x2 <= .5;
   lincon x1 + x2 <= .6;
   nlincon c1 >= 0;

   c1 = x1 * x1 - 2 * x2;

   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;

   f = .5 * (f1 * f1 + f2 * f2);
run;
```

Figure 7.5 shows the iteration history, and Figure 7.6 shows the solution to this problem.

Figure 7.5 Least Squares with Bounds, Linear and Nonlinear Constraints, Iteration History

PROC NLP: Nonlinear Minimization

Dual Quasi-Newton Optimization

Modified VMCWD Algorithm of Powell (1978, 1982)

Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

Lagrange Multiplier Update of Powell(1982)

Parameter Estimates	2
Lower Bounds	0
Upper Bounds	2
Linear Constraints	1
Nonlinear Constraints	1

Optimization Start			
Objective Function	2.750048788	**Maximum Constraint Violation**	0
Maximum Gradient of the Lagran Func	19.528027002		

Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Maximum Gradient Element of the Lagrange Function
1	0	9	1.21827	0	0.8823	0.437	5.845
2	0	10	0.78787	0	0.5262	1.000	2.616
3	0	12	0.72214	0	0.2500	0.147	2.849
4	0	13	0.55450	0	0.1977	1.000	2.509
5	0	14	0.42378	0	0.2537	1.000	0.789
6	0	16	0.39842	0	0.1574	0.114	0.760
7	0	18	0.35979	0	0.0649	0.366	0.320
8	0	19	0.35429	0	0.0548	1.000	1.683
9	0	20	0.33415	0	0.00758	1.000	0.119
10	0	21	0.33026	0	0.000455	1.000	0.121
11	0	22	0.33005	0	0.000044	1.000	0.00221
12	0	23	0.33003	0	5.683E-8	1.000	0.00012

Optimization Results			
Iterations	12	**Function Calls**	24
Gradient Calls	15	**Active Constraints**	0
Objective Function	0.330030744	**Maximum Constraint Violation**	0
Maximum Projected Gradient	3.0494342639	**Value Lagrange Function**	0.330030744
Maximum Gradient of the Lagran Func	3.0494342639	**Slope of Search Direction**	-5.683122E-8

Figure 7.6 Least Squares with Bounds, Linear and Nonlinear Constraints, Solution

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function
1	x1	0.246953	0.753017	-0.000013854
2	x2	0.030493	-3.049292	-0.000003421

Value of Objective Function = 0.3300307303

Value of Lagrange Function = 0.3300307155

Linear Constraints Evaluated at Solution

1 0.32255 = 0.6000 - 1.0000 * x1 - 1.0000 * x2

Values of Nonlinear Constraints

	Constraint		Value	Residual	Lagrange Multiplier		
[	2]	c1_G	9.699E-9	9.699E-9	1.5246	Active	NLIC

Not all of the optimization methods support nonlinear constraints. In particular the Levenberg-Marquardt method, the default for LSQ, does not support nonlinear constraints. (For more information about the particular algorithms, see the section "Optimization Algorithms" on page 597.) The Quasi-Newton method is the prime choice for solving nonlinear programs with nonlinear constraints. The option TECH=QUANEW in the PROC NLP statement causes the Quasi-Newton method to be used.

A Simple Maximum Likelihood Example

The following is a very simple example of a maximum likelihood estimation problem with the log likelihood function:

$$l(\mu,\sigma) = -\log(\sigma) - \frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

The maximum likelihood estimates of the parameters μ and σ form the solution to

$$\max_{\mu,\sigma>0} \sum_i l_i(\mu,\sigma)$$

where

$$l_i(\mu,\sigma) = -\log(\sigma) - \frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2$$

In the following DATA step, values for x are input into SAS data set X; this data set provides the values of x_i.

```
data x;
input x @@;
datalines;
1 3 4 5 7
;
```

In the following statements, the DATA=X specification drives the building of the objective function. When each observation in the DATA=X data set is read, a new term $l_i(\mu, \sigma)$ using the value of x_i is added to the objective function LOGLIK specified in the MAX statement.

```
proc nlp data=x vardef=n covariance=h pcov phes;
   profile mean sigma / alpha=.5 .1 .05 .01;
   max loglik;
   parms mean=0, sigma=1;
   bounds sigma > 1e-12;
   loglik=-0.5*((x-mean)/sigma)**2-log(sigma);
run;
```

After a few iterations of the default Newton-Raphson optimization algorithm, PROC NLP produces the results shown in Figure 7.7.

Figure 7.7 Maximum Likelihood Estimates

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	-1.33149E-10
2	sigma	2.000000	0.632456	3.162278	0.025031	5.6064147E-9

Value of Objective Function = -5.965735903

In unconstrained maximization, the gradient (that is, the vector of first derivatives) at the solution must be very close to zero and the Hessian matrix at the solution (that is, the matrix of second derivatives) must have nonpositive eigenvalues. The Hessian matrix is displayed in Figure 7.8.

Figure 7.8 Hessian Matrix

PROC NLP: Nonlinear Maximization

Hessian Matrix

	mean	sigma
mean	-1.250000003	1.331489E-10
sigma	1.331489E-10	-2.500000014

Determinant = 3.1250000245

Matrix has Only Negative Eigenvalues

Under reasonable assumptions, the approximate standard errors of the estimates are the square roots of the diagonal elements of the covariance matrix of the parameter estimates, which (because of the COV=H specification) is the same as the inverse of the Hessian matrix. The covariance matrix is shown in Figure 7.9.

Figure 7.9 Covariance Matrix

PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	mean	**sigma**
mean	0.7999999982	4.260766E-11
sigma	4.260766E-11	0.3999999978

Factor sigm = 1

Determinant = 0.3199999975

Matrix has 2 Positive Eigenvalue(s)

The PROFILE statement computes the values of the profile likelihood confidence limits on SIGMA and MEAN, as shown in Figure 7.10.

Figure 7.10 Confidence Limits

PROC NLP: Nonlinear Maximization

Wald and PL Confidence Limits							
N	**Parameter**	**Estimate**	**Alpha**	**Profile Likelihood Confidence Limits**		**Wald Confidence Limits**	
1	**mean**	4.000000	0.500000	3.384431	4.615569	3.396718	4.603282
1	**mean**	.	0.100000	2.305716	5.694284	2.528798	5.471202
1	**mean**	.	0.050000	1.849538	6.150462	2.246955	5.753045
1	**mean**	.	0.010000	0.670351	7.329649	1.696108	6.303892
2	**sigma**	2.000000	0.500000	1.638972	2.516078	1.573415	2.426585
2	**sigma**	.	0.100000	1.283506	3.748633	0.959703	3.040297
2	**sigma**	.	0.050000	1.195936	4.358321	0.760410	3.239590
2	**sigma**	.	0.010000	1.052584	6.064107	0.370903	3.629097

Syntax: NLP Procedure

Below are statements used in PROC NLP, listed in alphabetical order as they appear in the text that follows.

PROC NLP *options* ;
ARRAY *function names* ;
BOUNDS *boundary constraints* ;
BY *variables* ;
CRPJAC *variables* ;
DECVAR *function names* ;
GRADIENT *variables* ;
HESSIAN *variables* ;
INCLUDE *model files* ;
JACNLC *variables* ;
JACOBIAN *function names* ;
LABEL *decision variable labels* ;
LINCON *linear constraints* ;
MATRIX *matrix specification* ;
MIN, MAX, or LSQ *function names* ;
MINQUAD or MAXQUAD *matrix, vector, or number* ;
NLINCON *nonlinear constraints* ;
PROFILE *profile specification* ;
Program Statements *;* ;

Functional Summary

The following table outlines the options in PROC NLP classified by function.

Table 7.1 Functional Summary

Description	Statement	Option
Input Data Set Options:		
Input data set	PROC NLP	DATA=
Initial values and constraints	PROC NLP	INEST=
Quadratic objective function	PROC NLP	INQUAD=
Program statements	PROC NLP	MODEL=
Skip missing value observations	PROC NLP	NOMISS
Output Data Set Options:		
Variables and derivatives	PROC NLP	OUT=
Result parameter values	PROC NLP	OUTEST=
Program statements	PROC NLP	OUTMODEL=
Combine various OUT. . . statements	PROC NLP	OUTALL
CRP Jacobian in the OUTEST= data set	PROC NLP	OUTCRPJAC
Derivatives in the OUT= data set	PROC NLP	OUTDER=

Description	Statement	Option
Grid in the OUTEST= data set	PROC NLP	OUTGRID
Hessian in the OUTEST= data set	PROC NLP	OUTHESSIAN
Iterative output in the OUTEST= data set	PROC NLP	OUTITER
Jacobian in the OUTEST= data set	PROC NLP	OUTJAC
NLC Jacobian in the OUTEST= data set	PROC NLP	OUTNLCJAC
Time in the OUTEST= data set	PROC NLP	OUTTIME
Optimization Options:		
Minimization method	PROC NLP	TECH=
Update technique	PROC NLP	UPDATE=
Version of optimization technique	PROC NLP	VERSION=
Line-search method	PROC NLP	LINESEARCH=
Line-search precision	PROC NLP	LSPRECISION=
Type of Hessian scaling	PROC NLP	HESCAL=
Start for approximated Hessian	PROC NLP	INHESSIAN=
Iteration number for update restart	PROC NLP	RESTART=
Initial Value Options:		
Produce best grid points	PROC NLP	BEST=
Infeasible points in grid search	PROC NLP	INFEASIBLE
Pseudorandom initial values	PROC NLP	RANDOM=
Constant initial values	PROC NLP	INITIAL=
Derivative Options:		
Finite-difference derivatives	PROC NLP	FD=
Finite-difference derivatives	PROC NLP	FDHESSIAN=
Compute finite-difference interval	PROC NLP	FDINT=
Use only diagonal of Hessian	PROC NLP	DIAHES
Test gradient specification	PROC NLP	GRADCHECK=
Constraint Options:		
Range for active constraints	PROC NLP	LCEPSILON=
LM tolerance for deactivating	PROC NLP	LCDEACT=
Tolerance for dependent constraints	PROC NLP	LCSINGULAR=
Sum all observations for continuous functions	NLINCON	/ SUMOBS
Evaluate each observation for continuous functions	NLINCON	/ EVERYOBS
Termination Criteria Options:		
Maximum number of function calls	PROC NLP	MAXFUNC=
Maximum number of iterations	PROC NLP	MAXITER=
Minimum number of iterations	PROC NLP	MINITER=
Upper limit on real time	PROC NLP	MAXTIME=
Absolute function convergence criterion	PROC NLP	ABSCONV=
Absolute function convergence criterion	PROC NLP	ABSFCONV=
Absolute gradient convergence criterion	PROC NLP	ABSGCONV=

Description	Statement	Option
Absolute parameter convergence criterion	PROC NLP	ABSXCONV=
Relative function convergence criterion	PROC NLP	FCONV=
Relative function convergence criterion	PROC NLP	FCONV2=
Relative gradient convergence criterion	PROC NLP	GCONV=
Relative gradient convergence criterion	PROC NLP	GCONV2=
Relative parameter convergence criterion	PROC NLP	XCONV=
Used in FCONV, GCONV criterion	PROC NLP	FSIZE=
Used in XCONV criterion	PROC NLP	XSIZE=
Covariance Matrix Options:		
Type of covariance matrix	PROC NLP	COV=
σ^2 factor of COV matrix	PROC NLP	SIGSQ=
Determine factor of COV matrix	PROC NLP	VARDEF=
Absolute singularity for inertia	PROC NLP	ASINGULAR=
Relative M singularity for inertia	PROC NLP	MSINGULAR=
Relative V singularity for inertia	PROC NLP	VSINGULAR=
Threshold for Moore-Penrose inverse	PROC NLP	G4=
Tolerance for singular COV matrix	PROC NLP	COVSING=
Profile confidence limits	PROC NLP	CLPARM=
Printed Output Options:		
Display (almost) all printed output	PROC NLP	PALL
Suppress all printed output	PROC NLP	NOPRINT
Reduce some default output	PROC NLP	PSHORT
Reduce most default output	PROC NLP	PSUMMARY
Display initial values and gradients	PROC NLP	PINIT
Display optimization history	PROC NLP	PHISTORY
Display Jacobian matrix	PROC NLP	PJACOBI
Display crossproduct Jacobian matrix	PROC NLP	PCRPJAC
Display Hessian matrix	PROC NLP	PHESSIAN
Display Jacobian of nonlinear constraints	PROC NLP	PNLCJAC
Display values of grid points	PROC NLP	PGRID
Display values of functions in LSQ, MIN, MAX	PROC NLP	PFUNCTION
Display approximate standard errors	PROC NLP	PSTDERR
Display covariance matrix	PROC NLP	PCOV
Display eigenvalues for covariance matrix	PROC NLP	PEIGVAL
Print code evaluation problems	PROC NLP	PERROR
Print measures of real time	PROC NLP	PTIME
Display model program, variables	PROC NLP	LIST
Display compiled model program	PROC NLP	LISTCODE
Step Length Options:		
Damped steps in line search	PROC NLP	DAMPSTEP=
Maximum trust region radius	PROC NLP	MAXSTEP=
Initial trust region radius	PROC NLP	INSTEP=

Description	Statement	Option
Profile Point and Confidence Interval Options:		
Factor relating discrepancy function to χ^2 quantile	PROFILE	FFACTOR=
Scale for y values written to OUTEST= data set	PROFILE	FORCHI=
Upper bound for confidence limit search	PROFILE	FEASRATIO=
Write all confidence limit parameter estimates to OUTEST= data set	PROFILE	OUTTABLE
Miscellaneous Options:		
Number of accurate digits in objective function	PROC NLP	FDIGITS=
Number of accurate digits in nonlinear constraints	PROC NLP	CDIGITS=
General singularity criterion	PROC NLP	SINGULAR=
Do not compute inertia of matrices	PROC NLP	NOEIGNUM
Check optimality in neighborhood	PROC NLP	OPTCHECK=

PROC NLP Statement

PROC NLP *options* **;**

This statement invokes the NLP procedure. The following options are used with the PROC NLP statement.

ABSCONV=*r*

ABSTOL=*r*

specifies an absolute function convergence criterion. For minimization (maximization), termination requires $f(x^{(k)}) \leq (\geq)\ r$. The default value of ABSCONV is the negative (positive) square root of the largest double precision value.

ABSFCONV=*r*[*n*]

ABSFTOL=*r*[*n*]

specifies an absolute function convergence criterion. For all techniques except NMSIMP, termination requires a small change of the function value in successive iterations:

$$|f(x^{(k-1)}) - f(x^{(k)})| \leq r$$

For the NMSIMP technique the same formula is used, but $x^{(k)}$ is defined as the vertex with the lowest function value, and $x^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is $r = 0$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ABSGCONV=*r*[*n*]

ABSGTOL=*r*[*n*]

specifies the absolute gradient convergence criterion. Termination requires the maximum absolute gradient element to be small:

$$\max_j |g_j(x^{(k)})| \leq r$$

This criterion is not used by the NMSIMP technique. The default value is *r*=1E−5. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ABSXCONV=*r*[*n*]

ABSXTOL=*r*[*n*]

specifies the absolute parameter convergence criterion. For all techniques except NMSIMP, termination requires a small Euclidean distance between successive parameter vectors:

$$\| x^{(k)} - x^{(k-1)} \|_2 \leq r$$

For the NMSIMP technique, termination requires either a small length $\alpha^{(k)}$ of the vertices of a restart simplex

$$\alpha^{(k)} \leq r$$

or a small simplex size

$$\delta^{(k)} \leq r$$

where the simplex size $\delta^{(k)}$ is defined as the L_1 distance of the simplex vertex $y^{(k)}$ with the smallest function value to the other n simplex points $x_l^{(k)} \neq y^{(k)}$:

$$\delta^{(k)} = \sum_{x_l \neq y} \| x_l^{(k)} - y^{(k)} \|_1$$

The default value is *r*=1E−4 for the COBYLA NMSIMP technique, *r*=1E−8 for the standard NMSIMP technique, and *r*=0 otherwise. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ASINGULAR=*r*

ASING=*r*

specifies an absolute singularity criterion for measuring singularity of Hessian and crossproduct Jacobian and their projected forms, which may have to be converted to compute the covariance matrix. The default is the square root of the smallest positive double precision value. For more information, see the section "Covariance Matrix" on page 618.

BEST=*i*

produces the *i* best grid points only. This option not only restricts the output, it also can significantly reduce the computation time needed for sorting the grid point information.

CDIGITS=*r*

specifies the number of accurate digits in nonlinear constraint evaluations. Fractional values such as CDIGITS=4.7 are allowed. The default value is $r = -\log_{10}(\epsilon)$, where ϵ is the machine precision. The value of r is used to compute the interval length h for the computation of finite-difference approximations of the Jacobian matrix of nonlinear constraints.

CLPARM= PL | WALD | BOTH

is similar to but not the same as that used by other SAS procedures. Using CLPARM=BOTH is equivalent to specifying

```
PROFILE / ALPHA=0.5 0.1 0.05 0.01 OUTTABLE;
```

The CLPARM=BOTH option specifies that profile confidence limits (PL CLs) for all parameters and for $\alpha = .5, .1, .05, .01$ are computed and displayed or written to the OUTEST= data set. Computing the profile confidence limits for all parameters can be very expensive and should be avoided when a difficult optimization problem or one with many parameters is solved. The OUTTABLE option is valid only when an OUTEST= data set is specified in the PROC NLP statement. For CLPARM=BOTH, the table of displayed output contains the Wald confidence limits computed from the standard errors as well as the PL CLs. The Wald confidence limits are not computed (displayed or written to the OUTEST= data set) unless the approximate covariance matrix of parameters is computed.

COV= 1 | 2 | 3 | 4 | 5 | 6 | M | H | J | B | E | U

COVARIANCE= 1 | 2 | 3 | 4 | 5 | 6 | M | H | J | B | E | U

specifies one of six formulas for computing the covariance matrix. For more information, see the section "Covariance Matrix" on page 618.

COVSING=*r*

specifies a threshold $r > 0$ that determines whether the eigenvalues of a singular Hessian matrix or crossproduct Jacobian matrix are considered to be zero. For more information, see the section "Covariance Matrix" on page 618.

DAMPSTEP[=*r*]

DS[=*r*]

specifies that the initial step length value $\alpha^{(0)}$ for each line search (used by the QUANEW, HYQUAN, CONGRA, or NEWRAP technique) cannot be larger than r times the step length value used in the former iteration. If the DAMPSTEP option is specified but r is not specified, the default is r=2. The DAMPSTEP=r option can prevent the line-search algorithm from repeatedly stepping into regions where some objective functions are difficult to compute or where they could lead to floating point overflows during the computation of objective functions and their derivatives. The DAMPSTEP=r option can save time-costly function calls during the line searches of objective functions that result in very small steps. For more information, see the section "Restricting the Step Length" on page 614.

DATA=*SAS-data-set*

allows variables from the specified data set to be used in the specification of the objective function f. For more information, see the section "DATA= Input Data Set" on page 621.

DIAHES

specifies that only the diagonal of the Hessian or crossproduct Jacobian is used. This saves function evaluations but may slow the convergence process considerably. Note that the DIAHES option refers to both the Hessian and the crossproduct Jacobian when using the LSQ statement. When derivatives are specified using the HESSIAN or CRPJAC statement, these statements must refer only to the n diagonal derivative elements (otherwise, the $n(n+1)/2$ derivatives of the lower triangle must be specified). The DIAHES option is ignored if a quadratic programming with a constant Hessian is specified by TECH=QUADAS or TECH=LICOMP.

FCONV=*r*[*n*]

FTOL=*r*[*n*]

specifies the relative function convergence criterion. For all techniques except NMSIMP, termination requires a small relative change of the function value in successive iterations:

$$\frac{|f(x^{(k)}) - f(x^{(k-1)})|}{\max(|f(x^{(k-1)})|, FSIZE)} \leq r$$

where *FSIZE* is defined by the FSIZE= option. For the NMSIMP technique, the same formula is used, but $x^{(k)}$ is defined as the vertex with the lowest function value, and $x^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is $r = 10^{-\text{FDIGITS}}$ where *FDIGITS* is the value of the FDIGITS= option. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

FCONV2=*r*[*n*]

FTOL2=*r*[*n*]

FCONV2= option specifies another function convergence criterion. For least squares problems and all techniques except NMSIMP, termination requires a small predicted reduction

$$df^{(k)} \approx f(x^{(k)}) - f(x^{(k)} + s^{(k)})$$

of the objective function. The predicted reduction

$$\begin{aligned} df^{(k)} &= -g^{(k)T}s^{(k)} - \frac{1}{2}s^{(k)T}G^{(k)}s^{(k)} \\ &= -\frac{1}{2}s^{(k)T}g^{(k)} \\ &\leq r \end{aligned}$$

is based on approximating the objective function f by the first two terms of the Taylor series and substituting the Newton step

$$s^{(k)} = -G^{(k)-1}g^{(k)}$$

For the NMSIMP technique, termination requires a small standard deviation of the function values of the $n+1$ simplex vertices $x_l^{(k)}$, $l = 0, \ldots, n$,

$$\sqrt{\frac{1}{n+1}\sum_l (f(x_l^{(k)}) - \overline{f}(x^{(k)}))^2} \leq r$$

where $\overline{f}(x^{(k)}) = \frac{1}{n+1}\sum_l f(x_l^{(k)})$. If there are n_{act} boundary constraints active at $x^{(k)}$, the mean and standard deviation are computed only for the $n + 1 - n_{act}$ unconstrained vertices. The default value is *r*=1E−6 for the NMSIMP technique and the QUANEW technique with nonlinear constraints, and *r*=0 otherwise. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

FD[=FORWARD | CENTRAL | *number*]

specifies that all derivatives be computed using finite-difference approximations. The following specifications are permitted:

FD=FORWARD uses forward differences.

FD=CENTRAL uses central differences.

FD=*number* uses central differences for the initial and final evaluations of the gradient, Jacobian, and Hessian. During iteration, start with forward differences and switch to a corresponding central-difference formula during the iteration process when one of the following two criteria is satisfied:

- The absolute maximum gradient element is less than or equal to *number* times the ABSGCONV threshold.
- The term left of the GCONV criterion is less than or equal to $\max(1.0\mathrm{E}-6, number\times$ GCONV threshold). The 1.0E−6 ensures that the switch is done, even if you set the GCONV threshold to zero.

FD is equivalent to FD=100.

Note that the FD and FDHESSIAN options cannot apply at the same time. The FDHESSIAN option is ignored when only first-order derivatives are used, for example, when the LSQ statement is used and the HESSIAN is not explicitly needed (displayed or written to a data set). For more information, see the section "Finite-Difference Approximations of Derivatives" on page 607.

FDHESSIAN[=FORWARD | CENTRAL]

FDHES[=FORWARD | CENTRAL]

FDH[=FORWARD | CENTRAL]

specifies that second-order derivatives be computed using finite-difference approximations based on evaluations of the gradients.

FDHESSIAN=FORWARD	uses forward differences.
FDHESSIAN=CENTRAL	uses central differences.
FDHESSIAN	uses forward differences for the Hessian except for the initial and final output.

Note that the FD and FDHESSIAN options cannot apply at the same time. For more information, see the section "Finite-Difference Approximations of Derivatives" on page 607

FDIGITS=*r*

specifies the number of accurate digits in evaluations of the objective function. Fractional values such as FDIGITS=4.7 are allowed. The default value is $r = -\log_{10}(\epsilon)$, where ϵ is the machine precision. The value of r is used to compute the interval length h for the computation of finite-difference approximations of the derivatives of the objective function and for the default value of the FCONV= option.

FDINT= OBJ | CON | ALL

specifies how the finite-difference intervals h should be computed. For FDINT=OBJ, the interval h is based on the behavior of the objective function; for FDINT=CON, the interval h is based on the behavior of the nonlinear constraints functions; and for FDINT=ALL, the interval h is based on the behavior of the objective function and the nonlinear constraints functions. For more information, see the section "Finite-Difference Approximations of Derivatives" on page 607.

FSIZE=*r*

specifies the FSIZE parameter of the relative function and relative gradient termination criteria. The default value is $r = 0$. For more details, refer to the FCONV= and GCONV= options.

G4=*n*

is used when the covariance matrix is singular. The value $n > 0$ determines which generalized inverse is computed. The default value of n is 60. For more information, see the section "Covariance Matrix" on page 618.

GCONV=*r*[*n*]

GTOL=*r*[*n*]

specifies the relative gradient convergence criterion. For all techniques except the CONGRA and NMSIMP techniques, termination requires that the normalized predicted function reduction is small:

$$\frac{g(x^{(k)})^T [G^{(k)}]^{-1} g(x^{(k)})}{\max(|f(x^{(k)})|, FSIZE)} \leq r$$

where *FSIZE* is defined by the FSIZE= option. For the CONGRA technique (where a reliable Hessian estimate G is not available),

$$\frac{\| g(x^{(k)}) \|_2^2 \quad \| s(x^{(k)}) \|_2}{\| g(x^{(k)}) - g(x^{(k-1)}) \|_2 \max(|f(x^{(k)})|, \text{FSIZE})} \leq r$$

is used. This criterion is not used by the NMSIMP technique. The default value is r=1E−8. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

GCONV2=*r*[*n*]

GTOL2=*r*[*n*]

GCONV2= option specifies another relative gradient convergence criterion,

$$\max_j \frac{|g_j(x^{(k)})|}{\sqrt{f(x^{(k)}) G^{(k)}_{j,j}}} \leq r$$

This option is valid only when using the TRUREG, LEVMAR, NRRIDG, and NEWRAP techniques on least squares problems. The default value is $r = 0$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

GRADCHECK[= NONE | FAST | DETAIL]

GC[= NONE | FAST | DETAIL]

Specifying GRADCHECK=DETAIL computes a test vector and test matrix to check whether the gradient g specified by a GRADIENT statement (or indirectly by a JACOBIAN statement) is appropriate for the function f computed by the program statements. If the specification of the first derivatives is correct, the elements of the test vector and test matrix should be relatively small. For very large optimization problems, the algorithm can be too expensive in terms of computer time and memory. If the GRADCHECK option is not specified, a fast derivative test identical to the GRADCHECK=FAST specification is performed by default. It is possible to suppress the default derivative test by specifying GRADCHECK=NONE. For more information, see the section "Testing the Gradient Specification" on page 609.

HESCAL= 0 | 1 | 2 | 3

HS= 0 | 1 | 2 | 3

specifies the scaling version of the Hessian or crossproduct Jacobian matrix used in NRRIDG, TRUREG, LEVMAR, NEWRAP, or DBLDOG optimization. If the value of the HESCAL= option is not equal to zero, the first iteration and each restart iteration sets the diagonal scaling matrix $D^{(0)} = \mathrm{diag}(d_i^{(0)})$:

$$d_i^{(0)} = \sqrt{\max(|G_{i,i}^{(0)}|, \epsilon)}$$

where $G_{i,i}^{(0)}$ are the diagonal elements of the Hessian or crossproduct Jacobian matrix. In all other iterations, the diagonal scaling matrix $D^{(0)} = \mathrm{diag}(d_i^{(0)})$ is updated depending on the HESCAL= option:

HESCAL=0 specifies that no scaling is done

HESCAL=1 specifies the Moré (1978) scaling update:

$$d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

HESCAL=2 specifies the Dennis, Gay, and Welsch (1981) scaling update:

$$d_i^{(k+1)} = \max\left(0.6d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

HESCAL=3 specifies that d_i is reset in each iteration:

$$d_i^{(k+1)} = \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}$$

where ϵ is the relative machine precision. The default value is HESCAL=1 for LEVMAR minimization and HESCAL=0 otherwise. Scaling of the Hessian or crossproduct Jacobian matrix can be time-consuming in the case where general linear constraints are active.

INEST=*SAS-data-set*

INVAR=*SAS-data-set*

ESTDATA=*SAS-data-set*

can be used to specify the initial values of the parameters defined in a DECVAR statement as well as simple boundary constraints and general linear constraints. The INEST= data set can contain additional variables with names corresponding to constants used in the program statements. At the beginning of each run of PROC NLP, the values of the constants are read from the PARMS observation, initializing the constants in the program statements. For more information, see the section "INEST= Input Data Set" on page 621.

INFEASIBLE

IFP

specifies that the function values of both feasible and infeasible grid points are to be computed, displayed, and written to the OUTEST= data set, although only the feasible grid points are candidates for the starting point $x^{(0)}$. This option enables you to explore the shape of the objective function of points surrounding the feasible region. For the output, the grid points are sorted first with decreasing values of the maximum constraint violation. Points with the same value of the maximum constraint

violation are then sorted with increasing (minimization) or decreasing (maximization) value of the objective function. Using the BEST= option restricts only the number of best grid points in the displayed output, not those in the data set. The INFEASIBLE option affects both the displayed output and the output saved to the OUTEST= data set. The OUTGRID option can be used to write the grid points and their function values to an OUTEST= data set. After small modifications (deleting unneeded information), this data set can be used with the G3D procedure of SAS/GRAPH to generate a three-dimensional surface plot of the objective function depending on two selected parameters. For more information on grids, see the section "DECVAR Statement" on page 580.

INHESSIAN[=*r*]
INHESS[=*r*]

specifies how the initial estimate of the approximate Hessian is defined for the quasi-Newton techniques QUANEW, DBLDOG, and HYQUAN. There are two alternatives:

- The $= r$ specification is not used: the initial estimate of the approximate Hessian is set to the true Hessian or crossproduct Jacobian at $x^{(0)}$.
- The $= r$ specification is used: the initial estimate of the approximate Hessian is set to the multiple of the identity matrix rI.

By default, if INHESSIAN=*r* is not specified, the initial estimate of the approximate Hessian is set to the multiple of the identity matrix rI, where the scalar r is computed from the magnitude of the initial gradient. For most applications, this is a sufficiently good first approximation.

INITIAL=*r*

specifies a value r as the common initial value for all parameters for which no other initial value assignments by the DECVAR statement or an INEST= data set are made.

INQUAD=*SAS-data-set*

can be used to specify (the nonzero elements of) the matrix H, the vector g, and the scalar c of a quadratic programming problem, $f(x) = \frac{1}{2}x^T Hx + g^T x + c$. This option cannot be used together with the NLINCON statement. Two forms (*dense* and *sparse*) of the INQUAD= data set can be used. For more information, see the section "INQUAD= Input Data Set" on page 622.

INSTEP=*r*

For highly nonlinear objective functions, such as the EXP function, the default initial radius of the trust region algorithms TRUREG, DBLDOG, or LEVMAR or the default step length of the line-search algorithms can result in arithmetic overflows. If this occurs, decreasing values of $0 < r < 1$ should be specified, such as INSTEP=1E−1, INSTEP=1E−2, INSTEP=1E−4, and so on, until the iteration starts successfully.

- For trust region algorithms (TRUREG, DBLDOG, LEVMAR), the INSTEP= option specifies a factor $r > 0$ for the initial radius $\Delta^{(0)}$ of the trust region. The default initial trust region radius is the length of the scaled gradient. This step corresponds to the default radius factor of $r = 1$.
- For line-search algorithms (NEWRAP, CONGRA, QUANEW, HYQUAN), the INSTEP= option specifies an upper bound for the initial step length for the line search during the first five iterations. The default initial step length is $r = 1$.
- For the Nelder-Mead simplex algorithm (NMSIMP), the INSTEP=*r* option defines the size of the initial simplex.

For more details, see the section "Computational Problems" on page 615.

LCDEACT=*r*

LCD=*r*

specifies a threshold r for the Lagrange multiplier that decides whether an active inequality constraint remains active or can be deactivated. For a maximization (minimization), an active inequality constraint can be deactivated only if its Lagrange multiplier is greater (less) than the threshold value r. For maximization, r must be greater than zero; for minimization, r must be smaller than zero. The default value is

$$r = \pm \min(0.01, \max(0.1 \times ABSGCONV, 0.001 \times gmax^{(k)}))$$

where the $+$ stands for maximization, the $-$ for minimization, ABSGCONV is the value of the absolute gradient criterion, and $gmax^{(k)}$ is the maximum absolute element of the (projected) gradient $g^{(k)}$ or $Z^T g^{(k)}$.

LCEPSILON=*r*

LCEPS=*r*

LCE=*r*

specifies the range $r > 0$ for active and violated boundary and linear constraints. During the optimization process, the introduction of rounding errors can force PROC NLP to increase the value of r by a factor of 10, 100, ... If this happens it is indicated by a message written to the log. For more information, see the section "Linear Complementarity (LICOMP)" on page 601.

LCSINGULAR=*r*

LCSING=*r*

LCS=*r*

specifies a criterion $r > 0$ used in the update of the QR decomposition that decides whether an active constraint is linearly dependent on a set of other active constraints. The default value is r=1E−8. The larger r becomes, the more the active constraints are recognized as being linearly dependent. If the value of r is larger than 0.1, it is reset to 0.1.

LINESEARCH=*i*

LIS=*i*

specifies the line-search method for the CONGRA, QUANEW, HYQUAN, and NEWRAP optimization techniques. Refer to Fletcher (1987) for an introduction to line-search techniques. The value of i can be $1, \ldots, 8$. For CONGRA, QUANEW, and NEWRAP, the default value is i=2. A special line-search method is the default for the least squares technique HYQUAN that is based on an algorithm developed by Lindström and Wedin (1984). Although it needs more memory, this default line-search method sometimes works better with large least squares problems. However, by specifying LIS=i, $i = 1, \ldots, 8$, it is possible to use one of the standard techniques with HYQUAN.

LIS=1 specifies a line-search method that needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation.

LIS=2 specifies a line-search method that needs more function than gradient calls for quadratic and cubic interpolation and cubic extrapolation; this method is implemented as shown in Fletcher (1987) and can be modified to an exact line search by using the LSPRECISION= option.

LIS=3 specifies a line-search method that needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation; this method is implemented as

shown in Fletcher (1987) and can be modified to an exact line search by using the LSPRECISION= option.

LIS=4 specifies a line-search method that needs the same number of function and gradient calls for stepwise extrapolation and cubic interpolation.

LIS=5 specifies a line-search method that is a modified version of LIS=4.

LIS=6 specifies golden section line search (Polak 1971), which uses only function values for linear approximation.

LIS=7 specifies bisection line search (Polak 1971), which uses only function values for linear approximation.

LIS=8 specifies the Armijo line-search technique (Polak 1971), which uses only function values for linear approximation.

LIST

displays the model program and variable lists. The LIST option is a debugging feature and is not normally needed. This output is not included in either the default output or the output specified by the PALL option.

LISTCODE

displays the derivative tables and the compiled program code. The LISTCODE option is a debugging feature and is not normally needed. This output is not included in either the default output or the output specified by the PALL option. The option is similar to that used in MODEL procedure in SAS/ETS software.

LSPRECISION=*r*

LSP=*r*

specifies the degree of accuracy that should be obtained by the line-search algorithms LIS=2 and LIS=3. Usually an imprecise line search is inexpensive and sufficient for convergence to the optimum. For difficult optimization problems, a more precise and expensive line search may be necessary (Fletcher 1987). The second (default for NEWRAP, QUANEW, and CONGRA) and third line-search methods approach exact line search for small LSPRECISION= values. In the presence of numerical problems, it is advised to decrease the LSPRECISION= value to obtain a more precise line search. The default values are as follows:

TECH=	UPDATE=	LSP default
QUANEW	DBFGS, BFGS	$r = 0.4$
QUANEW	DDFP, DFP	$r = 0.06$
HYQUAN	DBFGS	$r = 0.1$
HYQUAN	DDFP	$r = 0.06$
CONGRA	all	$r = 0.1$
NEWRAP	no update	$r = 0.9$

For more details, refer to Fletcher (1987).

MAXFUNC=*i*
MAXFU=*i*
specifies the maximum number *i* of function calls in the optimization process. The default values are

- TRUREG, LEVMAR, NRRIDG, NEWRAP: 125
- QUANEW, HYQUAN, DBLDOG: 500
- CONGRA, QUADAS: 1000
- NMSIMP: 3000

Note that the optimization can be terminated only after completing a full iteration. Therefore, the number of function calls that are actually performed can exceed the number that is specified by the MAXFUNC= option.

MAXITER=*i* [*n*]
MAXIT=*i* [*n*]
specifies the maximum number *i* of iterations in the optimization process. The default values are:

- TRUREG, LEVMAR, NRRIDG, NEWRAP: 50
- QUANEW, HYQUAN, DBLDOG: 200
- CONGRA, QUADAS: 400
- NMSIMP: 1000

This default value is valid also when *i* is specified as a missing value. The optional second value *n* is valid only for TECH=QUANEW with nonlinear constraints. It specifies an upper bound *n* for the number of iterations of an algorithm used to reduce the violation of nonlinear constraints at a starting point. The default value is *n*=20.

MAXSTEP=*r* [*n*]
specifies an upper bound for the step length of the line-search algorithms during the first *n* iterations. By default, *r* is the largest double precision value and *n* is the largest integer available. Setting this option can increase the speed of convergence for TECH=CONGRA, TECH=QUANEW, TECH=HYQUAN, and TECH=NEWRAP.

MAXTIME=*r*
specifies an upper limit of *r* seconds of real time for the optimization process. The default value is the largest floating point double representation of the computer. Note that the time specified by the MAXTIME= option is checked only once at the end of each iteration. Therefore, the actual running time of the PROC NLP job may be longer than that specified by the MAXTIME= option. The actual running time includes the rest of the time needed to finish the iteration, time for the output of the (temporary) results, and (if required) the time for saving the results in an OUTEST= data set. Using the MAXTIME= option with a permanent OUTEST= data set enables you to separate large optimization problems into a series of smaller problems that need smaller amounts of real time.

MINITER=*i*
MINIT=*i*
specifies the minimum number of iterations. The default value is zero. If more iterations than are actually needed are requested for convergence to a stationary point, the optimization algorithms can behave strangely. For example, the effect of rounding errors can prevent the algorithm from continuing for the required number of iterations.

MODEL=*model-name, model-list*

MOD=*model-name, model-list*

MODFILE=*model-name, model-list*

reads the program statements from one or more input model files created by previous PROC NLP steps using the OUTMODEL= option. If it is necessary to include the program code at a special location in newly written code, the INCLUDE statement can be used instead of using the MODEL= option. Using both the MODEL= option and the INCLUDE statement with the same model file will include the same model twice, which can produce different results than including it once. The MODEL= option is similar to the option used in PROC MODEL in SAS/ETS software.

MSINGULAR=*r*

MSING=*r*

specifies a relative singularity criterion $r > 0$ for measuring singularity of Hessian and crossproduct Jacobian and their projected forms. The default value is 1E−12 if the SINGULAR= option is not specified and $\max(10 \times \epsilon, 1\mathrm{E}-4 \times SINGULAR)$ otherwise. For more information, see the section "Covariance Matrix" on page 618.

NOEIGNUM

suppresses the computation and output of the determinant and the inertia of the Hessian, crossproduct Jacobian, and covariance matrices. The inertia of a symmetric matrix are the numbers of negative, positive, and zero eigenvalues. For large applications, the NOEIGNUM option can save computer time.

NOMISS

is valid only for those variables of the DATA= data set that are referred to in program statements. If the NOMISS option is specified, observations with any missing value for those variables are skipped. If the NOMISS option is not specified, the missing value may result in a missing value of the objective function, implying that the corresponding BY group of data is not processed.

NOPRINT

NOP

suppresses the output.

OPTCHECK[=*r***]**

computes the function values $f(x_l)$ of a grid of points x_l in a small neighborhood of x^*. The x_l are located in a ball of radius r about x^*. If the OPTCHECK option is specified without r, the default value is $r = 0.1$ at the starting point and $r = 0.01$ at the terminating point. If a point x_l^* is found with a better function value than $f(x^*)$, then optimization is restarted at x_l^*. For more information on grids, see the section "DECVAR Statement" on page 580.

OUT=*SAS-data-set*

creates an output data set that contains those variables of a DATA= input data set referred to in the program statements plus additional variables computed by performing the program statements of the objective function, derivatives, and nonlinear constraints. The OUT= data set can also contain first- and second-order derivatives of these variables if the OUTDER= option is specified. The variables and derivatives are evaluated at x^*; for TECH=NONE, they are evaluated at x^0.

OUTALL

If an OUTEST= data set is specified, this option sets the OUTHESSIAN option if the MIN or MAX statement is used. If the LSQ statement is used, the OUTALL option sets the OUTCRPJAC option. If nonlinear constraints are specified using the NLINCON statement, the OUTALL option sets the OUTNLCJAC option.

OUTCRPJAC

If an OUTEST= data set is specified, the crossproduct Jacobian matrix of the m functions composing the least squares function is written to the OUTEST= data set.

OUTDER= 0 | 1 | 2

specifies whether or not derivatives are written to the OUT= data set. For OUTDER=2, first- and second-order derivatives are written to the data set; for OUTDER=1, only first-order derivatives are written; for OUTDER=0, no derivatives are written to the data set. The default value is OUTDER=0. Derivatives are evaluated at x^*.

OUTEST=*SAS-data-set*

OUTVAR=*SAS-data-set*

creates an output data set that contains the results of the optimization. This is useful for reporting and for restarting the optimization in a subsequent execution of the procedure. Information in the data set can include parameter estimates, gradient values, constraint information, Lagrangian values, Hessian values, Jacobian values, covariance, standard errors, and confidence intervals.

OUTGRID

writes the grid points and their function values to the OUTEST= data set. By default, only the feasible grid points are saved; however, if the INFEASIBLE option is specified, all feasible and infeasible grid points are saved. Note that the BEST= option does not affect the output of grid points to the OUTEST= data set. For more information on grids, see the section "DECVAR Statement" on page 580.

OUTHESSIAN

OUTHES

writes the Hessian matrix of the objective function to the OUTEST= data set. If the Hessian matrix is computed for some other reason (if, for example, the PHESSIAN option is specified), the OUTHESSIAN option is set by default.

OUTITER

writes during each iteration the parameter estimates, the value of the objective function, the gradient (if available), and (if OUTTIME is specified) the time in seconds from the start of the optimization to the OUTEST= data set.

OUTJAC

writes the Jacobian matrix of the m functions composing the least squares function to the OUTEST= data set. If the PJACOBI option is specified, the OUTJAC option is set by default.

OUTMODEL=*model-name*

OUTMOD=*model-name*

OUTM=*model-name*

specifies the name of an output model file to which the program statements are to be written. The program statements of this file can be included into the program statements of a succeeding PROC NLP run using the MODEL= option or the INCLUDE program statement. The OUTMODEL= option is

similar to the option used in PROC MODEL in SAS/ETS software. Note that the following statements are not part of the program code that is written to an OUTMODEL= data set: MIN, MAX, LSQ, MINQUAD, MAXQUAD, DECVAR, BOUNDS, BY, CRPJAC, GRADIENT, HESSIAN, JACNLC, JACOBIAN, LABEL, LINCON, MATRIX, and NLINCON.

OUTNLCJAC

If an OUTEST= data set is specified, the Jacobian matrix of the nonlinear constraint functions specified by the NLINCON statement is written to the OUTEST= data set. If the Jacobian matrix of the nonlinear constraint functions is computed for some other reason (if, for example, the PNLCJAC option is specified), the OUTNLCJAC option is set by default.

OUTTIME

is used if an OUTEST= data set is specified and if the OUTITER option is specified. If OUTTIME is specified, the time in seconds from the start of the optimization to the start of each iteration is written to the OUTEST= data set.

PALL

ALL

displays all optional output except the output generated by the PSTDERR, PCOV, LIST, or LISTCODE option.

PCOV

displays the covariance matrix specified by the COV= option. The PCOV option is set automatically if the PALL and COV= options are set.

PCRPJAC

PJTJ

displays the $n \times n$ crossproduct Jacobian matrix $J^T J$. If the PALL option is specified and the LSQ statement is used, this option is set automatically. If general linear constraints are active at the solution, the projected crossproduct Jacobian matrix is also displayed.

PEIGVAL

displays the distribution of eigenvalues if a G4 inverse is computed for the covariance matrix. The PEIGVAL option is useful for observing which eigenvalues of the matrix are recognized as zero eigenvalues when the generalized inverse is computed, and it is the basis for setting the COVSING= option in a subsequent execution of PROC NLP. For more information, see the section "Covariance Matrix" on page 618.

PERROR

specifies additional output for such applications where the program code for objective function or nonlinear constraints cannot be evaluated during the iteration process. The PERROR option is set by default during the evaluations at the starting point but not during the optimization process.

PFUNCTION

displays the values of all functions specified in a LSQ, MIN, or MAX statement for each observation read from the DATA= input data set. The PALL option sets the PFUNCTION option automatically.

PGRID

displays the function values from the grid search. For more information on grids, see the section "DECVAR Statement" on page 580.

PHESSIAN

PHES

displays the $n \times n$ Hessian matrix G. If the PALL option is specified and the MIN or MAX statement is used, this option is set automatically. If general linear constraints are active at the solution, the projected Hessian matrix is also displayed.

PHISTORY

PHIS

displays the optimization history. No optimization history is displayed for TECH=LICOMP. This output is included in both the default output and the output specified by the PALL option.

PINIT

PIN

displays the initial values and derivatives (if available). This output is included in both the default output and the output specified by the PALL option.

PJACOBI

PJAC

displays the $m \times n$ Jacobian matrix J. Because of the memory requirement for large least squares problems, this option is not invoked when using the PALL option.

PNLCJAC

displays the Jacobian matrix of nonlinear constraints specified by the NLINCON statement. The PNLCJAC option is set automatically if the PALL option is specified.

PSHORT

SHORT

PSH

restricts the amount of default output. If PSHORT is specified, then

- The initial values are not displayed.
- The listing of constraints is not displayed.
- If there is more than one function in the MIN, MAX, or LSQ statement, their values are not displayed.
- If the GRADCHECK option is used, only the test vector is displayed.

PSTDERR

STDERR

SE

computes standard errors that are defined as square roots of the diagonal elements of the covariance matrix. The t values and probabilities $> |t|$ are displayed together with the approximate standard errors. The type of covariance matrix must be specified using the COV= option. The SIGSQ= option, the VARDEF= option, and the special variables _NOBS_ and _DF_ defined in the program statements can be used to define a scalar factor σ^2 of the covariance matrix and the approximate standard errors. For more information, see the section "Covariance Matrix" on page 618.

PSUMMARY

SUMMARY

SUM

restricts the amount of default displayed output to a short form of iteration history and notes, warnings, and errors.

PTIME

specifies the output of four different but partially overlapping differences of real time:

- total running time
- total time for the evaluation of objective function, nonlinear constraints, and derivatives: shows the total time spent executing the programming statements specifying the objective function, derivatives, and nonlinear constraints, and (if necessary) their first- and second-order derivatives. This is the total time needed for code evaluation before, during, and after iterating.
- total time for optimization: shows the total time spent iterating.
- time for some CMP parsing: shows the time needed for parsing the program statements and its derivatives. In most applications this is a negligible number, but for applications that contain ARRAY statements or DO loops or use an optimization technique with analytic second-order derivatives, it can be considerable.

RANDOM=*i*

specifies a positive integer as a seed value for the pseudorandom number generator. Pseudorandom numbers are used as the initial value $x^{(0)}$.

RESTART=*i*

REST=*i*

specifies that the QUANEW, HYQUAN, or CONGRA algorithm is restarted with a steepest descent/ascent search direction after at most $i > 0$ iterations. Default values are as follows:

- CONGRA with UPDATE=PB: restart is done automatically so specification of *i* is not used
- CONGRA with UPDATE≠PB: $i = \min(10n, 80)$, where n is the number of parameters
- QUANEW, HYQUAN: *i* is the largest integer available

SIGSQ=sq

specifies a scalar factor $sq > 0$ for computing the covariance matrix. If the SIGSQ= option is specified, VARDEF=N is the default. For more information, see the section "Covariance Matrix" on page 618.

SINGULAR=*r*

SING=*r*

specifies the singularity criterion $r > 0$ for the inversion of the Hessian matrix and crossproduct Jacobian. The default value is 1E−8. For more information, refer to the MSINGULAR= and VSINGULAR= options.

TECH=*name*

TECHNIQUE=*name*

specifies the optimization technique. Valid values for it are as follows:

- CONGRA
  chooses one of four different conjugate gradient optimization algorithms, which can be more precisely specified with the UPDATE= option and modified with the LINESEARCH= option. When this option is selected, UPDATE=PB by default. For $n \geq 400$, CONGRA is the default optimization technique.
- DBLDOG
  performs a version of double dogleg optimization, which can be more precisely specified with the UPDATE= option. When this option is selected, UPDATE=DBFGS by default.
- HYQUAN
  chooses one of three different hybrid quasi-Newton optimization algorithms which can be more precisely defined with the VERSION= option and modified with the LINESEARCH= option. By default, VERSION=2 and UPDATE=DBFGS.
- LEVMAR
  performs the Levenberg-Marquardt minimization. For $n < 40$, this is the default minimization technique for least squares problems.
- LICOMP
  solves a quadratic program as a linear complementarity problem.
- NMSIMP
  performs the Nelder-Mead simplex optimization method.
- NONE
  does not perform any optimization. This option can be used
  - to do grid search without optimization
  - to compute and display derivatives and covariance matrices which cannot be obtained efficiently with any of the optimization techniques
- NEWRAP
  performs the Newton-Raphson optimization technique. The algorithm combines a line-search algorithm with ridging. The line-search algorithm LINESEARCH=2 is the default.
- NRRIDG
  performs the Newton-Raphson optimization technique. For $n \leq 40$ and non-linear least squares, this is the default.
- QUADAS
  performs a special quadratic version of the active set strategy.
- QUANEW
  chooses one of four quasi-Newton optimization algorithms which can be defined more precisely with the UPDATE= option and modified with the LINESEARCH= option. This is the default for $40 < n < 400$ or if there are nonlinear constraints.
- TRUREG
  performs the trust region optimization technique.

UPDATE=*method*

UPD=*method*

specifies the update method for the (dual) quasi-Newton, double dogleg, hybrid quasi-Newton, or conjugate gradient optimization technique. Not every update method can be used with each optimizer. For more information, see the section "Optimization Algorithms" on page 597. Valid values for *method* are as follows:

BFGS	performs the original BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the inverse Hessian matrix.
DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix.
DDFP	performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.
DFP	performs the original DFP (Davidon, Fletcher, & Powell) update of the inverse Hessian matrix.
PB	performs the automatic restart update method of Powell (1977) and Beale (1972).
FR	performs the Fletcher-Reeves update (Fletcher 1987).
PR	performs the Polak-Ribiere update (Fletcher 1987).
CD	performs a conjugate-descent update of Fletcher (1987).

VARDEF= DF | N

specifies the divisor d used in the calculation of the covariance matrix and approximate standard errors. If the SIGSQ= option is not specified, the default value is VARDEF=DF; otherwise, VARDEF=N is the default. For more information, see the section "Covariance Matrix" on page 618.

VERSION= 1 | 2 | 3

VS= 1 | 2 | 3

specifies the version of the hybrid quasi-Newton optimization technique or the version of the quasi-Newton optimization technique with nonlinear constraints.

For the hybrid quasi-Newton optimization technique,

VS=1 specifies version HY1 of Fletcher and Xu (1987).

VS=2 specifies version HY2 of Fletcher and Xu (1987).

VS=3 specifies version HY3 of Fletcher and Xu (1987).

For the quasi-Newton optimization technique with nonlinear constraints,

VS=1 specifies update of the μ vector like Powell (1978a, b) (update like VF02AD).

VS=2 specifies update of the μ vector like Powell (1982b) (update like VMCWD).

In both cases, the default value is VS=2.

VSINGULAR=*r*

VSING=*r*

specifies a relative singularity criterion $r > 0$ for measuring singularity of Hessian and crossproduct Jacobian and their projected forms, which may have to be converted to compute the covariance matrix. The default value is 1E−8 if the SINGULAR= option is not specified and the value of SINGULAR otherwise. For more information, see the section "Covariance Matrix" on page 618.

XCONV=*r*[*n*]

XTOL=*r*[*n*]

specifies the relative parameter convergence criterion. For all techniques except NMSIMP, termination requires a small relative parameter change in subsequent iterations:

$$\frac{\max_j |x_j^{(k)} - x_j^{(k-1)}|}{\max(|x_j^{(k)}|, |x_j^{(k-1)}|, \text{XSIZE})} \leq r$$

For the NMSIMP technique, the same formula is used, but $x_j^{(k)}$ is defined as the vertex with the lowest function value and $x_j^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is *r*=1E−8 for the NMSIMP technique and *r*=0 otherwise. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

XSIZE=*r*

specifies the parameter $r > 0$ of the relative parameter termination criterion. The default value is $r = 0$. For more details, see the XCONV= option.

ARRAY Statement

ARRAY *arrayname [dimensions] [$] [variables and constants] ;* **;**

The ARRAY statement is similar to, but not the same as, the ARRAY statement in the SAS DATA step. The ARRAY statement is used to associate a name (of no more than eight characters) with a list of variables and constants. The array name is used with subscripts in the program to refer to the array elements. The following code illustrates this:

```
array r[8] r1-r8;

do i = 1 to 8;
   r[i] = 0;
   end;
```

The ARRAY statement does not support all the features of the DATA step ARRAY statement. It cannot be used to give initial values to array elements. Implicit indexing of variables cannot be used; all array references must have explicit subscript expressions. Only exact array dimensions are allowed; lower-bound specifications are not supported and a maximum of six dimensions is allowed.

On the other hand, the ARRAY statement does allow both variables and constants to be used as array elements. (Constant array elements cannot have values assigned to them.) Both dimension specification and the list of

elements are optional, but at least one must be given. When the list of elements is not given or fewer elements than the size of the array are listed, array variables are created by suffixing element numbers to the array name to complete the element list.

BOUNDS Statement

BOUNDS *b_con [, b_con. . .]* **;**

where *b_con* is given in one of the following formats:

- number *operator* parameter_list *operator* number
- number *operator* parameter_list
- parameter_list *operator* number

and *operator* is $\leq$, $<$, $\geq$, $>$, or =.

Boundary constraints are specified with a BOUNDS statement. One- or two-sided boundary constraints are allowed. The list of boundary constraints are separated by commas. For example,

```
bounds 0 <= a1-a9 X <= 1, -1 <= c2-c5;
bounds b1-b10 y >= 0;
```

More than one BOUNDS statement can be used. If more than one lower (upper) bound for the same parameter is specified, the maximum (minimum) of these is taken. If the maximum l_j of all lower bounds is larger than the minimum of all upper bounds u_j for the same variable x_j, the boundary constraint is replaced by $x_j = l_j = \min(u_j)$ defined by the minimum of all upper bounds specified for x_j.

BY Statement

BY *variables* **;**

A BY statement can be used with PROC NLP to obtain separate analyses on DATA= data set observations in groups defined by the BY variables. That means, for values of the TECH= option other than NONE, an optimization problem is solved for each BY group separately. When a BY statement appears, the procedure expects the input DATA= data set to be sorted in order of the BY variables. If the input data set is not sorted in ascending order, it is necessary to use one of the following alternatives:

- Use the SORT procedure with a similar BY statement to sort the data.
- Use the BY statement option NOTSORTED or DESCENDING in the BY statement for the NLP procedure. As a cautionary note, the NOTSORTED option does not mean that the data are unsorted but rather that the data are arranged in groups (according to values of the BY variables) and that these groups are not necessarily in alphabetical or increasing numeric order.

- Use the DATASETS procedure (in Base SAS software) to create an index on the BY variables.

For more information on the BY statement, refer to the discussion in *SAS Language Reference: Concepts*. For more information on the DATASETS procedure, refer to the *SAS Procedures Guide*.

CRPJAC Statement

CRPJAC *variables* **;**

The CRPJAC statement defines the crossproduct Jacobian matrix $J^T J$ used in solving least squares problems. For more information, see the section "Derivatives" on page 596. If the DIAHES option is not specified, the CRPJAC statement lists $n(n+1)/2$ variable names, which correspond to the elements $(J^T J)_{j,k},\ j \geq k$ of the lower triangle of the symmetric crossproduct Jacobian matrix listed by rows. For example, the statements

```
lsq f1-f3;
decvar x1-x3;
crpjac jj1-jj6;
```

correspond to the crossproduct Jacobian matrix

$$J^T J = \begin{bmatrix} JJ1 & JJ2 & JJ4 \\ JJ2 & JJ3 & JJ5 \\ JJ4 & JJ5 & JJ6 \end{bmatrix}$$

If the DIAHES option is specified, only the n diagonal elements must be listed in the CRPJAC statement. The n rows and n columns of the crossproduct Jacobian matrix must be in the same order as the n corresponding parameter names listed in the DECVAR statement. To specify the values of nonzero derivatives, the variables specified in the CRPJAC statement have to be defined at the left-hand side of algebraic expressions in programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=levmar;
   lsq f1 f2;
   decvar x1 x2;
   gradient g1 g2;
   crpjac cpj1-cpj3;
   f1   = 10 * (x2 - x1 * x1);
   f2   = 1 - x1;
   g1   = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
   g2   =  100 * (x2 - x1 * x1);
   cpj1 = 400 * x1 * x1 + 1 ;
   cpj2 = -200 * x1;
   cpj3 = 100;
run;
```

DECVAR Statement

DECVAR *name_list [=numbers] [, name_list [=numbers] ...]* **;**

VAR *name_list [=numbers] [, name_list [=numbers] ...]* **;**

PARMS *name_list [=numbers] [, name_list [=numbers] ...]* **;**

PARAMETERS *name_list [=numbers] [, name_list [=numbers] ...]* **;**

The DECVAR statement lists the names of the $n > 0$ decision variables and specifies grid search and initial values for an iterative optimization process. The decision variables listed in the DECVAR statement cannot also be used in the MIN, MAX, MINQUAD, MAXQUAD, LSQ, GRADIENT, HESSIAN, JACOBIAN, CRPJAC, or NLINCON statement.

The DECVAR statement contains a list of decision variable names (not separated by commas) optionally followed by an equals sign and a list of numbers. If the number list consists of only one number, this number defines the initial value for all the decision variables listed to the left of the equals sign.

If the number list consists of more than one number, these numbers specify the grid locations for each of the decision variables listed left of the equals sign. The TO and BY keywords can be used to specify a number list for a grid search. When a grid of points is specified with a DECVAR statement, PROC NLP computes the objective function value at each grid point and chooses the best (feasible) grid point as a starting point for the optimization process. The use of the BEST= option is recommended to save computing time and memory for the storing and sorting of all grid point information. Usually only feasible grid points are included in the grid search. If the specified grid contains points located outside the feasible region and you are interested in the function values at those points, it is possible to use the INFEASIBLE option to compute (and display) their function values as well.

GRADIENT Statement

GRADIENT *variables* **;**

The GRADIENT statement defines the gradient vector which contains the first-order derivatives of the objective function f with respect to $x_1, \ldots, x_n$. For more information, see the section "Derivatives" on page 596. To specify the values of nonzero derivatives, the variables specified in the GRADIENT statement must be defined on the left-hand side of algebraic expressions in programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=congra;
   min y;
   decvar x1 x2;
   gradient g1 g2;
   y1 = 10 * (x2 - x1 * x1);
   y2 = 1 - x1;
   y  = .5 * (y1 * y1 + y2 * y2);
   g1 = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
   g2 = 100 * (x2 - x1 * x1);
run;
```

HESSIAN Statement

HESSIAN *variables* **;**

The HESSIAN statement defines the Hessian matrix G containing the second-order derivatives of the objective function f with respect to $x_1, \ldots, x_n$. For more information, see the section "Derivatives" on page 596.

If the DIAHES option is not specified, the HESSIAN statement lists $n(n+1)/2$ variable names which correspond to the elements $G_{j,k},\ j \geq k$, of the lower triangle of the symmetric Hessian matrix listed by rows. For example, the statements

```
min f;
decvar x1 - x3;
hessian g1-g6;
```

correspond to the Hessian matrix

$$G = \begin{bmatrix} G1 & G2 & G4 \\ G2 & G3 & G5 \\ G4 & G5 & G6 \end{bmatrix} = \begin{bmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 & \partial^2 f/\partial x_1 \partial x_3 \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 & \partial^2 f/\partial x_2 \partial x_3 \\ \partial^2 f/\partial x_3 \partial x_1 & \partial^2 f/\partial x_3 \partial x_2 & \partial^2 f/\partial x_3^2 \end{bmatrix}$$

If the DIAHES option is specified, only the n diagonal elements must be listed in the HESSIAN statement. The n rows and n columns of the Hessian matrix G must correspond to the order of the n parameter names listed in the DECVAR statement. To specify the values of nonzero derivatives, the variables specified in the HESSIAN statement must be defined on the left-hand side of algebraic expressions in the programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=nrridg;
   min f;
   decvar x1 x2;
   gradient g1 g2;
   hessian h1-h3;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
   f = .5 * (f1 * f1 + f2 * f2);
   g1 = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
   g2 = 100 * (x2 - x1 * x1);
   h1 = -200 * (x2 - 3 * x1 * x1) + 1;
   h2 = -200 * x1;
   h3 = 100;
run;
```

INCLUDE Statement

INCLUDE *model files* **;**

The INCLUDE statement can be used to append model code to the current model code. The contents of included model files, created using the OUTMODEL= option, are inserted into the model program at the position in which the INCLUDE statement appears.

JACNLC Statement

JACNLC *variables* **;**

The JACNLC statement defines the Jacobian matrix for the system of constraint functions $c_1(x), \ldots, c_{mc}(x)$. The statements list the $mc \times n$ variable names which correspond to the elements $CJ_{i,j}, i = 1, \ldots, mc;\ j = 1, \ldots, n$, of the Jacobian matrix by rows.

For example, the statements

```
nlincon c1-c3;
decvar  x1-x2;
jacnlc  cj1-cj6;
```

correspond to the Jacobian matrix

$$CJ = \begin{bmatrix} CJ1 & CJ2 \\ CJ3 & CJ4 \\ CJ5 & CJ6 \end{bmatrix} = \begin{bmatrix} \partial c_1/\partial x_1 & \partial c_1/\partial x_2 \\ \partial c_2/\partial x_1 & \partial c_2/\partial x_2 \\ \partial c_3/\partial x_1 & \partial c_3/\partial x_2 \end{bmatrix}$$

The mc rows of the Jacobian matrix must be in the same order as the mc corresponding names of nonlinear constraints listed in the NLINCON statement. The n columns of the Jacobian matrix must be in the same order as the n corresponding parameter names listed in the DECVAR statement. To specify the values of nonzero derivatives, the variables specified in the JACNLC statement must be defined on the left-hand side of algebraic expressions in programming statements.

For example,

```
array cd[3,4] cd1-cd12;
nlincon c1-c3 >= 0;
jacnlc cd1-cd12;

c1 = 8 - x1 * x1 - x2 * x2 - x3 * x3 - x4 * x4 -
       x1 + x2 - x3 + x4;
c2 = 10 - x1 * x1 - 2 * x2 * x2 - x3 * x3 - 2 * x4 * x4 +
       x1 + x4;
c3 = 5 - 2 * x1 * x2 - x2 * x2 - x3 * x3 - 2 * x1 + x2 + x4;

cd[1,1]= -1 - 2 * x1;   cd[1,2]= 1 - 2 * x2;
cd[1,3]= -1 - 2 * x3;   cd[1,4]= 1 - 2 * x4;
cd[2,1]=  1 - 2 * x1;   cd[2,2]= -4 * x2;
cd[2,3]= -2 * x3;       cd[2,4]= 1 - 4 * x4;
cd[3,1]= -2 - 4 * x1;   cd[3,2]= 1 - 2 * x2;
cd[3,3]= -2 * x3;       cd[3,4]= 1;
```

JACOBIAN Statement

JACOBIAN *variables* **;**

The JACOBIAN statement defines the JACOBIAN matrix J for a system of objective functions. For more information, see the section "Derivatives" on page 596.

The JACOBIAN statement lists $m \times n$ variable names that correspond to the elements $J_{i,j}, i = 1, \ldots, m;\ j = 1, \ldots, n$, of the Jacobian matrix listed by rows.

For example, the statements

```
lsq f1-f3;
decvar x1 x2;
jacobian j1-j6;
```

correspond to the Jacobian matrix

$$J = \begin{bmatrix} J1 & J2 \\ J3 & J4 \\ J5 & J6 \end{bmatrix} = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 \end{bmatrix}$$

The m rows of the Jacobian matrix must correspond to the order of the m function names listed in the MIN, MAX, or LSQ statement. The n columns of the Jacobian matrix must correspond to the order of the n decision variables listed in the DECVAR statement. To specify the values of nonzero derivatives, the variables specified in the JACOBIAN statement must be defined on the left-hand side of algebraic expressions in programming statements.

For example, consider the Rosenbrock function:

```
proc nlp tech=levmar;
   array j[2,2] j1-j4;
   lsq f1 f2;
   decvar x1 x2;
   jacobian j1-j4;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
   j[1,1] = -20 * x1;
   j[1,2] = 10;
   j[2,1] = -1;
   j[2,2] = 0;   /* is not needed */
run;
```

The JACOBIAN statement is useful only if more than one objective function is given in the MIN, MAX, or LSQ statement, or if a DATA= input data set specifies more than one function. If the MIN, MAX, or LSQ statement contains only one objective function and no DATA= input data set is used, the JACOBIAN and GRADIENT statements are equivalent. In the case of least squares minimization, the crossproduct Jacobian is used as an approximate Hessian matrix.

LABEL Statement

LABEL *variable='label' [,variable='label'. . .]* **;**

The LABEL statement can be used to assign labels (up to 40 characters) to the decision variables listed in the DECVAR statement. The INEST= data set can also be used to assign labels. The labels are attached to the output and are used in an OUTEST= data set.

LINCON Statement

LINCON *l_con [, l_con . . .]* **;**

where *l_con* is given in one of the following formats:

- linear_term *operator* number
- number *operator* linear_term

and linear_term is of the following form:

$$< +|- >< \text{number}* > \text{variable} < +|- < \text{number}* > \text{variable} \ldots >$$

The value of *operator* can be one of the following: $\leq$, $<$, $\geq$, $>$, or =.

The LINCON statement specifies equality or inequality constraints

$$\sum_{j=1}^{n} a_{ij} x_j \; \{\leq \mid = \mid \geq\} \; b_i \quad \text{for } i = 1, \ldots, m$$

separated by commas. For example, the constraint $4x_1 - 3x_2 = 0$ is expressed as

```
decvar x1 x2;
lincon 4 * x1 - 3 * x2 = 0;
```

and the constraints

$$10x_1 - x_2 \geq 10$$

$$x_1 + 5x_2 \geq 15$$

are expressed as

```
decvar x1 x2;
lincon 10 <= 10 * x1 - x2,
       x1 + 5 * x2 >= 15;
```

MATRIX Statement

MATRIX *M_name pattern_definitions* **;**

The MATRIX statement defines a matrix H and the vector g, which can be given in the MINQUAD or MAXQUAD statement. The matrix H and vector g are initialized to zero, so that only the nonzero elements are given. The five different forms of the MATRIX statement are illustrated with the following example:

$$H = \begin{bmatrix} 100 & 10 & 1 & 0 \\ 10 & 100 & 10 & 1 \\ 1 & 10 & 100 & 10 \\ 0 & 1 & 10 & 100 \end{bmatrix} \quad g = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad c = 0$$

Each MATRIX statement first names the matrix or vector and then lists its elements. If more than one MATRIX statement is given for the same matrix, the later definitions override the earlier ones.

The rows and columns in matrix *H* and vector *g* correspond to the order of decision variables in the DECVAR statement.

- **Full Matrix Definition:** The MATRIX statement consists of *H_name* or *g_name* followed by an equals sign and all (nonredundant) numerical values of the matrix *H* or vector *g*. Assuming symmetry, only the elements of the lower triangular part of the matrix *H* must be listed. This specification should be used mainly for small problems with almost dense *H* matrices.

```
MATRIX H=  100
           10 100
            1  10 100
            0   1  10 100;
MATRIX G= 1  2  3  4;
```

- **Band-diagonal Matrix Definition:** This form of *pattern definition* is useful if the *H* matrix has (almost) constant band-diagonal structure. The MATRIX statement consists of *H_name* followed by empty brackets [,], an equals sign, and a list of numbers to be assigned to the diagonal and successive subdiagonals.

```
MATRIX H[,]=  100 10 1;
MATRIX G= 1  2  3  4;
```

- **Sparse Matrix Definitions:** In each of the following three specification types, the *H_name* or *g_name* is followed by a list of *pattern definitions* separated by commas. Each *pattern definition* consists of a location specification in brackets on the left side of an equals sign that is followed by a list of numbers.

  - **(Sub)Diagonalwise:** This form of *pattern definition* is useful if the *H* matrix contains nonzero elements along diagonals or subdiagonals. The starting location is specified by an index pair in brackets $[i, j]$. The expression $k * num$ on the right-hand side specifies that num is assigned to the elements $[i, j], \ldots, [i + k - 1, j + k - 1]$ in a diagonal direction of the *H* matrix. The special case $k = 1$ can be used to assign values to single nonzero element locations in *H*.

```
MATRIX H [1,1]= 4 * 100,
         [2,1]= 3 *  10,
         [3,1]= 2 *   1;
MATRIX G [1,1]= 1  2  3  4;
```

  - **Columnwise Starting in Diagonal:** This form of *pattern definition* is useful if the *H* matrix contains nonzero elements columnwise starting in the diagonal. The starting location is specified by only one index *j* in brackets $[, j]$. The *k* numbers at the right-hand side are assigned to the elements $[j, j], \ldots, [\min(j + k - 1, n), j]$.

```
MATRIX H [,1]= 100  10   1,
         [,2]= 100  10   1,
         [,3]= 100  10,
         [,4]= 100;
MATRIX G [,1]= 1  2  3  4;
```

– **Rowwise Starting in First Column:** This form of *pattern definition* is useful if the H matrix contains nonzero elements rowwise ending in the diagonal. The starting location is specified by only one index i in brackets $[i,]$. The k numbers at the right-hand side are assigned to the elements $[i,1],\ldots,[i,\min(k,i)]$.

```
MATRIX H [1,]= 100,
         [2,]=  10 100,
         [3,]=   1  10 100,
         [4,]=   0   1  10 100;
MATRIX G [1,]= 1  2  3  4;
```

MIN, MAX, and LSQ Statements

MIN *variables* **;**

MAX *variables* **;**

LSQ *variables* **;**

The MIN, MAX, or LSQ statement specifies the objective functions. Only one of the three statements can be used at a time and at least one must be given. The MIN and LSQ statements are for minimizing the objective function, and the MAX statement is for maximizing the objective function. The MIN, MAX, or LSQ statement lists one or more variables naming the objective functions $f_i, i = 1,\ldots,m$ (later defined by SAS program code).

- If the MIN or MAX statement lists m function names $f_1,\ldots,f_m$, the objective function f is

$$f(x) = \sum_{i=1}^{m} f_i$$

- If the LSQ statement lists m function names $f_1,\ldots,f_m$, the objective function f is

$$f(x) = \frac{1}{2}\sum_{i=1}^{m} f_i^2(x)$$

Note that the LSQ statement can be used only if TECH=LEVMAR or TECH=HYQUAN.

MINQUAD and MAXQUAD Statements

MINQUAD *H_name [, g_name [, c_number]]* **;**

MAXQUAD *H_name [, g_name [, c_number]]* **;**

The MINQUAD and MAXQUAD statements specify the matrix H, vector g, and scalar c that define a quadratic objective function. The MINQUAD statement is for minimizing the objective function and the MAXQUAD statement is for maximizing the objective function.

The rows and columns in H and g correspond to the order of decision variables given in the DECVAR statement. Specifying the objective function with a MINQUAD or MAXQUAD statement indirectly defines the analytic derivatives for the objective function. Therefore, statements specifying derivatives are not valid in these cases. Also, only use these statements when TECH=LICOMP or TECH=QUADAS and no nonlinear constraints are imposed.

There are three ways of using the MINQUAD or MAXQUAD statement:

- **Using ARRAY Statements**:
  The names *H_name* and *g_name* specified in the MINQUAD or MAXQUAD statement can be used in ARRAY statements. This specification is mainly for small problems with almost dense H matrices.

```
proc nlp pall;
   array h[2,2] .4 0
                0 4;
   minquad h, -100;
   decvar x1 x2 = -1;
   bounds  2 <= x1 <= 50,
         -50 <= x2 <= 50;
   lincon 10 <= 10 * x1 - x2;
run;
```

- **Using Elementwise Setting**:
  The names *H_name* and *g_name* specified in the MINQUAD or MAXQUAD statement can be followed directly by one-dimensional indices specifying the corresponding elements of the matrix H and vector g. These element names can be used on the left side of numerical assignments. The one-dimensional index value l following *H_name*, which corresponds to the element H_{ij}, is computed by $l = (i-1)n + j, i \geq j$. The matrix H and vector g are initialized to zero, so that only the nonzero elements must be given. This specification is efficient for small problems with sparse H matrices.

```
proc nlp pall;
   minquad h, -100;
   decvar x1 x2;
   bounds  2 <= x1 <= 50,
         -50 <= x2 <= 50;
   lincon 10 <= 10 * x1 - x2;
   h1 = .4; h4 = 4;
run;
```

- **Using MATRIX Statements**:
  The names *H_name* and *g_name* specified in the MINQUAD or MAXQUAD statement can be used in MATRIX statements. There are different ways to specify the nonzero elements of the matrix H and vector g by MATRIX statements. The following example illustrates one way to use the MATRIX statement.

```
proc nlp all;
   matrix h[1,1] = .4 4;
   minquad h, -100;
   decvar x1 x2 = -1;
```

```
   bounds 2 <= x1 <= 50,
        -50 <= x2 <= 50;
   lincon 10 <= 10 * x1 - x2;
run;
```

NLINCON Statement

NLINCON *nlcon [, nlcon ...] [/ option]* **;**

NLC *nlcon [, nlcon ...] [/ option]* **;**

where *nlcon* is given in one of the following formats:

- number *operator* variable_list *operator* number
- number *operator* variable_list
- variable_list *operator* number

and *operator* is $\leq$, $<$, $\geq$, $>$, or =. The value of *option* can be SUMOBS or EVERYOBS.

General nonlinear equality and inequality constraints are specified with the NLINCON statement. The syntax of the NLINCON statement is similar to that of the BOUNDS statement with two small additions:

- The BOUNDS statement can contain only the names of decision variables. The NLINCON statement can also contain the names of continuous functions of the decision variables. These functions must be computed in the program statements, and since they can depend on the values of some of the variables in the DATA= data set, there are two possibilities:
  - If the continuous functions should be summed across all observations read from the DATA= data set, the NLINCON statement must be terminated by the / SUMOBS option.
  - If the continuous functions should be evaluated separately for each observation in the data set, the NLINCON statement must be terminated by the / EVERYOBS option. One constraint is generated for each observation in the data set.
- If the continuous function should be evaluated only once for the entire data set, the NLINCON statement has the same form as the BOUNDS statement. If this constraint does depend on the values of variables in the DATA= data set, it is evaluated using the data of the first observation.

One- or two-sided constraints can be specified in the NLINCON statement. However, equality constraints must be one-sided. The pairs of operators (<,<=) and (>,>=) are treated in the same way.

These three statements require the values of the three functions v_1, v_2, v_3 to be between zero and ten, and they are equivalent:

```
nlincon 0 <= v1-v3,
        v1-v3 <= 10;

nlincon 0 <= v1-v3 <= 10;

nlincon 10 >= v1-v3 >= 0;
```

Also, consider the Rosen-Suzuki problem. It has three nonlinear inequality constraints:

$$\begin{aligned} 8 - x_1^2 - x_2^2 - x_3^2 - x_4^2 - x_1 + x_2 - x_3 + x_4 &\geq 0 \\ 10 - x_1^2 - 2x_2^2 - x_3^2 - 2x_4^2 + x_1 + x_4 &\geq 0 \\ 5 - 2x_1^2 - x_2^2 - x_3^2 - 2x_1 + x_2 + x_4 &\geq 0 \end{aligned}$$

These are specified as

```
nlincon c1-c3 >= 0;

c1 = 8 - x1 * x1 - x2 * x2 - x3 * x3 - x4 * x4 -
         x1 + x2 - x3 + x4;
c2 = 10 - x1 * x1 - 2 * x2 * x2 - x3 * x3 - 2 * x4 * x4 +
         x1 + x4;
c3 = 5 - 2 * x1 * x1 - x2 * x2 - x3 * x3 - 2 * x1 + x2 + x4;
```

NOTE: QUANEW and NMSIMP are the only optimization subroutines that support the NLINCON statement.

PROFILE Statement

PROFILE *parms [/ [ALPHA= values] [options]]* **;**

where *parms* is given in the format *pnam_1 pnam_2 ... pnam_n*, and *values* is the list of α values in (0,1).

The PROFILE statement

- writes the (x, y) coordinates of profile points for each of the listed parameters to the OUTEST= data set
- displays, or writes to the OUTEST= data set, the profile likelihood confidence limits (PL CLs) for the listed parameters for the specified α values. If the approximate standard errors are available, the corresponding Wald confidence limits can be computed.

When computing the profile points or likelihood profile confidence intervals, PROC NLP assumes that a maximization of the log likelihood function is desired. Each point of the profile and each endpoint of the confidence interval is computed by solving corresponding nonlinear optimization problems.

The keyword PROFILE must be followed by the names of parameters for which the profile or the PL CLs should be computed. If the parameter name list is empty, the profiles and PL CLs for all parameters are computed. Then, optionally, the α values follow. The list of α values may contain TO and BY keywords. Each element must satisfy $0 < \alpha < 1$. The following is an example:

```
profile l11-l15 u1-u5 c /
        alpha= .9 to .1 by -.1 .09 to .01 by -.01;
```

Duplicate α values or values outside $(0, 1)$ are automatically eliminated from the list.

A number of additional options can be specified.

FFACTOR=*r* specifies the factor relating the discrepancy function $f(\theta)$ to the χ^2 quantile. The default value is $r = 2$.

FORCHI= F | CHI defines the scale for the y values written to the OUTEST= data set. For FORCHI=F, the y values are scaled to the values of the log likelihood function $f = f(\theta)$; for FORCHI=CHI, the y values are scaled so that $\hat{y} = \chi^2$. The default value is FORCHI=F.

FEASRATIO=*r* specifies a factor of the Wald confidence limit (or an approximation of it if standard errors are not computed) defining an upper bound for the search for confidence limits. In general, the range of x values in the profile graph is between $r = 1$ and $r = 2$ times the length of the corresponding Wald interval. For many examples, the χ^2 quantiles corresponding to small α values define a y level $\hat{y} - \frac{1}{2}q_1(1-\alpha)$, which is too far away from $\hat{y}$ to be reached by $y(x)$ for x within the range of twice the Wald confidence limit. The search for an intersection with such a y level at a practically infinite value of x can be computationally expensive. A smaller value for r can speed up computation time by restricting the search for confidence limits to a region closer to $\hat{x}$. The default value of $r = 1000$ practically disables the FEASRATIO= option.

OUTTABLE specifies that the complete set θ of parameter estimates rather than only $x = \theta_j$ for each confidence limit is written to the OUTEST= data set. This output can be helpful for further analyses on how small changes in $x = \theta_j$ affect the changes in the $\theta_i, i \neq j$.

For some applications, it may be computationally less expensive to compute the PL confidence limits for a few parameters than to compute the approximate covariance matrix of many parameters, which is the basis for the Wald confidence limits. However, the computation of the profile of the discrepancy function and the corresponding CLs in general will be much more time-consuming than that of the Wald CLs.

Program Statements

This section lists the program statements used to code the objective function and nonlinear constraints and their derivatives, and it documents the differences between program statements in the NLP procedure and program statements in the DATA step. The syntax of program statements used in PROC NLP is identical to that used in the CALIS, GENMOD, and MODEL procedures (refer to the *SAS/ETS User's Guide*).

Most of the program statements which can be used in the SAS DATA step can also be used in the NLP procedure. See the *SAS Language Guide* or base SAS documentation for a description of the SAS program

```
ABORT;
CALL name [ ( expression [, expression . . . ] ) ];
DELETE;
DO [ variable = expression
                [ TO expression ] [ BY expression ]
                [, expression [ TO expression ] [ BY expression ] . . . ]
    ]
    [ WHILE expression ] [ UNTIL expression ];
END;
GOTO statement_label;
IF expression;
IF expression THEN program_statement;
    ELSE program_statement;
variable = expression;
variable + expression;
LINK statement_label;
PUT [ variable] [=] [...];
RETURN;
SELECT [( expression )];
STOP;
SUBSTR( variable, index, length ) = expression;
WHEN (expression) program_statement;
    OTHERWISE program_statement;
```

statements. For the most part, the SAS program statements work as they do in the SAS DATA step as documented in the *SAS Language Guide*. However, there are several differences that should be noted.

- The ABORT statement does not allow any arguments.
- The DO statement does not allow a character index variable. Thus

  ```
  do i = 1,2,3;
  ```

  is supported; however,

  ```
  do i = 'A','B','C';
  ```

  is not.
- The PUT statement, used mostly for program debugging in PROC NLP, supports only some of the features of the DATA step PUT statement, and has some new features that the DATA step PUT statement does not:
  - The PROC NLP PUT statement does not support line pointers, factored lists, iteration factors, overprinting, _INFILE_, the colon (:) format modifier, or "$".
  - The PROC NLP PUT statement does support expressions, but the expression must be enclosed inside of parentheses. For example, the following statement displays the square root of x: `put (sqrt(x));`
  - The PROC NLP PUT statement supports the print item _PDV_ to print a formatted listing of all variables in the program. For example, the following statement displays a more readable listing of the variables than the _all_ print item: `put _pdv_;`

- The WHEN and OTHERWISE statements allow more than one target statement. That is, DO/END groups are not necessary for multiple statement WHENs. For example, the following syntax is valid:

```
SELECT;
WHEN ( exp1 )   stmt1;
                stmt2;
WHEN ( exp2 )   stmt3;
                stmt4;
END;
```

It is recommended to keep some kind of order in the input of NLP, that is, between the statements that define decision variables and constraints and the program code used to specify objective functions and derivatives.

Use of Special Variables in Program Code

Except for the quadratic programming techniques (QUADAS and LICOMP) that do not execute program statements during the iteration process, several special variables in the program code can be used to communicate with PROC NLP in special situations:

- **_OBS_** If a DATA= input data set is used, it is possible to access a variable _OBS_ which contains the number of the observation processed from the data set. You should not change the content of the _OBS_ variable. This variable enables you to modify the programming statements depending on the observation number processed in the DATA= input data set. For example, to set variable A to 1 when observation 10 is processed, and otherwise to 2, it is possible to specify

```
IF _OBS_ = 10 THEN A=1; ELSE A=2;
```

- **_ITER_** This variable is set by PROC NLP, and it contains the number of the current iteration of the optimization technique as it is displayed in the optimization history. You should not change the content of the _ITER_ variable. It is possible to read the value of this variable in order to modify the programming statements depending on the iteration number processed. For example, to display the content of the variables A, B, and C when there are more than 100 iterations processed, it is possible to use

```
IF _ITER_ > 100 THEN PUT A B C;
```

- **_DPROC_** This variable is set by PROC NLP to indicate whether the code is called only to obtain the values of the m objective functions f_i (_DPROC_=0) or whether specified derivatives (defined by the GRADIENT, JACOBIAN, CRPJAC, or HESSIAN statement) also have to be computed (_DPROC_=1). You should not change the content of the _DPROC_ variable. Checking the _DPROC_ variable makes it possible to save computer time by not performing derivative code that is not needed by the current call. In particular, when a DATA= input data set is used, the code is processed many times to compute only the function values. If the programming statements in the program contain the specification of computationally expensive first- and second-order derivatives, you can put the derivative code in an IF statement that is processed only if _DPROC_ is not zero.

- **_INDF_** The _INDF_ variable is set by PROC NLP to inform you of the source of calls to the function or derivative programming.

INDF=0 indicates the first function call in a grid search. This is also the first call evaluating the programming statements if there is a grid search defined by grid values in the DECVAR statement.

INDF=1 indicates further function calls in a grid search.

INDF=2 indicates the call for the feasible starting point. This is also the first call evaluating the programming statements if there is no grid search defined.

INDF=3 indicates calls from a gradient-checking algorithm.

INDF=4 indicates calls from the minimization algorithm. The _ITER_ variable contains the iteration number.

INDF=5 If the active set algorithm leaves the feasible region (due to rounding errors), an algorithm tries to return it into the feasible region; _INDF_=5 indicates a call that is done when such a step is successful.

INDF=6 indicates calls from a factorial test subroutine that tests the neighborhood of a point x for optimality.

INDF=7, 8 indicates calls from subroutines needed to compute finite-difference derivatives using only values of the objective function. No nonlinear constraints are evaluated.

INDF=9 indicates calls from subroutines needed to compute second-order finite-difference derivatives using analytic (specified) first-order derivatives. No nonlinear constraints are evaluated.

INDF=10 indicates calls where only the nonlinear constraints but no objective function are needed. The analytic derivatives of the nonlinear constraints are computed.

INDF=11 indicates calls where only the nonlinear constraints but no objective function are needed. The analytic derivatives of the nonlinear constraints are not computed.

INDF=-1 indicates the last call at the final solution.

You should not change the content of the _INDF_ variable.

- **_LIST_** You can set the _LIST_ variable to control the output during the iteration process:

  LIST=0 is equivalent to the NOPRINT option. It suppresses all output.

  LIST=1 is equivalent to the PSUMMARY but not the PHISTORY option. The optimization start and termination messages are displayed. However, the PSUMMARY option suppresses the output of the iteration history.

  LIST=2 is equivalent to the PSHORT option or to a combination of the PSUMMARY and PHISTORY options. The optimization start information, the iteration history, and termination message are displayed.

  LIST=3 is equivalent to not PSUMMARY, not PSHORT, and not PALL. The optimization start information, the iteration history, and the termination message are displayed.

  LIST=4 is equivalent to the PALL option. The extended optimization start information (also containing settings of termination criteria and other control parameters) is displayed.

  LIST=5 In addition to the iteration history, the vector $x^{(k)}$ of parameter estimates is displayed for each iteration k.

  LIST=6 In addition to the iteration history, the vector $x^{(k)}$ of parameter estimates and the gradient $g^{(k)}$ (if available) of the objective function are displayed for each iteration k.

It is possible to set the _LIST_ variable in the program code to obtain more or less output in each iteration of the optimization process. For example,

```
IF _ITER_ = 11      THEN _LIST_=5;
ELSE IF _ITER_ > 11 THEN _LIST_=1;
                    ELSE _LIST_=3;
```

- **_TOOBIG_** The value of _TOOBIG_ is initialized to 0 by PROC NLP, but you can set it to 1 during the iteration, indicating problems evaluating the program statements. The objective function and derivatives must be computable at the starting point. However, during the iteration it is possible to set the _TOOBIG_ variable to 1, indicating that the programming statements (computing the value of the objective function or the specified derivatives) cannot be performed for the current value of x_k. Some of the optimization techniques check the value of _TOOBIG_ and try to modify the parameter estimates so that the objective function (or derivatives) can be computed in a following trial.
- **_NOBS_** The value of the _NOBS_ variable is initialized by PROC NLP to the product of the number of functions *mfun* specified in the MIN, MAX or LSQ statement and the number of valid observations *nobs* in the current BY group of the DATA= input data set. The value of the _NOBS_ variable is used for computing the scalar factor of the covariance matrix (see the COV=, VARDEF=, and SIGSQ= options). If you reset the value of the _NOBS_ variable, the value that is available at the end of the iteration is used by PROC NLP to compute the scalar factor of the covariance matrix.
- **_DF_** The value of the _DF_ variable is initialized by PROC NLP to the number n of parameters specified in the DECVAR statement. The value of the _DF_ variable is used for computing the scalar factor d of the covariance matrix (see the COV=, VARDEF=, and SIGSQ= options). If you reset the value of the _DF_ variable, the value that is available at the end of the iteration is used by PROC NLP to compute the scalar factor of the covariance matrix.
- **_LASTF_** In each iteration (except the first one), the value of the _LASTF_ variable is set by PROC NLP to the final value of the objective function that was achieved during the last iteration. This value should agree with the value that is displayed in the iteration history and that is written in the OUTEST= data set when the OUTITER option is specified.

Details: NLP Procedure

Criteria for Optimality

PROC NLP solves

$$\begin{array}{ll} \min_{x \in \mathcal{R}^n} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i = 1, \ldots, m_e \\ & c_i(x) \geq 0, \quad i = m_e + 1, \ldots, m \end{array}$$

where f is the objective function and the c_i's are the constraint functions.

A point x is feasible if it satisfies all the constraints. The feasible region $\mathcal{G}$ is the set of all the feasible points. A feasible point x^* is a global solution of the preceding problem if no point in $\mathcal{G}$ has a smaller function value than $f(x^*)$. A feasible point x^* is a local solution of the problem if there exists some open neighborhood surrounding x^* in that no point has a smaller function value than $f(x^*)$. Nonlinear programming algorithms cannot consistently find global minima. All the algorithms in PROC NLP find a local minimum for this problem. If you need to check whether the obtained solution is a global minimum, you may have to run PROC NLP with different starting points obtained either at random or by selecting a point on a grid that contains $\mathcal{G}$.

Every local minimizer x^* of this problem satisfies the following local optimality conditions:

- The gradient (vector of first derivatives) $g(x^*) = \nabla f(x^*)$ of the objective function f (projected toward the feasible region if the problem is constrained) at the point x^* is zero.
- The Hessian (matrix of second derivatives) $G(x^*) = \nabla^2 f(x^*)$ of the objective function f (projected toward the feasible region $\mathcal{G}$ in the constrained case) at the point x^* is positive definite.

Most of the optimization algorithms in PROC NLP use iterative techniques that result in a sequence of points $x^0, \ldots, x^n, \ldots$, that converges to a local solution x^*. At the solution, PROC NLP performs tests to confirm that the (projected) gradient is close to zero and that the (projected) Hessian matrix is positive definite.

Karush-Kuhn-Tucker Conditions

An important tool in the analysis and design of algorithms in constrained optimization is the *Lagrangian function*, a linear combination of the objective function and the constraints:

$$L(x, \lambda) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x)$$

The coefficients λ_i are called *Lagrange multipliers*. This tool makes it possible to state necessary and sufficient conditions for a local minimum. The various algorithms in PROC NLP create sequences of points, each of which is closer than the previous one to satisfying these conditions.

Assuming that the functions f and c_i are twice continuously differentiable, the point x^* is a *local minimum* of the nonlinear programming problem, if there exists a vector $\lambda^* = (\lambda_1^*, \ldots, \lambda_m^*)$ that meets the following conditions.

1. First-order Karush-Kuhn-Tucker conditions:

$$\begin{array}{lll} c_i(x^*) = 0, & & i = 1, \ldots, m_e \\ c_i(x^*) \geq 0, & \lambda_i^* \geq 0, \quad \lambda_i^* c_i(x^*) = 0, & i = m_e + 1, \ldots, m \\ \nabla_x L(x^*, \lambda^*) = 0 & & \end{array}$$

2. Second-order conditions: Each nonzero vector $y \in \mathcal{R}^n$ that satisfies

$$y^T \nabla_x c_i(x^*) = 0 \left\{ \begin{array}{l} i = 1, \ldots, m_e \\ \forall i \in \{m_e + 1, \ldots, m : \lambda_i^* > 0\} \end{array} \right.$$

also satisfies

$$y^T \nabla_x^2 L(x^*, \lambda^*) y > 0$$

Most of the algorithms to solve this problem attempt to find a combination of vectors x and λ for which the gradient of the Lagrangian function with respect to x is zero.

Derivatives

The first- and second-order conditions of optimality are based on first and second derivatives of the objective function f and the constraints c_i.

The gradient vector contains the first derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$g(x) = \nabla f(x) = \left(\frac{\partial f}{\partial x_j} \right)$$

The $n \times n$ symmetric Hessian matrix contains the second derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$G(x) = \nabla^2 f(x) = \left(\frac{\partial^2 f}{\partial x_j \partial x_k} \right)$$

For least squares problems, the $m \times n$ Jacobian matrix contains the first-order derivatives of the m objective functions $f_i(x)$ with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$J(x) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

In the case of least squares problems, the crossproduct Jacobian

$$J^T J = \left(\sum_{i=1}^{m} \frac{\partial f_i}{\partial x_j} \frac{\partial f_i}{\partial x_k} \right)$$

is used as an approximate Hessian matrix. It is a very good approximation of the Hessian if the residuals at the solution are "small." (If the residuals are not sufficiently small at the solution, this approach may result in slow convergence.) The fact that it is possible to obtain Hessian approximations for this problem that do not require any computation of second derivatives means that least squares algorithms are more efficient than unconstrained optimization algorithms. Using the vector $f(x) = (f_1(x), \ldots, f_m(x))^T$ of function values, PROC NLP computes the gradient $g(x)$ by

$$g(x) = J^T(x) f(x)$$

The $mc \times n$ Jacobian matrix contains the first-order derivatives of the mc nonlinear constraint functions $c_i(x)$, $i = 1, \ldots, mc$, with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$CJ(x) = (\nabla c_1, \ldots, \nabla c_{mc}) = \left(\frac{\partial c_i}{\partial x_j} \right)$$

PROC NLP provides three ways to compute derivatives:

- It computes analytical first- and second-order derivatives of the objective function f with respect to the n variables x_j.
- It computes first- and second-order finite-difference approximations to the derivatives. For more information, see the section "Finite-Difference Approximations of Derivatives" on page 607.
- The user supplies formulas for analytical or numerical first- and second-order derivatives of the objective function in the GRADIENT, JACOBIAN, CRPJAC, and HESSIAN statements. The JACNLC statement can be used to specify the derivatives for the nonlinear constraints.

Optimization Algorithms

There are three groups of optimization techniques available in PROC NLP. A particular optimizer can be selected with the TECH= option in the PROC NLP statement.

Table 7.2 Karush-Kuhn-Tucker Conditions

Algorithm	TECH=
Linear Complementarity Problem	LICOMP
Quadratic Active Set Technique	QUADAS
Trust-Region Method	TRUREG
Newton-Raphson Method with Line Search	NEWRAP
Newton-Raphson Method with Ridging	NRRIDG
Quasi-Newton Methods (DBFGS, DDFP, BFGS, DFP)	QUANEW
Double Dogleg Method (DBFGS, DDFP)	DBLDOG
Conjugate Gradient Methods (PB, FR, PR, CD)	CONGRA
Nelder-Mead Simplex Method	NMSIMP
Levenberg-Marquardt Method	LEVMAR
Hybrid Quasi-Newton Methods (DBFGS, DDFP)	HYQUAN

Since no single optimization technique is invariably superior to others, PROC NLP provides a variety of optimization techniques that work well in various circumstances. However, it is possible to devise problems for which none of the techniques in PROC NLP can find the correct solution. Moreover, nonlinear optimization can be computationally expensive in terms of time and memory, so care must be taken when matching an algorithm to a problem.

All optimization techniques in PROC NLP use $O(n^2)$ memory except the conjugate gradient methods, which use only $O(n)$ memory and are designed to optimize problems with many variables. Since the techniques are iterative, they require the repeated computation of

- the function value (optimization criterion)
- the gradient vector (first-order partial derivatives)
- for some techniques, the (approximate) Hessian matrix (second-order partial derivatives)
- values of linear and nonlinear constraints
- the first-order partial derivatives (Jacobian) of nonlinear constraints

However, since each of the optimizers requires different derivatives and supports different types of constraints, some computational efficiencies can be gained. The following table shows, for each optimization technique, which derivatives are needed (FOD: first-order derivatives; SOD: second-order derivatives) and what kinds of constraints (BC: boundary constraints; LIC: linear constraints; NLC: nonlinear constraints) are supported.

Algorithm	FOD	SOD	BC	LIC	NLC
LICOMP	-	-	x	x	-
QUADAS	-	-	x	x	-
TRUREG	x	x	x	x	-
NEWRAP	x	x	x	x	-
NRRIDG	x	x	x	x	-
QUANEW	x	-	x	x	x
DBLDOG	x	-	x	x	-
CONGRA	x	-	x	x	-
NMSIMP	-	-	x	x	x
LEVMAR	x	-	x	x	-
HYQUAN	x	-	x	x	-

Preparation for Using Optimization Algorithms

It is rare that a problem is submitted to an optimization algorithm "as is." By making a few changes in your problem, you can reduce its complexity, which would increase the chance of convergence and save execution time.

- Whenever possible, use linear functions instead of nonlinear functions. PROC NLP will reward you with faster and more accurate solutions.
- Most optimization algorithms are based on quadratic approximations to nonlinear functions. You should try to avoid the use of functions that cannot be properly approximated by quadratic functions. Try to avoid the use of rational functions.

  For example, the constraint

  $$\frac{\sin(x)}{x+1} > 0$$

  should be replaced by the equivalent constraint

  $$\sin(x)(x+1) > 0$$

  and the constraint

  $$\frac{\sin(x)}{x+1} = 1$$

  should be replaced by the equivalent constraint

  $$\sin(x) - (x+1) = 0$$

- Try to avoid the use of exponential functions, if possible.
- If you can reduce the complexity of your function by the addition of a small number of variables, it may help the algorithm avoid stationary points.
- Provide the best starting point you can. A good starting point leads to better quadratic approximations and faster convergence.

Choosing an Optimization Algorithm

The factors that go into choosing a particular optimizer for a particular problem are complex and may involve trial and error. Several things must be taken into account. First, the structure of the problem has to be considered: Is it quadratic? least squares? Does it have linear or nonlinear constraints? Next, it is important to consider the type of derivatives of the objective function and the constraints that are needed and whether these are analytically tractable or not. This section provides some guidelines for making the right choices.

For many optimization problems, computing the gradient takes more computer time than computing the function value, and computing the Hessian sometimes takes *much* more computer time and memory than computing the gradient, especially when there are many decision variables. Optimization techniques that do not use the Hessian usually require more iterations than techniques that do use Hessian approximations (such as finite differences or BFGS update) and so are often slower. Techniques that do not use Hessians at all tend to be slow and less reliable.

The derivative compiler is not efficient in the computation of second-order derivatives. For large problems, memory and computer time can be saved by programming your own derivatives using the GRADIENT, JACOBIAN, CRPJAC, HESSIAN, and JACNLC statements. If you are not able to specify first- and second-order derivatives of the objective function, you can rely on finite-difference gradients and Hessian update formulas. This combination is frequently used and works very well for small and medium problems. For large problems, you are advised not to use an optimization technique that requires the computation of second derivatives.

The following provides some guidance for matching an algorithm to a particular problem.

- Quadratic Programming
  - **QUADAS**
  - **LICOMP**
- General Nonlinear Optimization
  - Nonlinear Constraints
    - **Small Problems: NMSIMP**
      Not suitable for highly nonlinear problems or for problems with $n > 20$.
    - **Medium Problems: QUANEW**
  - Only Linear Constraints
    - **Small Problems: TRUREG (NEWRAP, NRRIDG)**
      ($n \leq 40$) where the Hessian matrix is not expensive to compute. Sometimes NRRIDG can be faster than TRUREG, but TRUREG can be more stable. NRRIDG needs only one matrix with $n(n+1)/2$ double words; TRUREG and NEWRAP need two such matrices.
    - **Medium Problems: QUANEW (DBLDOG)**
      ($n \leq 200$) where the objective function and the gradient are much faster to evaluate than the Hessian. QUANEW and DBLDOG in general need more iterations than TRUREG, NRRIDG, and NEWRAP, but each iteration can be much faster. QUANEW and DBLDOG need only the gradient to update an approximate Hessian. QUANEW and DBLDOG need slightly less memory than TRUREG or NEWRAP (essentially one matrix with $n(n+1)/2$ double words).

    - **Large Problems: CONGRA**
      ($n > 200$) where the objective function and the gradient can be computed much faster than the Hessian and where too much memory is needed to store the (approximate) Hessian. CONGRA in general needs more iterations than QUANEW or DBLDOG, but each iteration can be much faster. Since CONGRA needs only a factor of n double-word memory, many large applications of PROC NLP can be solved only by CONGRA.
    - **No Derivatives: NMSIMP**
      ($n \leq 20$) where derivatives are not continuous or are very difficult to compute.

- Least Squares Minimization

  - **Small Problems: LEVMAR (HYQUAN)**
    ($n \leq 60$) where the crossproduct Jacobian matrix is inexpensive to compute. In general, LEVMAR is more reliable, but there are problems with high residuals where HYQUAN can be faster than LEVMAR.
  - **Medium Problems: QUANEW (DBLDOG)**
    ($n \leq 200$) where the objective function and the gradient are much faster to evaluate than the crossproduct Jacobian. QUANEW and DBLDOG in general need more iterations than LEVMAR or HYQUAN, but each iteration can be much faster.
  - **Large Problems: CONGRA**
  - **No Derivatives: NMSIMP**

Quadratic Programming Method

The QUADAS and LICOMP algorithms can be used to minimize or maximize a quadratic objective function,

$$f(x) = \frac{1}{2}x^T Gx + g^T x + c, \quad \text{with} \quad G^T = G$$

subject to linear or boundary constraints

$$Ax \geq b \quad \text{or} \quad l_j \leq x_j \leq u_j$$

where $x = (x_1, \ldots, x_n)^T$, $g = (g_1, \ldots, g_n)^T$, G is an $n \times n$ symmetric matrix, A is an $m \times n$ matrix of general linear constraints, and $b = (b_1, \ldots, b_m)^T$. The value of c modifies only the value of the objective function, not its derivatives, and the location of the optimizer x^* does not depend on the value of the constant term c. For QUADAS or LICOMP, the objective function must be specified using the MINQUAD or MAXQUAD statement or using an INQUAD= data set.

In this case, derivatives do not need to be specified because the gradient vector

$$\nabla f(x) = Gx + g$$

and the $n \times n$ Hessian matrix

$$\nabla^2 f(x) = G$$

are easily obtained from the data input.

Simple boundary and general linear constraints can be specified using the BOUNDS or LINCON statement or an INQUAD= or INEST= data set.

General Quadratic Programming (QUADAS)

The QUADAS algorithm is an active set method that iteratively updates the QT decomposition of the matrix A_k of active linear constraints and the Cholesky factor of the projected Hessian $Z_k^T G Z_k$ simultaneously. The update of active boundary and linear constraints is done separately; refer to Gill et al. (1984). Here Q is an $n_{free} \times n_{free}$ orthogonal matrix composed of vectors spanning the null space Z of A_k in its first $n_{free} - n_{alc}$ columns and range space Y in its last n_{alc} columns; T is an $n_{alc} \times n_{alc}$ triangular matrix of special form, $t_{ij} = 0$ for $i < n - j$, where n_{free} is the number of free parameters (n minus the number of active boundary constraints), and n_{alc} is the number of active linear constraints. The Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ and the QT decomposition are updated simultaneously when the active set changes.

Linear Complementarity (LICOMP)

The LICOMP technique solves a quadratic problem as a linear complementarity problem. It can be used only if G is positive (negative) semidefinite for minimization (maximization) and if the parameters are restricted to be positive.

This technique finds a point that meets the Karush-Kuhn-Tucker conditions by solving the linear complementary problem

$$w = Mz + q$$

with constraints

$$w^T z \geq 0, \quad w \geq 0, \quad z \geq 0,$$

where

$$z = \begin{bmatrix} x \\ \lambda \end{bmatrix} \quad M = \begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix} \quad q = \begin{bmatrix} g \\ -b \end{bmatrix}$$

Only the LCEPSILON= option can be used to specify a tolerance used in computations.

General Nonlinear Optimization

Trust-Region Optimization (TRUREG)

The trust region method uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function $f(x)$ have continuous first- and second-order derivatives inside the feasible region.

The trust region method iteratively optimizes a quadratic approximation to the nonlinear objective function within a hyperelliptic trust region with radius Δ that constrains the step length corresponding to the quality of the quadratic approximation. The trust region method is implemented using Dennis, Gay, and Welsch (1981), Gay (1983).

The trust region method performs well for small to medium problems and does not require many function, gradient, and Hessian calls. If the computation of the Hessian matrix is computationally expensive, use the UPDATE= option for update formulas (that gradually build the second-order information in the Hessian). For larger problems, the conjugate gradient algorithm may be more appropriate.

Newton-Raphson Optimization With Line-Search (NEWRAP)

The NEWRAP technique uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function have continuous first- and second-order derivatives inside the feasible region. If second-order derivatives are computed efficiently and precisely, the NEWRAP method may perform well for medium to large problems, and it does not need many function, gradient, and Hessian calls.

This algorithm uses a pure Newton step when the Hessian is positive definite and when the Newton step reduces the value of the objective function successfully. Otherwise, a combination of ridging and line search is done to compute successful steps. If the Hessian is not positive definite, a multiple of the identity matrix is added to the Hessian matrix to make it positive definite (Eskow and Schnabel 1991).

In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation (LIS=2).

Newton-Raphson Ridge Optimization (NRRIDG)

The NRRIDG technique uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function have continuous first- and second-order derivatives inside the feasible region.

This algorithm uses a pure Newton step when the Hessian is positive definite and when the Newton step reduces the value of the objective function successfully. If at least one of these two conditions is not satisfied, a multiple of the identity matrix is added to the Hessian matrix. If this algorithm is used for least squares problems, it performs a ridged Gauss-Newton minimization.

The NRRIDG method performs well for small to medium problems and does not need many function, gradient, and Hessian calls. However, if the computation of the Hessian matrix is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

Since NRRIDG uses an orthogonal decomposition of the approximate Hessian, each iteration of NRRIDG can be slower than that of NEWRAP, which works with Cholesky decomposition. However, usually NRRIDG needs fewer iterations than NEWRAP.

Quasi-Newton Optimization (QUANEW)

The (dual) quasi-Newton method uses the gradient $g(x^{(k)})$ and does not need to compute second-order derivatives since they are approximated. It works well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian, but in general it requires more iterations than the techniques TRUREG, NEWRAP, and NRRIDG, which compute second-order derivatives.

The QUANEW algorithm depends on whether or not there are nonlinear constraints.

Unconstrained or Linearly Constrained Problems If there are no nonlinear constraints, QUANEW is either

- the original quasi-Newton algorithm that updates an approximation of the inverse Hessian, or
- the dual quasi-Newton algorithm that updates the Cholesky factor of an approximate Hessian (default),

depending on the value of the UPDATE= option. For problems with general linear inequality constraints, the dual quasi-Newton methods can be more efficient than the original ones.

Four update formulas can be specified with the UPDATE= option:

DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.
DDFP	performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.

BFGS performs the original BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the inverse Hessian matrix.

DFP performs the original DFP (Davidon, Fletcher, & Powell) update of the inverse Hessian matrix.

In each iteration, a line search is done along the search direction to find an approximate optimum. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step length α satisfying the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit of the step length. Violating the left-side Goldstein condition can affect the positive definiteness of the quasi-Newton update. In those cases, either the update is skipped or the iterations are restarted with an identity matrix resulting in the steepest descent or ascent search direction. Line-search algorithms other than the default one can be specified with the LINESEARCH= option.

Nonlinearly Constrained Problems The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's (1978a, 1982b) *Variable Metric Constrained WatchDog* (VMCWD) algorithm. A similar but older algorithm (VF02AD) is part of the Harwell library. Both VMCWD and VF02AD use Fletcher's VE02AD algorithm (part of the Harwell library) for positive-definite quadratic programming. The PROC NLP QUANEW implementation uses a quadratic programming subroutine that updates and downdates the approximation of the Cholesky factor when the active set changes. The nonlinear QUANEW algorithm is not a feasible-point algorithm, and the value of the objective function need not decrease (minimization) or increase (maximization) monotonically. Instead, the algorithm tries to reduce a linear combination of the objective function and constraint violations, called the *merit function*.

The following are similarities and differences between this algorithm and the VMCWD algorithm:

- A modification of this algorithm can be performed by specifying VERSION=1, which replaces the update of the Lagrange vector μ with the original update of Powell (1978a, b) that is used in VF02AD. This can be helpful for some applications with linearly dependent active constraints.
- If the VERSION option is not specified or if VERSION=2 is specified, the evaluation of the Lagrange vector μ is performed in the same way as Powell (1982b) describes.
- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS (or DFP) update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, a restart is done with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.
- The Cholesky factor is loaded into the quadratic programming subroutine, automatically ensuring positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ and the QT decomposition are updated simultaneously when the active set changes. Refer to Gill et al. (1984) for more information.
- The line-search strategy is very similar to that of Powell (1982b). However, this algorithm does not call for derivatives during the line search, so the algorithm generally needs fewer derivative calls than function calls. VMCWD always requires the same number of derivative and function calls. Sometimes Powell's line-search method uses steps that are too long. In these cases, use the INSTEP= option to restrict the step length α.

- The watchdog strategy is similar to that of Powell (1982b); however, it does not return automatically after a fixed number of iterations to a former better point. A return here is further delayed if the observed function reduction is close to the expected function reduction of the quadratic model.
- The Powell termination criterion still is used (as FCONV2) but the QUANEW implementation uses two additional termination criteria (GCONV and ABSGCONV).

The nonlinear QUANEW algorithm needs the Jacobian matrix of the first-order derivatives (constraints normals) of the constraints $CJ(x)$.

You can specify two update formulas with the UPDATE= option:

DBFGS performs the dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default.

DDFP performs the dual DFP update of the Cholesky factor of the Hessian matrix.

This algorithm uses its own line-search technique. No options or parameters (except the INSTEP= option) controlling the line search in the other algorithms apply here. In several applications, large steps in the first iterations were troublesome. You can use the INSTEP= option to impose an upper bound for the step length α during the first five iterations. You may also use the INHESSIAN= option to specify a different starting approximation for the Hessian. Choosing simply the INHESSIAN option will use the Cholesky factor of a (possibly ridged) finite-difference approximation of the Hessian to initialize the quasi-Newton update process. The values of the LCSINGULAR=, LCEPSILON=, and LCDEACT= options, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the nonlinear constraints QUANEW algorithm.

Double Dogleg Optimization (DBLDOG)

The double dogleg optimization method combines the ideas of the quasi-Newton and trust region methods. The double dogleg algorithm computes in each iteration the step $s^{(k)}$ as a linear combination of the steepest descent or ascent search direction $s_1^{(k)}$ and a quasi-Newton search direction $s_2^{(k)}$:

$$s^{(k)} = \alpha_1 s_1^{(k)} + \alpha_2 s_2^{(k)}$$

The step is requested to remain within a prespecified trust region radius; refer to Fletcher (1987, p. 107). Thus, the DBLDOG subroutine uses the dual quasi-Newton update but does not perform a line search. Two update formulas can be specified with the UPDATE= option:

DBFGS performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.

DDFP performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.

The double dogleg optimization technique works well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian. The implementation is based on Dennis and Mei (1979) and Gay (1983) but is extended for dealing with boundary and linear constraints. DBLDOG generally needs more iterations than the techniques TRUREG, NEWRAP, or NRRIDG that need second-order derivatives, but each of the DBLDOG iterations is computationally cheap. Furthermore, DBLDOG needs only gradient calls for the update of the Cholesky factor of an approximate Hessian.

Conjugate Gradient Optimization (CONGRA)

Second-order derivatives are not used by CONGRA. The CONGRA algorithm can be expensive in function and gradient calls but needs only $O(n)$ memory for unconstrained optimization. In general, many iterations are needed to obtain a precise solution, but each of the CONGRA iterations is computationally cheap. Four different update formulas for generating the conjugate directions can be specified using the UPDATE= option:

PB	performs the automatic restart update method of Powell (1977) and Beale (1972). This is the default.
FR	performs the Fletcher-Reeves update (Fletcher 1987).
PR	performs the Polak-Ribiere update (Fletcher 1987).
CD	performs a conjugate-descent update of Fletcher (1987).

The default value is UPDATE=PB, since it behaved best in most test examples. You are advised to avoid the option UPDATE=CD, a it behaved worst in most test examples.

The CONGRA subroutine should be used for optimization problems with large n. For the unconstrained or boundary constrained case, CONGRA needs only $O(n)$ bytes of working memory, whereas all other optimization methods require order $O(n^2)$ bytes of working memory. During n successive iterations, uninterrupted by restarts or changes in the working set, the conjugate gradient algorithm computes a cycle of n conjugate search directions. In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step length α satisfying the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit for the step length. Other line-search algorithms can be specified with the LINESEARCH= option.

Nelder-Mead Simplex Optimization (NMSIMP)

The Nelder-Mead simplex method does not use any derivatives and does not assume that the objective function has continuous derivatives. The objective function itself needs to be continuous. This technique requires a large number of function evaluations. It is unlikely to give accurate results for $n \gg 40$.

Depending on the kind of constraints, one of the following Nelder-Mead simplex algorithms is used:

- unconstrained or only boundary constrained problems

  The original Nelder-Mead simplex algorithm is implemented and extended to boundary constraints. This algorithm does not compute the objective for infeasible points. This algorithm is automatically invoked if the LINCON or NLINCON statement is not specified.

- general linearly constrained or nonlinearly constrained problems

  A slightly modified version of Powell's (1992) COBYLA (Constrained Optimization BY Linear Approximations) implementation is used. This algorithm is automatically invoked if either the LINCON or the NLINCON statement is specified.

The original Nelder-Mead algorithm cannot be used for general linear or nonlinear constraints but can be faster for the unconstrained or boundary constrained case. The original Nelder-Mead algorithm changes the shape of the simplex adapting the nonlinearities of the objective function which contributes to an increased speed of convergence. The two NMSIMP subroutines use special sets of termination criteria. For more details, refer to the section "Termination Criteria" on page 610.

Powell's COBYLA Algorithm (COBYLA)

Powell's COBYLA algorithm is a sequential trust region algorithm (originally with a monotonically decreasing radius ρ of a spherical trust region) that tries to maintain a regular-shaped simplex over the iterations. A small modification was made to the original algorithm that permits an increase of the trust region radius ρ in special situations. A sequence of iterations is performed with a constant trust region radius ρ until the computed objective function reduction is much less than the predicted reduction. Then, the trust region radius ρ is reduced. The trust region radius is increased only if the computed function reduction is relatively close to the predicted reduction and the simplex is well-shaped. The start radius ρ_{beg} and the final radius ρ_{end} can be specified using ρ_{beg}=INSTEP and ρ_{end}=ABSXTOL. The convergence to small values of ρ_{end} (high precision) may take many calls of the function and constraint modules and may result in numerical problems. There are two main reasons for the slow convergence of the COBYLA algorithm:

- Only linear approximations of the objective and constraint functions are used locally.
- Maintaining the regular-shaped simplex and not adapting its shape to nonlinearities yields very small simplices for highly nonlinear functions (for example, fourth-order polynomials).

Nonlinear Least Squares Optimization

Levenberg-Marquardt Least Squares Method (LEVMAR)

The Levenberg-Marquardt method is a modification of the trust region method for nonlinear least squares problems and is implemented as in Moré (1978).

This is the recommended algorithm for small to medium least squares problems. Large least squares problems can be transformed into minimization problems, which can be processed with conjugate gradient or (dual) quasi-Newton techniques. In each iteration, LEVMAR solves a quadratically constrained quadratic minimization problem that restricts the step to stay at the surface of or inside an n- dimensional elliptical (or spherical) trust region. In each iteration, LEVMAR uses the crossproduct Jacobian matrix $J^T J$ as an approximate Hessian matrix.

Hybrid Quasi-Newton Least Squares Methods (HYQUAN)

In each iteration of one of the Fletcher and Xu (1987) (refer also to Al-Baali and Fletcher (1985,1986)) hybrid quasi-Newton methods, a criterion is used to decide whether a Gauss-Newton or a dual quasi-Newton search direction is appropriate. The VERSION= option can be used to choose one of three criteria (HY1, HY2, HY3) proposed by Fletcher and Xu (1987). The default is VERSION=2; that is, HY2. In each iteration, HYQUAN computes the crossproduct Jacobian (used for the Gauss-Newton step), updates the Cholesky factor of an approximate Hessian (used for the quasi-Newton step), and does a line search to compute an approximate minimum along the search direction. The default line-search technique used by HYQUAN is especially designed for least squares problems (refer to Lindström and Wedin (1984) and Al-Baali and Fletcher (1986)). Using the LINESEARCH= option you can choose a different line-search algorithm than the default one.

Two update formulas can be specified with the UPDATE= option:

DBFGS performs the dual BFGS (Broyden, Fletcher, Goldfarb, and Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.

DDFP performs the dual DFP (Davidon, Fletcher, and Powell) update of the Cholesky factor of the Hessian matrix.

The HYQUAN subroutine needs about the same amount of working memory as the LEVMAR algorithm. In most applications, LEVMAR seems to be superior to HYQUAN, and using HYQUAN is recommended only when problems are experienced with the performance of LEVMAR.

Finite-Difference Approximations of Derivatives

The FD= and FDHESSIAN= options specify the use of finite-difference approximations of the derivatives. The FD= option specifies that all derivatives are approximated using function evaluations, and the FDHESSIAN= option specifies that second-order derivatives are approximated using gradient evaluations.

Computing derivatives by finite-difference approximations can be very time-consuming, especially for second-order derivatives based only on values of the objective function (FD= option). If analytical derivatives are difficult to obtain (for example, if a function is computed by an iterative process), you might consider one of the optimization techniques that uses first-order derivatives only (TECH=QUANEW, TECH=DBLDOG, or TECH=CONGRA).

Forward-Difference Approximations

The forward-difference derivative approximations consume less computer time but are usually not as precise as those using central-difference formulas.

- First-order derivatives: n additional function calls are needed:

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$$

- Second-order derivatives based on function calls only (Dennis and Schnabel 1983, p. 80, 104): for dense Hessian, $n(n+3)/2$ additional function calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_j}$$

- Second-order derivatives based on gradient calls (Dennis and Schnabel 1983, p. 103): n additional gradient calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x + h_j e_j) - g_i(x)}{2h_j} + \frac{g_j(x + h_i e_i) - g_j(x)}{2h_i}$$

Central-Difference Approximations

- First-order derivatives: $2n$ additional function calls are needed:

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i}$$

- Second-order derivatives based on function calls only (Abramowitz and Stegun 1972, p. 884): for dense Hessian, $2n(n+1)$ additional function calls are needed:

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x+2h_ie_i) + 16f(x+h_ie_i) - 30f(x) + 16f(x-h_ie_i) - f(x-2h_ie_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x+h_ie_i+h_je_j) - f(x+h_ie_i-h_je_j) - f(x-h_ie_i+h_je_j) + f(x-h_ie_i-h_je_j)}{4h_ih_j}$$

- Second-order derivatives based on gradient: $2n$ additional gradient calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x+h_je_j) - g_i(x-h_je_j)}{4h_j} + \frac{g_j(x+h_ie_i) - g_j(x-h_ie_i)}{4h_i}$$

The FDIGITS= and CDIGITS= options can be used for specifying the number of accurate digits in the evaluation of objective function and nonlinear constraints. These specifications are helpful in determining an appropriate interval length h to be used in the finite-difference formulas.

The FDINT= option specifies whether the finite-difference intervals h should be computed using an algorithm of Gill, Murray, Saunders, and Wright (1983) or based only on the information of the FDIGITS= and CDIGITS= options. For FDINT=OBJ, the interval h is based on the behavior of the objective function; for FDINT=CON, the interval h is based on the behavior of the nonlinear constraints functions; and for FDINT=ALL, the interval h is based on the behaviors of both the objective function and the nonlinear constraints functions. Note that the algorithm of Gill, Murray, Saunders, and Wright (1983) to compute the finite-difference intervals h_j can be very expensive in the number of function calls. If the FDINT= option is specified, it is currently performed twice, the first time before the optimization process starts and the second time after the optimization terminates.

If FDINT= is not specified, the step lengths h_j, $j = 1, \ldots, n$, are defined as follows:

- for the forward-difference approximation of first-order derivatives using function calls and second-order derivatives using gradient calls: $h_j = \sqrt[2]{\eta_j}(1 + |x_j|)$,
- for the forward-difference approximation of second-order derivatives that use only function calls and all central-difference formulas: $h_j = \sqrt[3]{\eta_j}(1 + |x_j|)$,

where η is defined using the FDIGITS= option:

- If the number of accurate digits is specified with FDIGITS=r, η is set to 10^{-r}.
- If FDIGITS= is not specified, η is set to the machine precision ϵ.

For FDINT=OBJ and FDINT=ALL, the FDIGITS= specification is used in computing the forward and central finite-difference intervals.

If the problem has nonlinear constraints and the FD= option is specified, the first-order formulas are used to compute finite-difference approximations of the Jacobian matrix $JC(x)$. You can use the CDIGITS= option to specify the number of accurate digits in the constraint evaluations to define the step lengths h_j, $j = 1, \ldots, n$. For FDINT=CON and FDINT=ALL, the CDIGITS= specification is used in computing the forward and central finite-difference intervals.

NOTE: If you are unable to specify analytic derivatives and the finite-difference approximations provided by PROC NLP are not good enough to solve your problem, you may program better finite-difference approximations using the GRADIENT, JACOBIAN, CRPJAC, or HESSIAN statement and the program statements.

Hessian and CRP Jacobian Scaling

The rows and columns of the Hessian and crossproduct Jacobian matrix can be scaled when using the trust region, Newton-Raphson, double dogleg, and Levenberg-Marquardt optimization techniques. Each element $G_{i,j}, i, j = 1, \ldots, n$, is divided by the scaling factor $d_i \times d_j$, where the scaling vector $d = (d_1, \ldots, d_n)$ is iteratively updated in a way specified by the HESCAL=*i* option, as follows:

$i = 0$ No scaling is done (equivalent to $d_i = 1$).

$i \neq 0$ First iteration and each restart iteration:

$$d_i^{(0)} = \sqrt{\max(|G_{i,i}^{(0)}|, \epsilon)}$$

$i = 1$ refer to Moré (1978):

$$d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

$i = 2$ refer to Dennis, Gay, and Welsch (1981):

$$d_i^{(k+1)} = \max\left(0.6d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

$i = 3$ d_i is reset in each iteration:

$$d_i^{(k+1)} = \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}$$

where ϵ is the relative machine precision or, equivalently, the largest double precision value that when added to 1 results in 1.

Testing the Gradient Specification

There are three main ways to check the correctness of derivative specifications:

- Specify the FD= or FDHESSIAN= option in the PROC NLP statement to compute finite-difference approximations of first- and second-order derivatives. In many applications, the finite-difference approximations are computed with high precision and do not differ too much from the derivatives that are computed by specified formulas.
- Specify the GRADCHECK option in the PROC NLP statement to compute and display a test vector and a test matrix of the gradient values at the starting point $x^{(0)}$ by the method of Wolfe (1982). If you do not specify the GRADCHECK option, a fast derivative test identical to the GRADCHECK=FAST specification is done by default.

- If the default analytical derivative compiler is used or if derivatives are specified using the GRADIENT or JACOBIAN statement, the gradient or Jacobian computed at the initial point $x^{(0)}$ is tested by default using finite-difference approximations. In some examples, the relative test can show significant differences between the two forms of derivatives, resulting in a warning message indicating that the specified derivatives could be wrong, even if they are correct. This happens especially in cases where the magnitude of the gradient at the starting point $x^{(0)}$ is small.

The algorithm of Wolfe (1982) is used to check whether the gradient $g(x)$ specified by a GRADIENT statement (or indirectly by a JACOBIAN statement) is appropriate for the objective function $f(x)$ specified by the program statements.

Using function and gradient evaluations in the neighborhood of the starting point $x^{(0)}$, second derivatives are approximated by finite-difference formulas. Forward differences of gradient values are used to approximate the Hessian element G_{jk},

$$G_{jk} \approx H_{jk} = \frac{g_j(x + \delta e_k) - g_j(x)}{\delta}$$

where δ is a small step length and $e_k = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ is the unit vector along the kth coordinate axis. The test vector s, with

$$s_j = H_{jj} - \frac{2}{\delta}\left\{\frac{f(x + \delta e_j) - f(x)}{\delta} - g_j(x)\right\}$$

contains the differences between two sets of finite-difference approximations for the diagonal elements of the Hessian matrix

$$G_{jj} = \partial^2 f(x^{(0)})/\partial x_j^2, \quad j = 1, \ldots, n$$

The test matrix ΔH contains the absolute differences of symmetric elements in the approximate Hessian $|H_{jk} - H_{kj}|$, $j, k = 1, \ldots, n$, generated by forward differences of the gradient elements.

If the specification of the first derivatives is correct, the elements of the test vector and test matrix should be relatively small. The location of large elements in the test matrix points to erroneous coordinates in the gradient specification. For very large optimization problems, this algorithm can be too expensive in terms of computer time and memory.

Termination Criteria

All optimization techniques stop iterating at $x^{(k)}$ if at least one of a set of termination criteria is satisfied. PROC NLP also terminates if the point $x^{(k)}$ is fully constrained by n linearly independent active linear or boundary constraints, and all Lagrange multiplier estimates of active inequality constraints are greater than a small negative tolerance.

Since the Nelder-Mead simplex algorithm does not use derivatives, no termination criterion is available based on the gradient of the objective function. Powell's COBYLA algorithm uses only one more termination criterion. COBYLA is a trust region algorithm that sequentially reduces the radius ρ of a spherical trust region from a start radius ρ_{beg} = INSTEP to the final radius ρ_{end} = ABSXTOL. The default value is ρ_{end} = 1E−4. The convergence to small values of ρ_{end} (high precision) may take many calls of the function and constraint modules and may result in numerical problems.

In some applications, the small default value of the ABSGCONV= criterion is too difficult to satisfy for some of the optimization techniques. This occurs most often when finite-difference approximations of derivatives are used.

The default setting for the GCONV= option sometimes leads to early termination far from the location of the optimum. This is especially true for the special form of this criterion used in the CONGRA optimization.

The QUANEW algorithm for nonlinearly constrained optimization does not monotonically reduce the value of either the objective function or some kind of merit function which combines objective and constraint functions. Furthermore, the algorithm uses the watchdog technique with backtracking (Chamberlain et al. 1982). Therefore, no termination criteria were implemented that are based on the values (x or f) of successive iterations. In addition to the criteria used by all optimization techniques, three more termination criteria are currently available. They are based on satisfying the Karush-Kuhn-Tucker conditions, which require that the gradient of the Lagrange function is zero at the optimal point (x^*, λ^*):

$$\nabla_x L(x^*, \lambda^*) = 0$$

For more information, refer to the section "Criteria for Optimality" on page 594.

Active Set Methods

The parameter vector $x \in \mathcal{R}^n$ may be subject to a set of m linear equality and inequality constraints:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m_e$$
$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = m_e + 1, \ldots, m$$

The coefficients a_{ij} and right-hand sides b_i of the equality and inequality constraints are collected in the $m \times n$ matrix A and the $m-$vector b.

The m linear constraints define a feasible region $\mathcal{G}$ in $\mathcal{R}^n$ that must contain the point x^* that minimizes the problem. If the feasible region $\mathcal{G}$ is empty, no solution to the optimization problem exists.

All optimization techniques in PROC NLP (except those processing nonlinear constraints) are *active set methods*. The iteration starts with a feasible point $x^{(0)}$, which either is provided by the user or can be computed by the Schittkowski and Stoer (1979) algorithm implemented in PROC NLP. The algorithm then moves from one feasible point $x^{(k-1)}$ to a better feasible point $x^{(k)}$ along a feasible search direction $s^{(k)}$:

$$x^{(k)} = x^{(k-1)} + \alpha^{(k)} s^{(k)}, \quad \alpha^{(k)} > 0$$

Theoretically, the path of points $x^{(k)}$ never leaves the feasible region $\mathcal{G}$ of the optimization problem, but it can hit its boundaries. The active set $\mathcal{A}^{(k)}$ of point $x^{(k)}$ is defined as the index set of all linear equality constraints and those inequality constraints that are satisfied at $x^{(k)}$. If no constraint is active for $x^{(k)}$, the point is located in the interior of $\mathcal{G}$, and the active set $\mathcal{A}^{(k)}$ is empty. If the point $x^{(k)}$ in iteration k hits the boundary of inequality constraint i, this constraint i becomes active and is added to $\mathcal{A}^{(k)}$. Each equality or active inequality constraint reduces the dimension (degrees of freedom) of the optimization problem.

In practice, the active constraints can be satisfied only with finite precision. The LCEPSILON=*r* option specifies the range for active and violated linear constraints. If the point $x^{(k)}$ satisfies the condition

$$\left| \sum_{j=1}^{n} a_{ij} x_j^{(k)} - b_i \right| \leq t$$

where $t = r \times (|b_i| + 1)$, the constraint i is recognized as an active constraint. Otherwise, the constraint i is either an inactive inequality or a violated inequality or equality constraint. Due to rounding errors in computing the projected search direction, error can be accumulated so that an iterate $x^{(k)}$ steps out of the feasible region. In those cases, PROC NLP may try to pull the iterate $x^{(k)}$ into the feasible region. However, in some cases the algorithm needs to increase the feasible region by increasing the LCEPSILON=*r* value. If this happens it is indicated by a message displayed in the log output.

If you cannot expect an improvement in the value of the objective function by moving from an active constraint back into the interior of the feasible region, you use this inequality constraint as an equality constraint in the next iteration. That means the active set $\mathcal{A}^{(k+1)}$ still contains the constraint i. Otherwise you release the active inequality constraint and increase the dimension of the optimization problem in the next iteration.

A serious numerical problem can arise when some of the active constraints become (nearly) linearly dependent. Linearly dependent equality constraints are removed before entering the optimization. You can use the LCSINGULAR= option to specify a criterion r used in the update of the QR decomposition that decides whether an active constraint is linearly dependent relative to a set of other active constraints.

If the final parameter set x^* is subjected to n_{act} linear equality or active inequality constraints, the QR decomposition of the $n \times n_{act}$ matrix $\hat{A}^T$ of the linear constraints is computed by $\hat{A}^T = QR$, where Q is an $n \times n$ orthogonal matrix and R is an $n \times n_{act}$ upper triangular matrix. The n columns of matrix Q can be separated into two matrices, $Q = [Y, Z]$, where Y contains the first n_{act} orthogonal columns of Q and Z contains the last $n - n_{act}$ orthogonal columns of Q. The $n \times (n - n_{act})$ column-orthogonal matrix Z is also called the nullspace matrix of the active linear constraints $\hat{A}^T$. The $n - n_{act}$ columns of the $n \times (n - n_{act})$ matrix Z form a basis orthogonal to the rows of the $n_{act} \times n$ matrix $\hat{A}$.

At the end of the iteration process, the PROC NLP can display the *projected gradient*

$$g_Z = Z^T g$$

In the case of boundary constrained optimization, the elements of the projected gradient correspond to the gradient elements of the free parameters. A necessary condition for x^* to be a local minimum of the optimization problem is

$$g_Z(x^*) = Z^T g(x^*) = 0$$

The symmetric $n_{act} \times n_{act}$ matrix

$$G_Z = Z^T G Z$$

is called a *projected Hessian matrix*. A second-order necessary condition for x^* to be a local minimizer requires that the projected Hessian matrix is positive semidefinite. If available, the projected gradient and projected Hessian matrix can be displayed and written in an OUTEST= data set.

Those elements of the n_{act} vector of first-order estimates of *Lagrange multipliers*

$$\lambda = (\hat{A}\hat{A}^T)^{-1} \hat{A} Z Z^T g$$

which correspond to active inequality constraints indicate whether an improvement of the objective function can be obtained by releasing this active constraint. For minimization (maximization), a significant negative (positive) Lagrange multiplier indicates that a possible reduction (increase) of the objective function can be obtained by releasing this active linear constraint. The LCDEACT=*r* option can be used to specify a threshold r for the Lagrange multiplier that decides whether an active inequality constraint remains active or can be deactivated. The Lagrange multipliers are displayed (and written in an OUTEST= data set) only if linear constraints are active at the solution x^*. (In the case of boundary-constrained optimization, the Lagrange multipliers for active lower (upper) constraints are the negative (positive) gradient elements corresponding to the active parameters.)

Feasible Starting Point

Two algorithms are used to obtain a feasible starting point.

- When only boundary constraints are specified:
  - If the parameter x_j, $1 \leq j \leq n$, violates a two-sided boundary constraint (or an equality constraint) $l_j \leq x_j \leq u_j$, the parameter is given a new value inside the feasible interval, as follows:

$$x_j = \begin{cases} l_j, & \text{if } u_j \leq l_j \\ l_j + \frac{1}{2}(u_j - l_j), & \text{if } u_j - l_j < 4 \\ l_j + \frac{1}{10}(u_j - l_j), & \text{if } u_j - l_j \geq 4 \end{cases}$$

  - If the parameter x_j, $1 \leq j \leq n$, violates a one-sided boundary constraint $l_j \leq x_j$ or $x_j \leq u_j$, the parameter is given a new value near the violated boundary, as follows:

$$x_j = \begin{cases} l_j + \max(1, \frac{1}{10}l_j), & \text{if } x_j < l_j \\ u_j - \max(1, \frac{1}{10}u_j), & \text{if } x_j > u_j \end{cases}$$

- When general linear constraints are specified, the algorithm of Schittkowski and Stoer (1979) computes a feasible point, which may be quite far from a user-specified infeasible point.

Line-Search Methods

In each iteration k, the (dual) quasi-Newton, hybrid quasi-Newton, conjugate gradient, and Newton-Raphson minimization techniques use iterative line-search algorithms that try to optimize a linear, quadratic, or cubic approximation of f along a feasible descent search direction $s^{(k)}$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}, \quad \alpha^{(k)} > 0$$

by computing an approximately optimal scalar $\alpha^{(k)}$.

Therefore, a line-search algorithm is an iterative process that optimizes a nonlinear function $f = f(\alpha)$ of one parameter (α) within each iteration k of the optimization technique, which itself tries to optimize a

linear or quadratic approximation of the nonlinear objective function $f = f(x)$ of n parameters x. Since the outside iteration process is based only on the approximation of the objective function, the inside iteration of the line-search algorithm does not have to be perfect. Usually, the choice of α significantly reduces (in a minimization) the objective function. Criteria often used for termination of line-search algorithms are the Goldstein conditions (refer to Fletcher (1987)).

Various line-search algorithms can be selected using the LINESEARCH= option. The line-search method LINESEARCH=2 seems to be superior when function evaluation consumes significantly less computation time than gradient evaluation. Therefore, LINESEARCH=2 is the default value for Newton-Raphson, (dual) quasi-Newton, and conjugate gradient optimizations.

A special default line-search algorithm for TECH=HYQUAN is useful only for least squares problems and cannot be chosen by the LINESEARCH= option. This method uses three columns of the $m \times n$ Jacobian matrix, which for large m can require more memory than using the algorithms designated by LINESEARCH=1 through LINESEARCH=8.

The line-search methods LINESEARCH=2 and LINESEARCH=3 can be modified to exact line search by using the LSPRECISION= option (specifying the σ parameter in Fletcher (1987)). The line-search methods LINESEARCH=1, LINESEARCH=2, and LINESEARCH=3 satisfy the left-hand-side and right-hand-side Goldstein conditions (refer to Fletcher (1987)). When derivatives are available, the line-search methods LINESEARCH=6, LINESEARCH=7, and LINESEARCH=8 try to satisfy the right-hand-side Goldstein condition; if derivatives are not available, these line-search algorithms use only function calls.

Restricting the Step Length

Almost all line-search algorithms use iterative extrapolation techniques which can easily lead them to (feasible) points where the objective function f is no longer defined. (e.g., resulting in indefinite matrices for ML estimation) or difficult to compute (e.g., resulting in floating point overflows). Therefore, PROC NLP provides options restricting the step length α or trust region radius Δ, especially during the first main iterations.

The inner product $g^T s$ of the gradient g and the search direction s is the slope of $f(\alpha) = f(x+\alpha s)$ along the search direction s. The default starting value $\alpha^{(0)} = \alpha^{(k,0)}$ in each line-search algorithm ($\min_{\alpha>0} f(x+\alpha s)$) during the main iteration k is computed in three steps:

1. The first step uses either the difference $df = |f^{(k)} - f^{(k-1)}|$ of the function values during the last two consecutive iterations or the final step length value α^- of the last iteration $k-1$ to compute a first value of $\alpha_1^{(0)}$.

   - Not using the DAMPSTEP=*r* option:

   $$\alpha_1^{(0)} = \begin{cases} step, & \text{if } 0.1 \le step \le 10 \\ 10, & \text{if } step > 10 \\ 0.1, & \text{if } step < 0.1 \end{cases}$$

   with

   $$step = \begin{cases} df/|g^T s|, & \text{if } |g^T s| \ge \epsilon \max(100\,df, 1) \\ 1, & \text{otherwise} \end{cases}$$

This value of $\alpha_1^{(0)}$ can be too large and lead to a difficult or impossible function evaluation, especially for highly nonlinear functions such as the EXP function.

- Using the DAMPSTEP=*r* option:

$$\alpha_1^{(0)} = \min(1, r\alpha^-)$$

The initial value for the new step length can be no larger than r times the final step length α^- of the previous iteration. The default value is $r = 2$.

2. During the first five iterations, the second step enables you to reduce $\alpha_1^{(0)}$ to a smaller starting value $\alpha_2^{(0)}$ using the INSTEP=*r* option:

$$\alpha_2^{(0)} = \min(\alpha_1^{(0)}, r)$$

After more than five iterations, $\alpha_2^{(0)}$ is set to $\alpha_1^{(0)}$.

3. The third step can further reduce the step length by

$$\alpha_3^{(0)} = \min(\alpha_2^{(0)}, \min(10, u))$$

where u is the maximum length of a step inside the feasible region.

The INSTEP=*r* option lets you specify a smaller or larger radius Δ of the trust region used in the first iteration of the trust region, double dogleg, and Levenberg-Marquardt algorithms. The default initial trust region radius $\Delta^{(0)}$ is the length of the scaled gradient (Moré 1978). This step corresponds to the default radius factor of $r = 1$. In most practical applications of the TRUREG, DBLDOG, and LEVMAR algorithms, this choice is successful. However, for bad initial values and highly nonlinear objective functions (such as the EXP function), the default start radius can result in arithmetic overflows. If this happens, you may try decreasing values of INSTEP=*r*, $0 < r < 1$, until the iteration starts successfully. A small factor r also affects the trust region radius $\Delta^{(k+1)}$ of the next steps because the radius is changed in each iteration by a factor $0 < c \leq 4$, depending on the ratio ρ expressing the goodness of quadratic function approximation. Reducing the radius Δ corresponds to increasing the ridge parameter λ, producing smaller steps directed more closely toward the (negative) gradient direction.

Computational Problems

First Iteration Overflows

If you use bad initial values for the parameters, the computation of the value of the objective function (and its derivatives) can lead to arithmetic overflows in the first iteration. The line-search algorithms that work with cubic extrapolation are especially sensitive to arithmetic overflows. If an overflow occurs with an optimization technique that uses line search, you can use the INSTEP= option to reduce the length of the first trial step during the line search of the first five iterations or use the DAMPSTEP or MAXSTEP= option to restrict the step length of the initial α in subsequent iterations. If an arithmetic overflow occurs in the first iteration of the trust region, double dogleg, or Levenberg-Marquardt algorithm, you can use the INSTEP= option to reduce the default trust region radius of the first iteration. You can also change the minimization technique or the line-search method. If none of these methods helps, consider the following actions:

- scale the parameters

- provide better initial values
- use boundary constraints to avoid the region where overflows may happen
- change the algorithm (specified in program statements) which computes the objective function

Problems in Evaluating the Objective Function

The starting point $x^{(0)}$ must be a point that can be evaluated by all the functions involved in your problem. However, during optimization the optimizer may iterate to a point $x^{(k)}$ where the objective function or nonlinear constraint functions and their derivatives cannot be evaluated. If you can identify the problematic region, you can prevent the algorithm from reaching it by adding another constraint to the problem. Another possibility is a modification of the objective function that will produce a large, undesired function value. As a result, the optimization algorithm reduces the step length and stays closer to the point that has been evaluated successfully in the previous iteration. For more information, refer to the section "Missing Values in Program Statements" on page 631.

Problems with Quasi-Newton Methods for Nonlinear Constraints

The sequential quadratic programming algorithm in QUANEW, which is used for solving nonlinearly constrained problems, can have problems updating the Lagrange multiplier vector μ. This usually results in very high values of the Lagrangian function and in *watchdog* restarts indicated in the iteration history. If this happens, there are three actions you can try:

- By default, the Lagrange vector μ is evaluated in the same way as Powell (1982b) describes. This corresponds to VERSION=2. By specifying VERSION=1, a modification of this algorithm replaces the update of the Lagrange vector μ with the original update of Powell (1978a, b), which is used in VF02AD.
- You can use the INSTEP= option to impose an upper bound for the step length α during the first five iterations.
- You can use the INHESSIAN= option to specify a different starting approximation for the Hessian. Choosing only the INHESSIAN option will use the Cholesky factor of a (possibly ridged) finite-difference approximation of the Hessian to initialize the quasi-Newton update process.

Other Convergence Difficulties

There are a number of things to try if the optimizer fails to converge.

- Check the derivative specification:
  If derivatives are specified by using the GRADIENT, HESSIAN, JACOBIAN, CRPJAC, or JACNLC statement, you can compare the specified derivatives with those computed by finite-difference approximations (specifying the FD and FDHESSIAN option). Use the GRADCHECK option to check if the gradient g is correct. For more information, refer to the section "Testing the Gradient Specification" on page 609.

- Forward-difference derivatives specified with the FD= or FDHESSIAN= option may not be precise enough to satisfy strong gradient termination criteria. You may need to specify the more expensive central-difference formulas or use analytical derivatives. The finite-difference intervals may be too small or too big and the finite-difference derivatives may be erroneous. You can specify the FDINT= option to compute better finite-difference intervals.
- Change the optimization technique:
  For example, if you use the default TECH=LEVMAR, you can
  - change to TECH=QUANEW or to TECH=NRRIDG
  - run some iterations with TECH=CONGRA, write the results in an OUTEST= data set, and use them as initial values specified by an INEST= data set in a second run with a different TECH= technique
- Change or modify the update technique and the line-search algorithm:
  This method applies only to TECH=QUANEW, TECH=HYQUAN, or TECH=CONGRA. For example, if you use the default update formula and the default line-search algorithm, you can
  - change the update formula with the UPDATE= option
  - change the line-search algorithm with the LINESEARCH= option
  - specify a more precise line search with the LSPRECISION= option, if you use LINESEARCH=2 or LINESEARCH=3
- Change the initial values by using a grid search specification to obtain a set of good feasible starting values.

Convergence to Stationary Point

The (projected) gradient at a stationary point is zero and that results in a zero step length. The stopping criteria are satisfied.

There are two ways to avoid this situation:

- Use the DECVAR statement to specify a grid of feasible starting points.
- Use the OPTCHECK= option to avoid terminating at the stationary point.

The signs of the eigenvalues of the (reduced) Hessian matrix contain information regarding a stationary point:

- If all eigenvalues are positive, the Hessian matrix is positive definite and the point is a minimum point.
- If some of the eigenvalues are positive and all remaining eigenvalues are zero, the Hessian matrix is positive semidefinite and the point is a minimum or saddle point.
- If all eigenvalues are negative, the Hessian matrix is negative definite and the point is a maximum point.
- If some of the eigenvalues are negative and all remaining eigenvalues are zero, the Hessian matrix is negative semidefinite and the point is a maximum or saddle point.
- If all eigenvalues are zero, the point can be a minimum, maximum, or saddle point.

Precision of Solution

In some applications, PROC NLP may result in parameter estimates that are not precise enough. Usually this means that the procedure terminated too early at a point too far from the optimal point. The termination criteria define the size of the termination region around the optimal point. Any point inside this region can be accepted for terminating the optimization process. The default values of the termination criteria are set to satisfy a reasonable compromise between the computational effort (computer time) and the precision of the computed estimates for the most common applications. However, there are a number of circumstances where the default values of the termination criteria specify a region that is either too large or too small. If the termination region is too large, it can contain points with low precision. In such cases, you should inspect the log or list output to find the message stating which termination criterion terminated the optimization process. In many applications, you can obtain a solution with higher precision by simply using the old parameter estimates as starting values in a subsequent run where you specify a smaller value for the termination criterion that was satisfied at the previous run.

If the termination region is too small, the optimization process may take longer to find a point inside such a region or may not even find such a point due to rounding errors in function values and derivatives. This can easily happen in applications where finite-difference approximations of derivatives are used and the GCONV and ABSGCONV termination criteria are too small to respect rounding errors in the gradient values.

Covariance Matrix

The COV= option must be specified to compute an approximate covariance matrix for the parameter estimates under asymptotic theory for least squares, maximum-likelihood, or Bayesian estimation, with or without corrections for degrees of freedom as specified by the VARDEF= option.

Two groups of six different forms of covariance matrices (and therefore approximate standard errors) can be computed corresponding to the following two situations:

- The LSQ statement is specified, which means that least squares estimates are being computed:

$$\min f(x) = \sum_{i=1}^{m} f_i^2(x)$$

- The MIN or MAX statement is specified, which means that maximum-likelihood or Bayesian estimates are being computed:

$$\text{opt } f(x) = \sum_{i=1}^{m} f_i(x)$$

  where opt is either min or max.

In either case, the following matrices are used:

$$G = \nabla^2 f(x)$$

$$J(f) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

$$JJ(f) = J(f)^T J(f)$$

$$V = J(f)^T \text{diag}(f_i^2) J(f)$$
$$W = J(f)^T \text{diag}(f_i^\dagger) J(f)$$

where

$$f_i^\dagger = \begin{cases} 0, & \text{if } f_i = 0 \\ 1/f_i, & \text{otherwise} \end{cases}$$

For unconstrained minimization, or when none of the final parameter estimates are subjected to linear equality or active inequality constraints, the formulas of the six types of covariance matrices are as follows:

Table 7.3 Central-Difference Approximations

COV		MIN or MAX Statement	LSQ Statement
1	M	$(\text{_NOBS_}/d)G^{-1}JJ(f)G^{-1}$	$(\text{_NOBS_}/d)G^{-1}VG^{-1}$
2	H	$(\text{_NOBS_}/d)G^{-1}$	$\sigma^2 G^{-1}$
3	J	$(1/d)W^{-1}$	$\sigma^2 JJ(f)^{-1}$
4	B	$(1/d)G^{-1}WG^{-1}$	$\sigma^2 G^{-1}JJ(f)G^{-1}$
5	E	$(\text{_NOBS_}/d)JJ(f)^{-1}$	$(1/d)V^{-1}$
6	U	$(\text{_NOBS_}/d)W^{-1}JJ(f)W^{-1}$	$(\text{_NOBS_}/d)JJ(f)^{-1}VJJ(f)^{-1}$

The value of d depends on the VARDEF= option and on the value of the _NOBS_ variable:

$$d = \begin{cases} \max(1, \text{_NOBS_} - \text{_DF_}), & \text{for VARDEF=DF} \\ \text{_NOBS_}, & \text{for VARDEF=N} \end{cases}$$

where _DF_ is either set in the program statements or set by default to n (the number of parameters) and _NOBS_ is either set in the program statements or set by default to *nobs* × *mfun*, where *nobs* is the number of observations in the data set and *mfun* is the number of functions listed in the LSQ, MIN, or MAX statement.

The value σ^2 depends on the specification of the SIGSQ= option and on the value of d:

$$\sigma^2 = \begin{cases} sq \times _NOBS_/d, & \text{if SIGSQ=}sq \text{ is specified} \\ 2f(x^*)/d, & \text{if SIGSQ= is not specified} \end{cases}$$

where $f(x^*)$ is the value of the objective function at the optimal parameter estimates x^*.

The two groups of formulas distinguish between two situations:

- For least squares estimates, the error variance can be estimated from the objective function value and is used in three of the six different forms of covariance matrices. If you have an independent estimate of the error variance, you can specify it with the SIGSQ= option.
- For maximum-likelihood or Bayesian estimates, the objective function should be the logarithm of the likelihood or of the posterior density when using the MAX statement.

For minimization, the inversion of the matrices in these formulas is done so that negative eigenvalues are considered zero, resulting always in a positive semidefinite covariance matrix.

In small samples, estimates of the covariance matrix based on asymptotic theory are often too small and should be used with caution.

If the final parameter estimates are subjected to $n_{act} > 0$ linear equality or active linear inequality constraints, the formulas of the covariance matrices are modified similar to Gallant (1987) and Cramer (1986, p. 38) and additionally generalized for applications with singular matrices. In the constrained case, the value of d used in the scalar factor σ^2 is defined by

$$d = \begin{cases} \max(1, \text{_NOBS_} - \text{_DF_} + n_{act}), & \text{for VARDEF=DF} \\ \text{_NOBS_}, & \text{for VARDEF=N} \end{cases}$$

where n_{act} is the number of active constraints and _NOBS_ is set as in the unconstrained case.

For minimization, the covariance matrix should be positive definite; for maximization it should be negative definite. There are several options available to check for a rank deficiency of the covariance matrix:

- The ASINGULAR=, MSINGULAR=, and VSINGULAR= options can be used to set three singularity criteria for the inversion of the matrix A needed to compute the covariance matrix, when A is either the Hessian or one of the crossproduct Jacobian matrices. The singularity criterion used for the inversion is

  $$|d_{j,j}| \leq \max(ASING, VSING \times |A_{j,j}|, MSING \times \max(|A_{1,1}|, \ldots, |A_{n,n}|))$$

  where $d_{j,j}$ is the diagonal pivot of the matrix A, and *ASING*, *VSING* and *MSING* are the specified values of the ASINGULAR=, VSINGULAR=, and MSINGULAR= options. The default values are

  - *ASING*: the square root of the smallest positive double precision value
  - *MSING*: 1E−12 if the SINGULAR= option is not specified and $\max(10 \times \epsilon, 1\text{E} - 4 \times SINGULAR)$ otherwise, where ϵ is the machine precision
  - *VSING*: 1E−8 if the SINGULAR= option is not specified and the value of SINGULAR otherwise

  NOTE: In many cases, a normalized matrix $D^{-1}AD^{-1}$ is decomposed and the singularity criteria are modified correspondingly.

- If the matrix A is found singular in the first step, a generalized inverse is computed. Depending on the G4= option, a generalized inverse is computed that satisfies either all four or only two Moore-Penrose conditions. If the number of parameters n of the application is less than or equal to G4=*i*, a G4 inverse is computed; otherwise only a G2 inverse is computed. The G4 inverse is computed by (the computationally very expensive but numerically stable) eigenvalue decomposition; the G2 inverse is computed by Gauss transformation. The G4 inverse is computed using the eigenvalue decomposition $A = Z\Lambda Z^T$, where Z is the orthogonal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues, $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$. If the PEIGVAL option is specified, the eigenvalues λ_i are displayed. The G4 inverse of A is set to

  $$A^- = Z\Lambda^- Z^T$$

  where the diagonal matrix $\Lambda^- = \text{diag}(\lambda_1^-, \ldots, \lambda_n^-)$ is defined using the COVSING= option:

  $$\lambda_i^- = \begin{cases} 1/\lambda_i, & \text{if } |\lambda_i| > COVSING \\ 0, & \text{if } |\lambda_i| \leq COVSING \end{cases}$$

  If the COVSING= option is not specified, the nr smallest eigenvalues are set to zero, where nr is the number of rank deficiencies found in the first step.

For optimization techniques that do not use second-order derivatives, the covariance matrix is usually computed using finite-difference approximations of the derivatives. By specifying TECH=NONE, any of the covariance matrices can be computed using analytical derivatives. The covariance matrix specified by the COV= option can be displayed (using the PCOV option) and is written to the OUTEST= data set.

Input and Output Data Sets

DATA= Input Data Set

The DATA= data set is used only to specify an objective function f that is a combination of m other functions f_i. For each function f_i, $i = 1, \ldots, m$, listed in a MAX, MIN, or LSQ statement, each observation l, $l = 1, \ldots, nobs$, in the DATA= data set defines a specific function f_{il} that is evaluated by substituting the values of the variables of this observation into the program statements. If the MAX or MIN statement is used, the $m \times nobs$ specific functions f_{il} are added to a single objective function f. If the LSQ statement is used, the sum-of-squares f of the $m \times nobs$ specific functions f_{il} is minimized. The NOMISS option causes observations with missing values to be skipped.

INEST= Input Data Set

The INEST= (or INVAR=, or ESTDATA=) input data set can be used to specify the initial values of the parameters defined in a DECVAR statement as well as boundary constraints and the more general linear constraints which could be imposed on these parameters. This form of input is similar to the dense format input used in PROC LP.

The variables of the INEST= data set are

- a character variable _TYPE_ that indicates the type of the observation
- n numeric variables with the parameter names used in the DECVAR statement
- the BY variables that are used in a DATA= input data set
- a numeric variable _RHS_ specifying the right-hand-side constants (needed only if linear constraints are used)
- additional variables with names corresponding to constants used in the program statements

The content of the _TYPE_ variable defines the meaning of the observation of the INEST= data set. PROC NLP recognizes the following _TYPE_ values:

- PARMS, which specifies initial values for parameters. Additional variables can contain the values of constants that are referred to in program statements. The values of the constants in the PARMS observation initialize the constants in the program statements.
- UPPERBD | UB, which specifies upper bounds. A missing value indicates that no upper bound is specified for the parameter.
- LOWERBD | LB, which specifies lower bounds. A missing value indicates that no lower bound is specified for the parameter.

- LE | <= | <, which specifies linear constraint $\sum_j a_{ij}x_j \leq b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.
- GE | >= | >, which specifies linear constraint $\sum_j a_{ij}x_j \geq b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.
- EQ | =, which specifies linear constraint $\sum_j a_{ij}x_j = b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.

The constraints specified in an INEST= data set are added to the constraints specified in the BOUNDS and LINCON statements. You can use an OUTEST= data set as an INEST= data set in a subsequent run of PROC NLP. However, be aware that the OUTEST= data set also contains the boundary and general linear constraints specified in the previous run of PROC NLP. When you are using this OUTEST= data set without changes as an INEST= data set, PROC NLP adds the constraints from the data set to the constraints specified by a BOUNDS and LINCON statement. Although PROC NLP automatically eliminates multiple identical constraints you should avoid specifying the same constraint twice.

INQUAD= Input Data Set

Two types of INQUAD= data sets can be used to specify the objective function of a quadratic programming problem for TECH=QUADAS or TECH=LICOMP,

$$f(x) = \frac{1}{2}x^T Gx + g^T x + c, \quad \text{with} \quad G^T = G$$

The *dense* INQUAD= data set must contain all numerical values of the symmetric matrix G, the vector g, and the scalar c. Using the *sparse* INQUAD= data set enables you to specify only the nonzero positions in matrix G and vector g. Those locations that are not set by the *sparse* INQUAD= data set are assumed to be zero.

Dense INQUAD= Data Set

A dense INQUAD= data set must contain two character variables, _TYPE_ and _NAME_, and at least n numeric variables whose names are the parameter names. The _TYPE_ variable takes the following values:

- QUAD lists the n values of the row of the G matrix that is defined by the parameter name used in the _NAME_ variable.
- LINEAR lists the n values of the g vector.
- CONST sets the value of the scalar c and cannot contain different numerical values; however, it could contain up to $n - 1$ missing values.
- PARMS specifies initial values for parameters.
- UPPERBD | UB specifies upper bounds. A missing value indicates that no upper bound is specified.
- LOWERBD | LB specifies lower bounds. A missing value indicates that no lower bound is specified.
- LE | <= | < specifies linear constraint $\sum_j a_{ij}x_j \leq b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.
- GE | >= | > specifies linear constraint $\sum_j a_{ij}x_j \geq b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.

- EQ | = specifies linear constraint $\sum_j a_{ij} x_j = b_i$. The n parameter values contain the coefficients a_{ij}, and the _RHS_ variable contains the right-hand side b_i. Missing values indicate zeros.

Constraints specified in a dense INQUAD= data set are added to the constraints specified in BOUNDS and LINCON statements.

Sparse INQUAD= Data Set

A sparse INQUAD= data set must contain three character variables _TYPE_, _ROW_, and _COL_, and one numeric variable _VALUE_. The _TYPE_ variable can assume two values:

- QUAD specifies that the _ROW_ and _COL_ variables define the row and column locations of the values in the G matrix.
- LINEAR specifies that the _ROW_ variable defines the row locations of the values in the g vector. The _COL_ variable is not used.

Using both the MODEL= option and the INCLUDE statement with the same model file will include the file twice (erroneous in most cases).

OUT= Output Data Set

The OUT= data set contains those variables of a DATA= input data set that are referred to in the program statements and additional variables computed by the program statements for the objective function. Specifying the NOMISS option enables you to skip observations with missing values in variables used in the program statements. The OUT= data set can also contain first- and second-order derivatives of these variables if the OUTDER= option is specified. The variables and derivatives are the final parameter estimates x^* or (for TECH=NONE) the initial value x^0.

The variables of the OUT= data set are

- the BY variables and all other variables that are used in a DATA= input data set and referred to in the program code
- a variable _OBS_ containing the number of observations read from a DATA= input data set, where the counting is restarted with the start of each BY group. If there is no DATA= input data set, then _OBS_=1.
- a character variable _TYPE_ naming the type of the observation
- the parameter variables listed in the DECVAR statement
- the function variables listed in the MIN, MAX, or LSQ statement
- all other variables computed in the program statements
- the character variable _WRT_ (if OUTDER=1) containing the *with respect to* variable for which the first-order derivatives are written in the function variables
- the two character variables _WRT1_ and _WRT2_ (if OUTDER=2) containing the two *with respect to* variables for which the first- and second-order derivatives are written in the function variables

OUTEST= Output Data Set

The OUTEST= or OUTVAR= output data set saves the optimization solution of PROC NLP. You can use the OUTEST= or OUTVAR= data set as follows:

- to save the values of the objective function on grid points to examine, for example, surface plots using PROC G3D (use the OUTGRID option)
- to avoid any costly computation of analytical (first- or second-order) derivatives during optimization when they are needed only upon termination. In this case a two-step approach is recommended:
  1. In a first execution, the optimization is done; that is, optimal parameter estimates are computed, and the results are saved in an OUTEST= data set.
  2. In a subsequent execution, the optimal parameter estimates in the previous OUTEST= data set are read in an INEST= data set and used with TECH=NONE to compute further results, such as analytical second-order derivatives or some kind of covariance matrix.
- to restart the procedure using parameter estimates as initial values
- to split a time-consuming optimization problem into a series of smaller problems using intermediate results as initial values in subsequent runs. (Refer to the MAXTIME=, MAXIT=, and MAXFUNC= options to trigger stopping.)
- to write the value of the objective function, the parameter estimates, the time in seconds starting at the beginning of the optimization process and (if available) the gradient to the OUTEST= data set during the iterations. After the PROC NLP run is completed, the convergence progress can be inspected by graphically displaying the iterative information. (Refer to the OUTITER option.)

The variables of the OUTEST= data set are

- the BY variables that are used in a DATA= input data set
- a character variable _TECH_ naming the optimization technique used
- a character variable _TYPE_ specifying the type of the observation
- a character variable _NAME_ naming the observation. For a linear constraint, the _NAME_ variable indicates whether the constraint is active at the solution. For the initial observations, the _NAME_ variable indicates if the number in the _RHS_ variable corresponds to the number of positive, negative, or zero eigenvalues.
- n numeric variables with the parameter names used in the DECVAR statement. These variables contain a point x of the parameter space, lower or upper bound constraints, or the coefficients of linear constraints.
- a numeric variable _RHS_ (right-hand side) that is used for the right-hand-side value b_i of a linear constraint or for the value $f = f(x)$ of the objective function at a point x of the parameter space
- a numeric variable _ITER_ that is zero for initial values, equal to the iteration number for the OUTITER output, and missing for the result output

The _TYPE_ variable identifies how to interpret the observation. If _TYPE_ is

- PARMS then parameter-named variables contain the coordinates of the resulting point x^*. The _RHS_ variable contains $f(x^*)$.
- INITIAL then parameter-named variables contain the feasible starting point $x^{(0)}$. The _RHS_ variable contains $f(x^{(0)})$.
- GRIDPNT then (if the OUTGRID option is specified) parameter-named variables contain the coordinates of any point $x^{(k)}$ used in the grid search. The _RHS_ variable contains $f(x^{(k)})$.
- GRAD then parameter-named variables contain the gradient at the initial or final estimates.
- STDERR then parameter-named variables contain the approximate standard errors (square roots of the diagonal elements of the covariance matrix) if the COV= option is specified.
- _NOBS_ then (if the COV= option is specified) all parameter variables contain the value of _NOBS_ used in computing the σ^2 value in the formula of the covariance matrix.
- UPPERBD | UB then (if there are boundary constraints) the parameter variables contain the upper bounds.
- LOWERBD | LB then (if there are boundary constraints) the parameter variables contain the lower bounds.
- NACTBC then all parameter variables contain the number n_{abc} of active boundary constraints at the solution x^*.
- ACTBC then (if there are active boundary constraints) the observation indicate which parameters are actively constrained, as follows:

  NAME=GE the active lower bounds

  NAME=LE the active upper bounds

  NAME=EQ the active equality constraints

- NACTLC then all parameter variables contain the number n_{alc} of active linear constraints that are recognized as linearly independent.
- NLDACTLC then all parameter variables contain the number of active linear constraints that are recognized as linearly dependent.
- LE then (if there are linear constraints) the observation contains the ith linear constraint $\sum_j a_{ij}x_j \leq b_i$. The parameter variables contain the coefficients a_{ij}, $j = 1, \ldots, n$, and the _RHS_ variable contains b_i. If the constraint i is active at the solution x^*, then _NAME_=ACTLC or _NAME_=LDACTLC.
- GE then (if there are linear constraints) the observation contains the ith linear constraint $\sum_j a_{ij}x_j \geq b_i$. The parameter variables contain the coefficients a_{ij}, $j = 1, \ldots, n$, and the _RHS_ variable contains b_i. If the constraint i is active at the solution x^*, then _NAME_=ACTLC or _NAME_=LDACTLC.
- EQ then (if there are linear constraints) the observation contains the ith linear constraint $\sum_j a_{ij}x_j = b_i$. The parameter variables contain the coefficients a_{ij}, $j = 1, \ldots, n$, the _RHS_ variable contains b_i, and _NAME_=ACTLC or _NAME_=LDACTLC.

- LAGRANGE then (if at least one of the linear constraints is an equality constraint or an active inequality constraint) the observation contains the vector of Lagrange multipliers. The Lagrange multipliers of active boundary constraints are listed first followed by those of active linear constraints and those of active nonlinear constraints. Lagrange multipliers are available only for the set of linearly independent active constraints.
- PROJGRAD then (if there are linear constraints) the observation contains the $n - n_{act}$ values of the projected gradient $g_Z = Z^T g$ in the variables corresponding to the first $n - n_{act}$ parameters.
- JACOBIAN then (if the PJACOBI or OUTJAC option is specified) the m observations contain the m rows of the $m \times n$ Jacobian matrix. The _RHS_ variable contains the row number l, $l = 1, \ldots, m$.
- HESSIAN then the first n observations contain the n rows of the (symmetric) Hessian matrix. The _RHS_ variable contains the row number j, $j = 1, \ldots, n$, and the _NAME_ variable contains the corresponding parameter name.
- PROJHESS then the first $n - n_{act}$ observations contain the $n - n_{act}$ rows of the projected Hessian matrix $Z^T G Z$. The _RHS_ variable contains the row number j, $j = 1, \ldots, n - n_{act}$, and the _NAME_ variable is blank.
- CRPJAC then the first n observations contain the n rows of the (symmetric) crossproduct Jacobian matrix at the solution. The _RHS_ variable contains the row number j, $j = 1, \ldots, n$, and the _NAME_ variable contains the corresponding parameter name.
- PROJCRPJ then the first $n - n_{act}$ observations contain the $n - n_{act}$ rows of the projected crossproduct Jacobian matrix $Z^T (J^T J) Z$. The _RHS_ variable contains the row number j, $j = 1, \ldots, n - n_{act}$, and the _NAME_ variable is blank.
- COV1, COV2, COV3, COV4, COV5, or COV6 then (depending on the COV= option) the first n observations contain the n rows of the (symmetric) covariance matrix of the parameter estimates. The _RHS_ variable contains the row number j, $j = 1, \ldots, n$, and the _NAME_ variable contains the corresponding parameter name.
- DETERMIN contains the determinant $det = a \times 10^b$ of the matrix specified by the value of the _NAME_ variable where a is the value of the first variable in the DECVAR statement and b is in _RHS_.
- NEIGPOS, NEIGNEG, or NEIGZER then the _RHS_ variable contains the number of positive, negative, or zero eigenvalues of the matrix specified by the value of the _NAME_ variable.
- COVRANK then the _RHS_ variable contains the rank of the covariance matrix.
- SIGSQ then the _RHS_ variable contains the scalar factor of the covariance matrix.
- _TIME_ then (if the OUTITER option is specified) the _RHS_ variable contains the number of seconds passed since the start of the optimization.
- TERMINAT then if optimization terminated at a point satisfying one of the termination criteria, an abbreviation of the corresponding criteria is given to the _NAME_ variable. Otherwise _NAME_=PROBLEMS.

If for some reason the procedure does not terminate successfully (for example, no feasible initial values can be computed or the function value or derivatives at the starting point cannot be computed), the OUTEST= data set may contain only part of the observations (usually only the PARMS and GRAD observation).

NOTE: Generally you can use an OUTEST= data set as an INEST= data set in a further run of PROC NLP. However, be aware that the OUTEST= data set also contains the boundary and general linear constraints specified in the previous run of PROC NLP. When you are using this OUTEST= data set without changes as an INEST= data set, PROC NLP adds the constraints from the data set to the constraints specified by a BOUNDS or LINCON statement. Although PROC NLP automatically eliminates multiple identical constraints you should avoid specifying the same constraint twice.

Output of Profiles

The following observations are written to the OUTEST= data set only when the PROFILE statement or CLPARM option is specified.

Table 7.4 Output of Profiles

TYPE	_NAME_	_RHS_	Meaning of Observation
PLC_LOW	parname	y value	coordinates of lower CL for α
PLC_UPP	parname	y value	coordinates of upper CL for α
WALD_CL	LOWER	y value	lower Wald CL for α in _ALPHA_
WALD_CL	UPPER	y value	upper Wald CL for α in _ALPHA_
PL_CL	LOWER	y value	lower PL CL for α in _ALPHA_
PL_CL	UPPER	y value	upper PL CL for α in _ALPHA_
PROFILE	L(THETA)	missing	y value corresponding to x in following _NAME_=THETA
PROFILE	THETA	missing	x value corresponding to y in previous _NAME_=L(THETA)

Assume that the PROFILE statement specifies n_p parameters and n_α confidence levels. For CLPARM, $n_p = n$ and $n_\alpha = 4$.

- _TYPE_=PLC_LOW and _TYPE_=PLC_UPP:
  If the CLPARM= option or the PROFILE statement with the OUTTABLE option is specified, then the complete set θ of parameter estimates (rather than only the confidence limit $x = \theta_j$) is written to the OUTEST= data set for each side of the confidence interval. This output may be helpful for further analyses on how small changes in $x = \theta_j$ affect the changes in the other $\theta_i, i \neq j$. The _ALPHA_ variable contains the corresponding value of α. There should be no more than $2n_\alpha n_p$ observations. If the confidence limit cannot be computed, the corresponding observation is not available.

- _TYPE_=WALD_CL:
  If CLPARM=WALD, CLPARM=BOTH, or the PROFILE statement with α values is specified, then the Wald confidence limits are written to the OUTEST= data set for each of the default or specified values of α. The _ALPHA_ variable contains the corresponding value of α. There should be $2n_\alpha$ observations.

- _TYPE_=PL_CL:
  If CLPARM=PL, CLPARM=BOTH, or the PROFILE statement with α values is specified, then the PL confidence limits are written to the OUTEST= data set for each of the default or specified values of α. The _ALPHA_ variable contains the corresponding values of α. There should be $2n_\alpha$ observations; some observations may have missing values.

- _TYPE_=PROFILE:
  If CLPARM=PL, CLPARM=BOTH, or the CLPARM= statement with or without α values is specified, then a set of (x, y) point coordinates in two adjacent observations with _NAME_=L(THETA) (y value) and _NAME_=THETA (x value) is written to the OUTEST= data set. The _RHS_ and _ALPHA_ variables are not used (are set to missing). The number of observations depends on the difficulty of the optimization problems.

OUTMODEL= Output Data Set

The program statements for objective functions, nonlinear constraints, and derivatives can be saved into an OUTMODEL= output data set. This data set can be used in an INCLUDE program statement or as a MODEL= input data set in subsequent calls of PROC NLP. The OUTMODEL= option is similar to the option used in PROC MODEL in SAS/ETS software.

Storing Programs in Model Files

Models can be saved to and recalled from SAS catalog files. SAS catalogs are special files which can store many kinds of data structures as separate units in one SAS file. Each separate unit is called an entry, and each entry has an entry type that identifies its structure to the SAS system.

In general, to save a model, use the OUTMODEL=*name* option in the PROC NLP statement, where *name* is specified as *libref.catalog.entry*, *libref.entry*, or *entry*. The *libref*, *catalog*, and *entry* names must be valid SAS names no more than 8 characters long. The *catalog* name is restricted to 7 characters on the CMS operating system. If not given, the *catalog* name defaults to MODELS, and the *libref* defaults to WORK. The entry type is always MODEL. Thus, OUTMODEL=X writes the model to the file WORK.MODELS.X.MODEL.

The MODEL= option is used to read in a model. A list of model files can be specified in the MODEL= option, and a range of names with numeric suffixes can be given, as in MODEL=(MODEL1-MODEL10). When more than one model file is given, the list must be placed in parentheses, as in MODEL=(A B C). If more than one model file is specified, the files are combined in the order listed in the MODEL= option.

When the MODEL= option is specified in the PROC NLP statement and model definition statements are also given later in the PROC NLP step, the model files are read in first, in the order listed, and the model program specified in the PROC NLP step is appended after the model program read from the MODEL= files.

The INCLUDE statement can be used to append model code to the current model code. The contents of the model files are inserted into the current model at the position where the INCLUDE statement appears.

Note that the following statements are not part of the program code that is written to an OUTMODEL= data set: MIN, MAX, LSQ, MINQUAD, MAXQUAD, DECVAR, BOUNDS, BY, CRPJAC, GRADIENT, HESSIAN, JACNLC, JACOBIAN, LABEL, LINCON, MATRIX, and NLINCON.

Displayed Output

Procedure Initialization

After the procedure has processed the problem, it displays summary information about the problem and the options that you have selected. It may also display a list of linearly dependent constraints and other information about the constraints and parameters.

Optimization Start

At the start of optimization the procedure displays

- the number of constraints that are active at the starting point, or more precisely, the number of constraints that are currently members of the working set. If this number is followed by a plus sign, there are more active constraints, of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the starting point
- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- for the TRUREG and LEVMAR subroutines, the initial radius of the trust region around the starting point

Iteration History

In general, the iteration history consists of one line of output containing the most important information for each iteration. The iteration-extensive Nelder-Mead simplex method, however, displays only one line for several internal iterations. This technique skips the output for some iterations because

- some of the termination tests (size and standard deviation) are rather time-consuming compared to the simplex operations and are done once every five simplex operations
- the resulting history output is smaller

The _LIST_ variable (refer to the section "Program Statements" on page 590) also enables you to display the parameter estimates $x^{(k)}$ and the gradient $g^{(k)}$ in all or some selected iterations k.

The iteration history always includes the following (the words in parentheses indicate the column header output):

- the iteration number (iter)
- the number of iteration restarts (nrest)
- the number of function calls (nfun)
- the number of active constraints (act)
- the value of the optimization criterion (optcrit)
- the difference between adjacent function values (difcrit)
- the maximum of the absolute (projected) gradient components (maxgrad)

An apostrophe trailing the number of active constraints indicates that at least one of the active constraints was released from the active set due to a significant Lagrange multiplier.

The optimization history is displayed by default because it is important to check for possible convergence problems.

Optimization Termination

The output of the optimization history ends with a short output of information concerning the optimization result:

- the number of constraints that are active at the final point, or more precisely, the number of constraints that are currently members of the working set. When this number is followed by a plus sign, it indicates that there are more active constraints of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the final point
- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- other information that is specific for the optimization technique

The NOPRINT option suppresses all output to the list file and only errors, warnings, and notes are displayed to the log file. The PALL option sets a large group of some of the commonly used specific displaying options, the PSHORT option suppresses some, and the PSUMMARY option suppresses almost all of the default output. The following table summarizes the correspondence between the general and the specific print options.

Table 7.5 Optimization Termination

Output Options	PALL	default	PSHORT	PSUMMARY	Output
	y	y	y	y	Summary of optimization
	y	y	y	n	Parameter estimates
	y	y	y	n	Gradient of objective func
PHISTORY	y	y	y	n	Iteration history
PINIT	y	y	n	n	Setting of initial values
	y	y	n	n	Listing of constraints
PGRID	y	n	n	n	Results of grid search
PNLCJAC	y	n	n	n	Jacobian nonlin. constr.
PFUNCTION	y	n	n	n	Values of functions
PEIGVAL	y	n	n	n	Eigenvalue distribution
PCRPJAC	y	n	n	n	Crossproduct Jacobian
PHESSIAN	y	n	n	n	Hessian matrix
PSTDERR	y	n	n	n	Approx. standard errors
PCOV	y	n	n	n	Covariance matrices
PJACOBI	n	n	n	n	Jacobian
LIST	n	n	n	n	Model program, variables
LISTCODE	n	n	n	n	Compiled model program

Convergence Status

Upon termination, the NLP procedure creates an ODS table called "ConvergenceStatus." You can use this name to reference the table when using the Output Delivery System (ODS) to select tables and create output data sets. Within the "ConvergenceStatus" table there are two variables, "Status" and "Reason," which contain the status of the optimization run. For the "Status" variable, a value of zero indicates that one of the convergence criteria is satisfied; a nonzero value indicates otherwise. In all cases, an explicit interpretation of the status code is displayed as a string stored in the "Reason" variable. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Missing Values

Missing Values in Program Statements

There is one very important reason for using missing values in program statements specifying the values of the objective functions and derivatives: it may not be possible to evaluate the program statements for a particular point x. For example, the extrapolation formula of one of the line-search algorithms may generate large x values for which the EXP function cannot be evaluated without floating point overflow. The compiler of the program statements may check for such situations automatically, but it would be safer if you check the feasibility of your program statements. In some cases, the specification of boundary or linear constraints for parameters can avoid such situations. In many other cases, you can indicate that x is a *bad* point simply by returning a missing value for the objective function. In such cases the optimization algorithms in PROC NLP shorten the step length α or reduce the trust region radius so that the next point will be closer to the point that was already successfully evaluated at the last iteration. Note that the starting point $x^{(0)}$ must be a point for which the program statements can be evaluated.

Missing Values in Input Data Sets

Observations with missing values in the DATA= data set for variables used in the objective function can lead to a missing value of the objective function implying that the corresponding BY group of data is not processed. The NOMISS option can be used to skip those observations of the DATA= data set for which relevant variables have missing values. Relevant variables are those that are referred to in program statements.

There can be different reasons to include observations with missing values in the INEST= data set. The value of the _RHS_ variable is not used in some cases and can be missing. Missing values for the variables corresponding to parameters in the _TYPE_ variable are as follows:

- PARMS observations cause those parameters to have initial values assigned by the DECVAR statement or by the RANDOM= or INITIAL= option.
- UPPERBD or LOWERBD observations cause those parameters to be unconstrained by upper or lower bounds.
- LE, GE, or EQ observations cause those parameters to have zero values in the constraint.

In general, missing values are treated as zeros.

Computational Resources

Since nonlinear optimization is an iterative process that depends on many factors, it is difficult to estimate how much computer time is necessary to compute an optimal solution satisfying one of the termination criteria. The MAXTIME=, MAXITER=, and MAXFUNC= options can be used to restrict the amount of real time, the number of iterations, and the number of function calls in a single run of PROC NLP.

In each iteration k, the NRRIDG and LEVMAR techniques use symmetric Householder transformations to decompose the $n \times n$ Hessian (crossproduct Jacobian) matrix G,

$$G = V^T T V\ , \quad V \text{ orthogonal}, \quad T \text{ tridiagonal}$$

to compute the (Newton) search direction s:

$$s^{(k)} = -G^{(k-1)} g^{(k)}\ , \quad k = 1, 2, 3, \ldots$$

The QUADAS, TRUREG, NEWRAP, and HYQUAN techniques use the Cholesky decomposition to solve the same linear system while computing the search direction. The QUANEW, DBLDOG, CONGRA, and NMSIMP techniques do not need to invert or decompose a Hessian or crossproduct Jacobian matrix and thus require fewer computational resources then the first group of techniques.

The larger the problem, the more time is spent computing function values and derivatives. Therefore, many researchers compare optimization techniques by counting and comparing the respective numbers of function, gradient, and Hessian (crossproduct Jacobian) evaluations. You can save computer time and memory by specifying derivatives (using the GRADIENT, JACOBIAN, CRPJAC, or HESSIAN statement) since you will typically produce a more efficient representation than the internal derivative compiler.

Finite-difference approximations of the derivatives are expensive since they require additional function or gradient calls.

- Forward-difference formulas:
  - First-order derivatives: n additional function calls are needed.
  - Second-order derivatives based on function calls only: for a dense Hessian, $n(n+3)/2$ additional function calls are needed.
  - Second-order derivatives based on gradient calls: n additional gradient calls are needed.
- Central-difference formulas:
  - First-order derivatives: $2n$ additional function calls are needed.
  - Second-order derivatives based on function calls only: for a dense Hessian, $2n(n+1)$ additional function calls are needed.
  - Second-order derivatives based on gradient: $2n$ additional gradient calls are needed.

Many applications need considerably more time for computing second-order derivatives (Hessian matrix) than for first-order derivatives (gradient). In such cases, a (dual) quasi-Newton or conjugate gradient technique is recommended, which does not require second-order derivatives.

The following table shows for each optimization technique which derivatives are needed (FOD: first-order derivatives; SOD: second-order derivatives), what kinds of constraints are supported (BC: boundary constraints; LIC: linear constraints), and the minimal memory (number of double floating point numbers) required. For various reasons, there are additionally about $7n + m$ double floating point numbers needed.

Quadratic Programming	FOD	SOD	BC	LIC	Memory
LICOMP	-	-	x	x	$18n + 3nn$
QUADAS	-	-	x	x	$1n + 2nn/2$
General Optimization	FOD	SOD	BC	LIC	Memory
TRUREG	x	x	x	x	$4n + 2nn/2$
NEWRAP	x	x	x	x	$2n + 2nn/2$
NRRIDG	x	x	x	x	$6n + nn/2$
QUANEW	x	-	x	x	$1n + nn/2$
DBLDOG	x	-	x	x	$7n + nn/2$
CONGRA	x	-	x	x	$3n$
NMSIMP	-	-	x	x	$4n + nn$
Least Squares	FOD	SOD	BC	LIC	Memory
LEVMAR	x	-	x	x	$6n + nn/2$
HYQUAN	x	-	x	x	$2n + nn/2 + 3m$

Notes:

- Here, n denotes the number of parameters, nn the squared number of parameters, and $nn/2 := n(n+1)/2$.
- The value of m is the product of the number of functions specified in the MIN, MAX, or LSQ statement and the maximum number of observations in each BY group of a DATA= input data set. The following table also contains the number v of variables in the DATA= data set that are used in the program statements.
- For a diagonal Hessian matrix, the $nn/2$ term in QUADAS, TRUREG, NEWRAP, and NRRIDG is replaced by n.
- If the TRUREG, NRRIDG, or NEWRAP method is used to minimize a least squares problem, the second derivatives are replaced by the crossproduct Jacobian matrix.
- The memory needed by the TECH=NONE specification depends on the output specifications (typically, it needs $3n + nn/2$ double floating point numbers and an additional mn if the Jacobian matrix is required).

The total amount of memory needed to run an optimization technique consists of the technique-specific memory listed in the preceding table, plus additional blocks of memory as shown in the following table.

	double	int	long	8byte
Basic Requirement	$7n + m$	n	$3n$	$n + m$
DATA= data set	v	-	-	v
JACOBIAN statement	$m(n+2)$	-	-	-
CRPJAC statement	$nn/2$	-	-	-
HESSIAN statement	$nn/2$	-	-	-
COV= option	$(2*)nn/2 + n$	-	-	-
Scaling vector	n	-	-	-
BOUNDS statement	$2n$	n	-	-
Bounds in INEST=	$2n$	-	-	-
LINCON and TRUREG	$c(n+1) + nn + nn/2 + 4n$	$3c$	-	-
LINCON and other	$c(n+1) + nn + 2nn/2 + 4n$	$3c$	-	-

Notes:

- For TECH=LICOMP, the total amount of memory needed for the linear or boundary constrained case is $18(n+c)+3(n+c)(n+c)$, where c is the number of constraints.
- The amount of memory needed to specify derivatives with a GRADIENT, JACOBIAN, CRPJAC, or HESSIAN statement (shown in this table) is small compared to that needed for using the internal function compiler to compute the derivatives. This is especially so for second-order derivatives.
- If the CONGRA technique is used, specifying the GRADCHECK=DETAIL option requires an additional $nn/2$ double floating point numbers to store the finite-difference Hessian matrix.

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory used by the SAS System. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit, which is sufficient for many applications. However, to solve most realistic optimization problems, the NLP procedure might require more memory. Increasing the memory limit can reduce the chance of an out-of-memory condition.

NOTE: The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify -MEMSIZE 0 to indicate all available memory should be used, but this setting should be used with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify -MEMSIZE 0. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, -MEMSIZE 500M might be sufficient to enable the procedure to run without an out-of-memory condition. When problems have millions of variables, -MEMSIZE 1000M or higher might be needed. These are "rules of thumb"—problems with atypical structure, density, or other characteristics can increase the optimizer's memory requirements.

The MEMSIZE option can be specified at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the SAS Companion for your operating environment.

To report a procedure's memory consumption, you can use the FULLSTIMER option. The syntax is described in the SAS Companion for your operating environment.

Rewriting NLP Models for PROC OPTMODEL

This section covers techniques for converting NLP procedure models to OPTMODEL procedure models. For information about the OPTMODEL procedure, see Chapter 5, "The OPTMODEL Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

To illustrate the basics, consider the following first version of the NLP model for Example 7.7:

```
/*************************************************************/
/* Rewriting NLP Models for PROC OPTMODEL                    */
/*************************************************************/

proc nlp all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools = 1;
   bounds 0 <= amountx amounty amounta amountb amountc,
               amountx <= 100,
               amounty <= 200,
          0 <= pooltox pooltoy ctox ctoy,
          1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
          pooltox + ctox = amountx,
          pooltoy + ctoy = amounty,
          ctox + ctoy    = amountc;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

These statements define a model that has bounds, linear constraints, nonlinear constraints, and a simple objective function. The following statements are a straightforward conversion of the PROC NLP statements to PROC OPTMODEL form:

```
proc optmodel;
   var amountx init 1 >= 0 <= 100,
       amounty init 1 >= 0 <= 200;
   var amounta init 1 >= 0,
       amountb init 1 >= 0,
       amountc init 1 >= 0;
   var pooltox init 1 >= 0,
       pooltoy init 1 >= 0;
   var ctox init 1 >= 0,
       ctoy init 1 >= 0;
   var pools init 1 >=1 <= 3;
   con amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
   number costa, costb, costc, costx, costy;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   max f = costx * amountx + costy * amounty
          - costa * amounta - costb * amountb - costc * amountc;
   con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
```

```
        nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
        nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
              = 0;
   solve;
   print amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
```

The PROC OPTMODEL variable declarations are split into individual declarations because PROC OPTMODEL does not permit name lists in its declarations. In the OPTMODEL procedure, variable bounds are part of the variable declaration instead of a separate BOUNDS statement. The PROC NLP statements are as follows:

```
parms amountx amounty amounta amountb amountc
      pooltox pooltoy ctox ctoy pools = 1;
bounds 0 <= amountx amounty amounta amountb amountc,
            amountx <= 100,
            amounty <= 200,
       0 <= pooltox pooltoy ctox ctoy,
       1 <= pools <= 3;
```

The following PROC OPTMODEL statements are equivalent to the PROC NLP statements:

```
var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
var ctox init 1 >= 0,
    ctoy init 1 >= 0;
var pools init 1 >= 1 <= 3;
```

The linear constraints are declared in the NLP model with the following statement:

```
lincon amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
```

The following linear constraint declarations in the PROC OPTMODEL model are quite similar to the PROC NLP LINCON declarations:

```
con amounta + amountb = pooltox + pooltoy,
    pooltox + ctox = amountx,
    pooltoy + ctoy = amounty,
    ctox + ctoy    = amountc;
```

But PROC OPTMODEL provides much more flexibility in defining linear constraints. For example, coefficients can be named parameters or any other expression that evaluates to a constant.

The cost parameters are declared explicitly in the PROC OPTMODEL model. Unlike the DATA step or PROC NLP, PROC OPTMODEL requires names to be declared before they are used. There are multiple ways to set the values of these parameters. The preceding example used assignments. The values could have been made part of the declaration by using the INIT *expression* clause or the = *expression* clause. The values could also have been read from a data set with the READ DATA statement.

Note in the original NLP statements that the assignment to a parameter such as costa occurs every time the objective function is evaluated. However, the assignment occurs just once in the PROC OPTMODEL code, when the assignment statement is processed. This works because the values are constant. But the PROC OPTMODEL statements permit the parameters to be reassigned later to interactively modify the model.

The following statements define the objective f in the NLP model:

```
max f;
. . .
f = costx * amountx + costy * amounty
    - costa * amounta - costb * amountb - costc * amountc;
```

The PROC OPTMODEL version of the objective is defined with the same expression text, as follows:

```
max f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
```

But in PROC OPTMODEL the MAX statement and the assignment to the name f in the PROC NLP statements are combined. There are advantages and disadvantages to this approach. The PROC OPTMODEL formulation is much closer to the mathematical formulation of the model. However, if there are multiple intermediate variables being used to structure the objective, then multiple IMPVAR declarations are required.

In the PROC NLP model the nonlinear constraints use the following syntax:

```
nlincon nlc1-nlc2 >= 0.,
        nlc3 = 0.;
. . .
nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
```

In the PROC OPTMODEL model the equivalent statements are as follows:

```
con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
    nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
    nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
          = 0;
```

The nonlinear constraints in PROC OPTMODEL use the same syntax as linear constraints. In fact, if the variable pools were declared as a parameter, then all the preceding constraints would be linear. The nonlinear

constraint in PROC OPTMODEL combines the NLINCON statement of PROC NLP with the assignment in the PROC NLP statements. As in objective expressions, objective names can be used in nonlinear constraint expressions to structure the formula.

The PROC OPTMODEL model does not use a RUN statement to invoke the solver. Instead the solver is invoked interactively by the SOLVE statement in PROC OPTMODEL. By default, the OPTMODEL procedure prints much less data about the optimization process. Generally these data consist of messages from the solver (such as the termination reason) in addition to a short status display. The PROC OPTMODEL statements add a PRINT statement in order to display the variable estimates from the solver.

The model for Example 7.8 illustrates how to convert PROC NLP statements that handle arrays into PROC OPTMODEL form. The PROC NLP model is as follows:

```
proc nlp tech=tr pall;
   array c[10] -6.089 -17.164 -34.054  -5.914 -24.721
               -14.986 -24.100 -10.708 -26.662 -22.179;
   array x[10] x1-x10;
   min y;
   parms x1-x10 = .1;
   bounds 1.e-6 <= x1-x10;
   lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
          1. = x4 + 2. * x5 + x6 + x7,
          1. = x3 + x7  + x8 + 2. * x9 + x10;
   s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
   y = 0.;
   do j = 1 to 10;
      y = y + x[j] * (c[j] + log(x[j] / s));
   end;
run;
```

The model finds an equilibrium state for a mixture of chemicals. The following statements show a corresponding PROC OPTMODEL model:

```
proc optmodel;
   set CMP = 1..10;
   number c{CMP} = [-6.089 -17.164 -34.054  -5.914 -24.721
                    -14.986 -24.100 -10.708 -26.662 -22.179];
   var x{CMP} init 0.1 >= 1.e-6;
   con 2. = x[1] + 2. * x[2] + 2. * x[3] + x[6] + x[10],
       1. = x[4] + 2. * x[5] + x[6] + x[7],
       1. = x[3] + x[7]  + x[8] + 2. * x[9] + x[10];
   /* replace the variable s in the NLP model */
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   solve;
   print x y;
```

The PROC OPTMODEL model uses the set CMP to represent the set of compounds, which are numbered 1 to 10 in the example. The array c was initialized by using the equivalent PROC OPTMODEL syntax. The individual array locations could also have been initialized by assignment or READ DATA statements.

The VAR declaration for variable x combines the VAR and BOUNDS statements of the PROC NLP model. The index set of the array is based on the set of compounds CMP, to simplify changes to the model.

The linear constraints are similar in form to the PROC NLP model. However, the PROC OPTMODEL

version uses the array form of the variable names because the OPTMODEL procedure treats arrays as distinct variables, not as aliases of lists of scalar variables.

The implicit variable s replaces the intermediate variable of the same name in the PROC NLP model. This is an example of translating an intermediate variable from the other models to PROC OPTMODEL. An alternative way is to use an additional constraint for every intermediate variable. In the preceding statements, instead of declaring objective s, you can use the following statements:

```
. . .
var s;
con s = sum{i in CMP} x[i];
. . .
```

Note that this alternative formulation passes an extra variable and constraint to the solver. This formulation can sometimes be solved more efficiently, depending on the characteristics of the model.

The PROC OPTMODEL version uses a SUM operator over the set CMP, which enhances the flexibility of the model to accommodate possible changes in the set of compounds.

In the PROC NLP model the objective function y is determined by an explicit loop. With PROC OPTMODEL, the DO loop is replaced by a SUM aggregation operation. The accumulation in the PROC NLP model is now performed by PROC OPTMODEL with the SUM operator.

This PROC OPTMODEL model can be further generalized. Note that the array initialization and constraints assume a fixed set of compounds. You can rewrite the model to handle an arbitrary number of compounds and chemical elements. The new model loads the linear constraint coefficients from a data set along with the objective coefficients for the parameter c, as follows:

```
data comp;
   input c a_1 a_2 a_3;
   datalines;
-6.089   1 0 0
-17.164  2 0 0
-34.054  2 0 1
-5.914   0 1 0
-24.721  0 2 0
-14.986  1 1 0
-24.100  0 1 1
-10.708  0 0 1
-26.662  0 0 2
-22.179  1 0 1
;

data atom;
   input b @@;
   datalines;
2. 1. 1.
;

proc optmodel;
   set CMP;
   set ELT;
   number c{CMP};
```

```
number a{ELT,CMP};
number b{ELT};
read data atom into ELT=[_n_] b;
read data comp into CMP=[_n_]
     c {i in ELT} < a[i,_n_]=col("a_"||i) >;
var x{CMP} init 0.1 >= 1.e-6;
con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
impvar s = sum{i in CMP} x[i];
min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
print a b;
solve;
print x;
```

This version adds coefficients for the linear constraints to the COMP data set. The data set variable a_*n* represents the number of atoms in the compound for element *n*. The READ DATA statement for COMP uses the iterated column syntax to read each of the data set variables a_*n* into the appropriate location in the array a. In this example the expanded data set variable names are a_1, a_2, and a_3.

The preceding version also adds a new set, ELT, of chemical elements and a numeric parameter, b, that represents the left-hand side of the linear constraints. The data values for the parameters ELT and b are read from the data set ATOM. The model can handle varying sets of chemical elements because of this extra data set and the new parameters.

The linear constraints have been converted to a single indexed family of constraints. One constraint is applied for each chemical element in the set ELT. The constraint expression uses a simple form that applies generally to linear constraints. The following PRINT statement in the model shows the values read from the data sets to define the linear constraints:

```
print a b;
```

The PRINT statements in the model produce the results shown in Output 7.11.

Figure 7.11 PROC OPTMODEL Output

The OPTMODEL Procedure

	a									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	2	2	0	0	1	0	0	0	1
2	0	0	0	1	2	1	1	0	0	0
3	0	0	1	0	0	0	1	1	2	1

	[1] b
1	2
2	1
3	1

Figure 7.11 *continued*

[1]	x
1	0.04066848
2	0.14773067
3	0.78315260
4	0.00141459
5	0.48524616
6	0.00069358
7	0.02739955
8	0.01794757
9	0.03731444
10	0.09687143

In the preceding model the chemical elements and compounds are designated by numbers. So in the PRINT output, for example, the row that is labeled "3" represents the amount of the compound H_2O. PROC OPTMODEL is capable of using more symbolic strings to designate array indices. The following version of the model uses strings to index arrays:

```
data comp;
   input name $ c a_h a_n a_o;
   datalines;
H     -6.089   1 0 0
H2    -17.164  2 0 0
H2O   -34.054  2 0 1
N     -5.914   0 1 0
N2    -24.721  0 2 0
NH    -14.986  1 1 0
NO    -24.100  0 1 1
O     -10.708  0 0 1
O2    -26.662  0 0 2
OH    -22.179  1 0 1
;
data atom;
   input name $ b;
   datalines;
H  2.
N  1.
O  1.
;
proc optmodel;
  set<string> CMP;
  set<string> ELT;
  number c{CMP};
  number a{ELT,CMP};
  number b{ELT};
  read data atom into ELT=[name] b;
  read data comp into CMP=[name]
       c {i in ELT} < a[i,name]=col("a_"||i) >;
  var x{CMP} init 0.1 >= 1.e-6;
  con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
  impvar s = sum{i in CMP} x[i];
```

```
min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
solve;
print x;
```

In this model, the sets CMP and ELT are now sets of strings. The data sets provide the names of the compounds and elements. The names of the data set variables for atom counts in the data set COMP now include the chemical element symbol as part of their spelling. For example, the atom count for element H (hydrogen) is named a_h. Note that these changes did not require any modification to the specifications of the linear constraints or the objective.

The PRINT statement in the preceding statements produces the results shown in Output 7.12. The indices of variable x are now strings that represent the actual compounds.

Figure 7.12 PROC OPTMODEL Output with Strings

The OPTMODEL Procedure

[1]	x
H	0.04066848
H2	0.14773067
H2O	0.78315260
N	0.00141459
N2	0.48524616
NH	0.00069358
NO	0.02739955
O	0.01794757
O2	0.03731444
OH	0.09687143

Examples: NLP Procedure

Example 7.1: Using the DATA= Option

This example illustrates the use of the DATA= option. The Bard function (refer to Moré, Garbow, and Hillstrom (1981)) is a least squares problem with $n = 3$ parameters and $m = 15$ functions f_k:

$$f(x) = \frac{1}{2}\sum_{k=1}^{15} f_k^2(x), \quad x = (x_1, x_2, x_3)$$

where

$$f_k(x) = y_k - \left(x_1 + \frac{u_k}{v_k x_2 + w_k x_3}\right)$$

with $u_k = k$, $v_k = 16 - k$, $w_k = \min(u_k, v_k)$, and

$$y = (.14, .18, .22, .25, .29, .32, .35, .39, .37, .58, .73, .96, 1.34, 2.10, 4.39)$$

The minimum function value $f(x^*) = 4.107\text{E}{-3}$ is at the point $(0.08, 1.13, 2.34)$. The starting point $x^0 = (1, 1, 1)$ is used. The following is the naive way of specifying the objective function.

```
proc nlp tech=levmar;
   lsq y1-y15;
   parms x1-x3 = 1;
   tmp1 = 15 * x2 + min(1,15) * x3;
   y1 = 0.14 - (x1 + 1 / tmp1);
   tmp1 = 14 * x2 + min(2,14) * x3;
   y2 = 0.18 - (x1 + 2 / tmp1);
   tmp1 = 13 * x2 + min(3,13) * x3;
   y3 = 0.22 - (x1 + 3 / tmp1);
   tmp1 = 12 * x2 + min(4,12) * x3;
   y4 = 0.25 - (x1 + 4 / tmp1);
   tmp1 = 11 * x2 + min(5,11) * x3;
   y5 = 0.29 - (x1 + 5 / tmp1);
   tmp1 = 10 * x2 + min(6,10) * x3;
   y6 = 0.32 - (x1 + 6 / tmp1);
   tmp1 = 9 * x2 + min(7,9) * x3;
   y7 = 0.35 - (x1 + 7 / tmp1);
   tmp1 = 8 * x2 + min(8,8) * x3;
   y8 = 0.39 - (x1 + 8 / tmp1);
   tmp1 = 7 * x2 + min(9,7) * x3;
   y9 = 0.37 - (x1 + 9 / tmp1);
   tmp1 = 6 * x2 + min(10,6) * x3;
   y10 = 0.58 - (x1 + 10 / tmp1);
   tmp1 = 5 * x2 + min(11,5) * x3;
   y11 = 0.73 - (x1 + 11 / tmp1);
   tmp1 = 4 * x2 + min(12,4) * x3;
   y12 = 0.96 - (x1 + 12 / tmp1);
   tmp1 = 3 * x2 + min(13,3) * x3;
   y13 = 1.34 - (x1 + 13 / tmp1);
   tmp1 = 2 * x2 + min(14,2) * x3;
   y14 = 2.10 - (x1 + 14 / tmp1);
   tmp1 = 1 * x2 + min(15,1) * x3;
   y15 = 4.39 - (x1 + 15 / tmp1);
run;
```

A more economical way to program this problem uses the DATA= option to input the 15 terms in $f(x)$.

```
data bard;
   input r @@;
      w1 = 16. - _n_;
      w2 = min(_n_ , 16. - _n_);
      datalines;
.14  .18  .22  .25  .29  .32  .35  .39
.37  .58  .73  .96 1.34 2.10 4.39
;

proc nlp data=bard tech=levmar;
   lsq y;
   parms x1-x3 = 1.;
```

```
   y = r - (x1 + _obs_ / (w1 * x2 + w2 * x3));
run;
```

Another way you can specify the objective function uses the ARRAY statement and an explicit do loop, as in the following code.

```
proc nlp tech=levmar;
   array r[15] .14  .18  .22  .25  .29  .32  .35  .39  .37  .58
               .73  .96 1.34 2.10 4.39 ;
   array y[15] y1-y15;
   lsq y1-y15;
   parms x1-x3 = 1.;
   do i = 1 to 15;
      w1 = 16. - i;
      w2 = min(i , w1);
      w3 = w1 * x2 + w2 * x3;
      y[i] = r[i] - (x1 + i / w3);
   end;
run;
```

Example 7.2: Using the INQUAD= Option

This example illustrates the INQUAD= option for specifying a quadratic programming problem:

$$\min \quad f(x) = \frac{1}{2}x^T G x + g^T x + c, \quad \text{with} \quad G^T = G$$

Suppose that $c = -100$, $G = \text{diag}(.4, 4)$ and $2 \leq x_1 \leq 50$, $-50 \leq x_2 \leq 50$, and $10 \leq 10x_1 - x_2$.

You specify the constant c and the Hessian G in the data set QUAD1. Notice that the _TYPE_ variable contains the keywords that identify how the procedure should interpret the observations.

```
data quad1;
   input _type_ $ _name_ $ x1 x2;
   datalines;
const    .   -100  -100
quad    x1    0.4     0
quad    x2      0     4
;
```

You specify the QUAD1 data set with the INQUAD= option. Notice that the names of the variables in the QUAD1 data set and the _NAME_ variable match the names of the parameters in the PARMS statement.

```
proc nlp inquad=quad1 all;
   min ;
   parms x1 x2 = -1;
   bounds  2 <= x1 <= 50,
         -50 <= x2 <= 50;
   lincon 10 <= 10 * x1 - x2;
run;
```

Alternatively, you can use a sparse format for specifying the G matrix, eliminating the zeros. You use the special variables _ROW_, _COL_, and _VALUE_ to give the nonzero row and column names and value.

```
data quad2;
   input _type_ $ _row_ $ _col_ $ _value_;
   datalines;
const   .      .     -100
quad    x1     x1     0.4
quad    x2     x2       4
;
```

You can also include the constraints in the QUAD data set. Notice how the _TYPE_ variable contains keywords that identify how the procedure is to interpret the values in each observation.

```
data quad3;
   input _type_ $ _name_ $ x1    x2    _rhs_;
   datalines;
const       .     -100   -100    .
quad        x1    0.02      0    .
quad        x2    0.00      2    .
parms       .       -1     -1    .
lowerbd     .        2    -50    .
upperbd     .       50     50    .
ge          .       10     -1   10
;

proc nlp inquad=quad3;
   min ;
   parms x1 x2;
run;
```

Example 7.3: Using the INEST=Option

This example illustrates the use of the INEST= option for specifying a starting point and linear constraints. You name a data set with the INEST= option. The format of this data set is similar to the format of the QUAD data set described in the previous example.

Consider the Hock and Schittkowski (1981) Problem # 24:

$$\min f(x) = \frac{((x_1 - 3)^2 - 9)x_2^3}{27\sqrt{3}}$$

subject to:

$$\begin{aligned} 0 &\leq x_1, x_2 \\ 0 &\leq .57735x_1 - x_2 \\ 0 &\leq x_1 + 1.732x_2 \\ 6 &\geq x_1 + 1.732x_2 \end{aligned}$$

with minimum function value $f(x^*) = -1$ at $x^* = (3, \sqrt{3})$. The feasible starting point is $x^0 = (1, .5)$.

You can specify this model in PROC NLP as follows:

```
proc nlp tech=trureg outest=res;
   min y;
   parms x1 = 1,
         x2 = .5;
   bounds  0 <= x1-x2;
   lincon  .57735 * x1 -         x2 >=  0,
                   x1 + 1.732 * x2 >=  0,
                  -x1 - 1.732 * x2 >= -6;
   y = (((x1 - 3)**2 - 9.) * x2**3) / (27 * sqrt(3));
run;
```

Note that none of the data for this model are in a data set. Alternatively, you can save the starting point (1, .5) and the linear constraints in a data set. Notice that the _TYPE_ variable contains keywords that identify how the procedure is to interpret each of the observations and that the parameters in the problems X1 and X2 are variables in the data set. The observation with _TYPE_=LOWERBD gives the lower bounds on the parameters. The observation with _TYPE_=GE gives the coefficients for the first constraint. Similarly, the subsequent observations contain specifications for the other constraints. Also notice that the special variable _RHS_ contains the right-hand-side values.

```
data betts1(type=est);
   input _type_ $ x1 x2 _rhs_;
   datalines;
parms    1        .5       .
lowerbd  0        0        .
ge       .57735   -1       .
ge       1        1.732    .
le       1        1.732    6
;
```

Now you can solve this problem with the following code. Notice that you specify the objective function and the parameters.

```
proc nlp inest=betts1 tech=trureg;
   min y;
   parms x1 x2;
   y = (((x1 - 3)**2 - 9) * x2**3) / (27 * sqrt(3));
run;
```

You can even include any constants used in the program statements in the INEST= data set. In the following code the variables A, B, C, and D contain some of the constants used in calculating the objective function Y.

```
data betts2(type=est);
   input _type_ $ x1    x2   _rhs_    a   b   c   d;
   datalines;
parms    1        .5       .     3   9  27   3
lowerbd  0        0        .     .   .   .   .
ge       .57735   -1       0     .   .   .   .
ge       1        1.732    0     .   .   .   .
le       1        1.732    6     .   .   .   .
;
```

Notice that in the program statement for calculating Y, the constants are replaced by the A, B, C, and D variables.

```
proc nlp inest=betts2 tech=trureg;
   min y;
   parms x1 x2;
   y = (((x1 - a)**2 - b) * x2**3) / (c * sqrt(d));
run;
```

Example 7.4: Restarting an Optimization

This example shows how you can restart an optimization problem using the OUTEST=, INEST=, OUTMODEL=, and MODEL= options and how to save output into an OUT= data set. The least squares solution of the Rosenbrock function using the trust region method is used.

The following code solves the problem and saves the model in the MODEL data set and the solution in the EST and OUT1 data sets.

```
proc nlp tech=trureg outmodel=model outest=est out=out1;
   lsq y1 y2;
   parms x1 = -1.2 ,
         x2 =  1.;
   y1 = 10. * (x2 - x1 * x1);
   y2 = 1. - x1;
run;

proc print data=out1;
run;
```

The final parameter estimates $x^* = (1, 1)$ and the values of the functions f_1 =Y1 and f_2 =Y2 are written into an OUT= data set, shown in Output 7.4.1. Since OUTDER=0 is the default, the OUT= data set does not contain the Jacobian matrix.

Output 7.4.1 Solution in an OUT= Data Set

Obs	_OBS_	_TYPE_	y1	y2	x2	x1
1	1	0	3.3307E-16	1	1	

Next, the procedure reads the optimal parameter estimates from the EST data set and the model from the MODEL data set. It does not do any optimization (TECH=NONE), but it saves the Jacobian matrix to the OUT=OUT2 data set because of the option OUTDER=1. It also displays the Jacobian matrix because of the option PJAC; the Jacobian matrix is shown in Output 7.4.2. Output 7.4.3 shows the contents of the OUT2 data set, which also contains the Jacobian matrix.

```
proc nlp tech=none model=model inest=est out=out2 outder=1 pjac PHISTORY;
   lsq y1 y2;
   parms x1 x2;
run;

proc print data=out2;
run;
```

Output 7.4.2 Jacobian Matrix Output

PROC NLP: Least Squares Minimization

Jacobian Matrix	
x1	**x2**
-20	10
-1	0

Output 7.4.3 Jacobian Matrix in an OUT= Data Set

Obs	_OBS_	_TYPE_	y1	y2	_WRT_	x2	x1
1	1		0	0		1	1
2	1	ANALYTIC	10	0	x2	1	1
3	1	ANALYTIC	-20	-1	x1	1	1

Example 7.5: Approximate Standard Errors

The NLP procedure provides a variety of ways for estimating parameters in nonlinear statistical models and for obtaining approximate standard errors and covariance matrices for the estimators. These methods are illustrated by estimating the mean of a random sample from a normal distribution with mean μ and standard deviation σ. The simplicity of the example makes it easy to compare the results of different methods in NLP with the usual estimator, the sample mean.

The following data step is used:

```
data x;
   input x @@;
datalines;
1 3 4 5 7
;
```

The standard error of the mean, computed with $n-1$ degrees of freedom, is 1. The usual maximum-likelihood approximation to the standard error of the mean, using a variance divisor of n rather than $n-1$, is 0.894427.

The sample mean is a least squares estimator, so it can be computed using an LSQ statement. Moreover, since this model is linear, the Hessian matrix and crossproduct Jacobian matrix are identical, and all three versions of the COV= option yield the same variance and standard error of the mean. Note that COV=j means that the crossproduct Jacobian is used. This is chosen because it requires the least computation.

```
proc nlp data=x cov=j pstderr pshort PHISTORY;
   lsq resid;
   parms mean=0;
   resid=x-mean;
run;
```

The results are the same as the usual estimates.

Output 7.5.1 Parameter Estimates

PROC NLP: Least Squares Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	1.000000	4.000000	0.016130	0

Value of Objective Function = 10

PROC NLP can also compute maximum-likelihood estimates of μ and σ. In this case it is convenient to minimize the negative log likelihood. To get correct standard errors for maximum-likelihood estimators, the SIGSQ=1 option is required. The following program shows COV=1 but the output that follows has COV=2 and COV=3.

```
proc nlp data=x cov=1 sigsq=1 pstderr phes pcov pshort;
      min nloglik;
   parms mean=0, sigma=1;
   bounds 1e-12 < sigma;
   nloglik=.5*((x-mean)/sigma)**2 + log(sigma);
run;
```

The variance divisor is n instead of $n - 1$, so the standard error of the mean is 0.894427 instead of 1. The standard error of the mean is the same with all six types of covariance matrix, but the standard error of the standard deviation varies. The sampling distribution of the standard deviation depends on the higher moments of the population distribution, so different methods of estimation can produce markedly different estimates of the standard error of the standard deviation.

Output 7.5.2 shows the output when COV=1, Output 7.5.3 shows the output when COV=2, and Output 7.5.4 shows the output when COV=3.

Output 7.5.2 Solution for COV=1

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	1.33149E-10
2	sigma	2.000000	0.458258	4.364358	0.007260	-5.606415E-9

Value of Objective Function = 5.9657359028

Hessian Matrix

	mean	sigma
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 1: M = (NOBS/d) inv(G) JJ(f) inv(G)

	mean	sigma
mean	0.8	1.980107E-11
sigma	1.980107E-11	0.2099999991

Factor sigm = 1

Determinant = 0.1679999993

Matrix has Only Positive Eigenvalues

Output 7.5.3 Solution for COV=2

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	1.33149E-10
2	sigma	2.000000	0.632456	3.162278	0.025031	-5.606415E-9

Output 7.5.3 *continued*

Value of Objective Function = 5.9657359028

Hessian Matrix		
	mean	**sigma**
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	mean	**sigma**
mean	0.7999999982	4.260766E-11
sigma	4.260766E-11	0.3999999978

Factor sigm = 1

Determinant = 0.3199999975

Matrix has Only Positive Eigenvalues

Output 7.5.4 Solution for COV=3

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	**mean**	4.000000	0.509136	7.856442	0.000537	1.301733E-10
2	**sigma**	2.000000	0.419936	4.762634	0.005048	-5.940302E-9

Value of Objective Function = 5.9657359028

Hessian Matrix		
	mean	**sigma**
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 3: J = (1/d) inv(W)		
	mean	**sigma**
mean	0.2592197879	1.062283E-11
sigma	1.062283E-11	0.1763460041

Output 7.5.4 *continued*

Factor sigm = 0.2

Determinant = 0.0457123738

Matrix has Only Positive Eigenvalues

Under normality, the maximum-likelihood estimators of μ and σ are independent, as indicated by the diagonal Hessian matrix in the previous example. Hence, the maximum-likelihood estimate of μ can be obtained by using any fixed value for σ, such as 1. However, if the fixed value of σ differs from the actual maximum-likelihood estimate (in this case 2), the model is misspecified and the standard errors obtained with COV=2 or COV=3 are incorrect. It is therefore necessary to use COV=1, which yields consistent estimates of the standard errors under a variety of forms of misspecification of the error distribution.

```
proc nlp data=x cov=1 sigsq=1 pstderr pcov pshort;
   min sqresid;
   parms mean=0;
   sqresid=.5*(x-mean)**2;
run;
```

This formulation produces the same standard error of the mean, 0.894427 (see Output 7.5.5).

Output 7.5.5 Solution for Fixed σ and COV=1

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	0

Value of Objective Function = 10

Covariance Matrix 1: M = (NOBS/d) inv(G) JJ(f) inv(G)	mean
mean	0.8

Factor sigm = 1

The maximum-likelihood formulation with fixed σ is actually a least squares problem. The objective function, parameter estimates, and Hessian matrix are the same as those in the first example in this section using the LSQ statement. However, the Jacobian matrix is different, each row being multiplied by twice the residual. To treat this formulation as a least squares problem, the SIGSQ=1 option can be omitted. But since the Jacobian is not the same as in the formulation using the LSQ statement, the COV=1 | M and COV=3 | J options, which use the Jacobian, do not yield correct standard errors. The correct standard error is obtained with COV=2 | H, which uses only the Hessian matrix:

```
proc nlp data=x cov=2 pstderr pcov pshort;
   min sqresid;
   parms mean=0;
   sqresid=.5*(x-mean)**2;
run;
```

The results are the same as in the first example.

Output 7.5.6 Solution for Fixed σ and COV=2

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	mean	4.000000	0.500000	8.000000	0.001324	0

Value of Objective Function = 10

Covariance Matrix 2: H = (NOBS/d) inv(G)	mean
mean	0.25

Factor sigm = 1.25

In summary, to obtain appropriate standard errors for least squares estimates, you can use the LSQ statement with any of the COV= options, or you can use the MIN statement with COV=2. To obtain appropriate standard errors for maximum-likelihood estimates, you can use the MIN statement with the negative log likelihood or the MAX statement with the log likelihood, and in either case you can use any of the COV= options provided that you specify SIGSQ=1. You can also use a log-likelihood function with a misspecified scale parameter provided that you use SIGSQ=1 and COV=1. For nonlinear models, all of these methods yield approximations based on asymptotic theory, and should therefore be interpreted cautiously.

Example 7.6: Maximum Likelihood Weibull Estimation

Two-Parameter Weibull Estimation

The following data are taken from Lawless (1982, p. 193) and represent the number of days it took rats painted with a carcinogen to develop carcinoma. The last two observations are censored data from a group of 19 rats:

```
data pike;
   input days cens @@;
   datalines;
143  0  164  0  188  0  188  0
190  0  192  0  206  0  209  0
213  0  216  0  220  0  227  0
230  0  234  0  246  0  265  0
304  0  216  1  244  1
;
```

Suppose that you want to show how to compute the maximum likelihood estimates of the scale parameter σ (α in Lawless), the shape parameter c (β in Lawless), and the location parameter θ (μ in Lawless). The observed likelihood function of the three-parameter Weibull transformation (Lawless 1982, p. 191) is

$$L(\theta,\sigma,c) = \frac{c^m}{\sigma^m} \prod_{i \in D} \left(\frac{t_i - \theta}{\sigma} \right)^{c-1} \prod_{i=1}^{p} \exp\left(- \left(\frac{t_i - \theta}{\sigma} \right)^c \right)$$

and the log likelihood is

$$l(\theta,\sigma,c) = m \log c - mc \log \sigma + (c-1) \sum_{i \in D} \log(t_i - \theta) - \sum_{i=1}^{p} \left(\frac{t_i - \theta}{\sigma} \right)^c$$

The log likelihood function can be evaluated only for $\sigma > 0$, $c > 0$, and $\theta < \min_i t_i$. In the estimation process, you must enforce these conditions using lower and upper boundary constraints. The three-parameter Weibull estimation can be numerically difficult, and it usually pays off to provide good initial estimates. Therefore, you first estimate σ and c of the two-parameter Weibull distribution for constant $\theta = 0$. You then use the optimal parameters $\hat{\sigma}$ and $\hat{c}$ as starting values for the three-parameter Weibull estimation.

Although the use of an INEST= data set is not really necessary for this simple example, it illustrates how it is used to specify starting values and lower boundary constraints:

```
data par1(type=est);
   keep _type_ sig c theta;
   _type_='parms'; sig = .5;
      c = .5; theta = 0; output;
   _type_='lb'; sig = 1.0e-6;
      c = 1.0e-6; theta = .;  output;
run;
```

The following PROC NLP call specifies the maximization of the log likelihood function for the two-parameter Weibull estimation for constant $\theta = 0$:

```
proc nlp data=pike tech=tr inest=par1 outest=opar1
     outmodel=model cov=2 vardef=n pcov phes;
           max logf;
   parms sig c;
   profile sig c / alpha = .9 to .1 by -.1 .09 to .01 by -.01;

   x_th = days - theta;
   s    = - (x_th / sig)**c;
   if cens=0 then s + log(c) - c*log(sig) + (c-1)*log(x_th);
   logf = s;
run;
```

After a few iterations you obtain the solution given in Output 7.6.1.

Output 7.6.1 Optimization Results

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	sig	234.318611	9.645908	24.292021	9.050475E-16	1.3372183E-9
2	c	6.083147	1.068229	5.694611	0.000017269	-7.859278E-9

Value of Objective Function = -88.23273515

Since the gradient has only small elements and the Hessian (shown in Output 7.6.2) is negative definite (has only negative eigenvalues), the solution defines an isolated maximum point.

Output 7.6.2 Hessian Matrix at x^*

PROC NLP: Nonlinear Maximization

Hessian Matrix		
	sig	**c**
sig	-0.011457556	0.0257527577
c	0.0257527577	-0.934221388

Determinant = 0.0100406894

Matrix has Only Negative Eigenvalues

The square roots of the diagonal elements of the approximate covariance matrix of parameter estimates are the approximate standard errors (ASE's). The covariance matrix is given in Output 7.6.3.

Output 7.6.3 Covariance Matrix

PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	sig	**c**
sig	93.043549863	2.5648395794
c	2.5648395794	1.141112488

Factor sigm = 1

Determinant = 99.594754608

Matrix has 2 Positive Eigenvalue(s)

The confidence limits in Output 7.6.4 correspond to the α values in the PROFILE statement.

Output 7.6.4 Confidence Limits

PROC NLP: Nonlinear Maximization

Wald and PL Confidence Limits							
				Profile Likelihood Confidence Limits		Wald Confidence Limits	
N	Parameter	Estimate	Alpha				
1	sig	234.318611	0.900000	233.111324	235.532695	233.106494	235.530729
1	sig	.	0.800000	231.886549	236.772876	231.874849	236.762374
1	sig	.	0.700000	230.623280	238.063824	230.601846	238.035377
1	sig	.	0.600000	229.292797	239.436639	229.260292	239.376931
1	sig	.	0.500000	227.855829	240.935290	227.812545	240.824678
1	sig	.	0.400000	226.251597	242.629201	226.200410	242.436813
1	sig	.	0.300000	224.372260	244.643392	224.321270	244.315953
1	sig	.	0.200000	221.984557	247.278423	221.956882	246.680341
1	sig	.	0.100000	218.390824	251.394102	218.452504	250.184719
1	sig	.	0.090000	217.884162	251.987489	217.964960	250.672263
1	sig	.	0.080000	217.326988	252.645278	217.431654	251.205569
1	sig	.	0.070000	216.708814	253.383546	216.841087	251.796136
1	sig	.	0.060000	216.008815	254.228034	216.176649	252.460574
1	sig	.	0.050000	215.199301	255.215496	215.412978	253.224245
1	sig	.	0.040000	214.230116	256.411041	214.508337	254.128885
1	sig	.	0.030000	213.020874	257.935686	213.386118	255.251105
1	sig	.	0.020000	211.369067	260.066128	211.878873	256.758350
1	sig	.	0.010000	208.671091	263.687174	209.472398	259.164825
2	c	6.083147	0.900000	5.950029	6.217752	5.948912	6.217382
2	c	.	0.800000	5.815559	6.355576	5.812514	6.353780
2	c	.	0.700000	5.677909	6.499187	5.671537	6.494757
2	c	.	0.600000	5.534275	6.651789	5.522967	6.643327
2	c	.	0.500000	5.380952	6.817880	5.362638	6.803656
2	c	.	0.400000	5.212344	7.004485	5.184103	6.982191
2	c	.	0.300000	5.018784	7.225733	4.975999	7.190295
2	c	.	0.200000	4.776379	7.506166	4.714157	7.452137
2	c	.	0.100000	4.431310	7.931669	4.326067	7.840227
2	c	.	0.090000	4.382687	7.991457	4.272075	7.894220
2	c	.	0.080000	4.327815	8.056628	4.213014	7.953280
2	c	.	0.070000	4.270773	8.129238	4.147612	8.018682
2	c	.	0.060000	4.207130	8.211221	4.074029	8.092265
2	c	.	0.050000	4.134675	8.306218	3.989457	8.176837
2	c	.	0.040000	4.049531	8.418782	3.889274	8.277021
2	c	.	0.030000	3.945037	8.559677	3.764994	8.401300
2	c	.	0.020000	3.805759	8.749130	3.598076	8.568219
2	c	.	0.010000	3.588814	9.056751	3.331572	8.834722

Three-Parameter Weibull Estimation

You now prepare for the three-parameter Weibull estimation by using PROC UNIVARIATE to obtain the smallest data value for the upper boundary constraint for θ. For this small problem, you can do this much more simply by just using a value slightly smaller than the minimum data value 143.

```
/* Calculate upper bound for theta parameter */
proc univariate data=pike noprint;
   var days;
   output out=stats n=nobs min=minx range=range;
run;

data stats;
   set stats;
   keep _type_ theta;

   /* 1. write parms observation */
   theta = minx - .1 * range;
   if theta < 0 then theta = 0;
   _type_ = 'parms';
   output;

   /* 2. write ub observation */
   theta = minx * (1 - 1e-4);
   _type_ = 'ub';
   output;
run;
```

The data set PAR2 specifies the starting values and the lower and upper bounds for the three-parameter Weibull problem:

```
proc sort data=opar1;
   by _type_;
run;

data par2(type=est);
   merge opar1(drop=theta) stats;
   by _type_;
   keep _type_ sig c theta;
   if _type_ in ('parms' 'lowerbd' 'ub');
run;
```

The following PROC NLP call uses the MODEL= input data set containing the log likelihood function that was saved during the two-parameter Weibull estimation:

```
proc nlp data=pike tech=tr inest=par2 outest=opar2
      model=model cov=2 vardef=n pcov phes;
   max logf;
   parms sig c theta;
   profile sig c theta / alpha = .5 .1 .05 .01;
run;
```

After a few iterations, you obtain the solution given in Output 7.6.5.

Output 7.6.5 Optimization Results

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > |t|	Gradient Objective Function
1	sig	108.382632	32.573219	3.327354	0.003540	-7.403602E-9
2	c	2.711474	1.058755	2.561003	0.019108	-0.000001148
3	theta	122.026036	28.692260	4.252925	0.000430	-0.000000160

Value of Objective Function = -87.32424712

From inspecting the first- and second-order derivatives at the optimal solution, you can verify that you have obtained an isolated maximum point. The Hessian matrix is shown in Output 7.6.6.

Output 7.6.6 Hessian Matrix

PROC NLP: Nonlinear Maximization

Hessian Matrix

	sig	c	theta
sig	-0.010639974	0.0453887849	-0.010033749
c	0.0453887849	-4.078687944	-0.083026333
theta	-0.010033749	-0.083026333	-0.014752091

Determinant = 0.0000502116

Matrix has Only Negative Eigenvalues

The square roots of the diagonal elements of the approximate covariance matrix of parameter estimates are the approximate standard errors. The covariance matrix is given in Output 7.6.7.

Output 7.6.7 Covariance Matrix

PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)

	sig	c	theta
sig	1061.025982	29.92625548	-890.0932211
c	29.92625548	1.1209709237	-26.66351895
theta	-890.0932211	-26.66351895	823.25594666

Factor sigm = 1

Determinant = 19915.719564

Matrix has 3 Positive Eigenvalue(s)

The difference between the Wald and profile CLs for parameter PHI2 are remarkable, especially for the upper 95% and 99% limits, as shown in Output 7.6.8.

Output 7.6.8 Confidence Limits

PROC NLP: Nonlinear Maximization

Wald and PL Confidence Limits

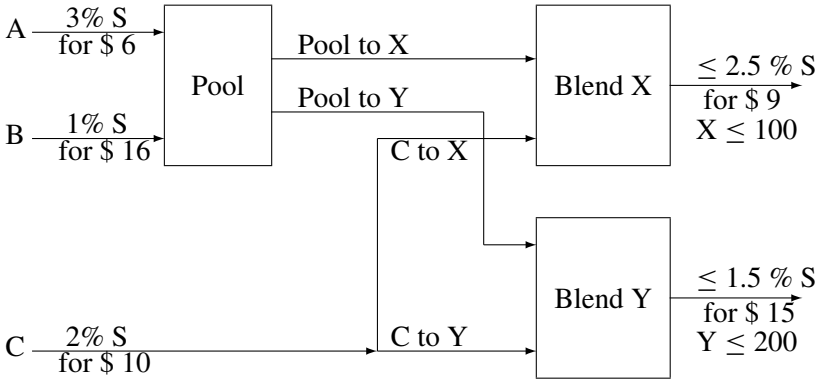
N	Parameter	Estimate	Alpha	Profile Likelihood Confidence Limits		Wald Confidence Limits	
1	sig	108.381969	0.500000	91.811550	141.564599	86.412463	130.351475
1	sig	.	0.100000	76.502367	.	54.805733	161.958205
1	sig	.	0.050000	72.215888	.	44.541946	172.221992
1	sig	.	0.010000	64.262383	.	24.481957	192.281981
2	c	2.711456	0.500000	2.139297	3.704051	1.997356	3.425555
2	c	.	0.100000	1.574162	9.250080	0.970007	4.452904
2	c	.	0.050000	1.424853	19.536314	0.636392	4.786520
2	c	.	0.010000	1.163097	19.560763	-0.015640	5.438552
3	theta	122.026662	0.500000	91.027154	135.095457	102.674832	141.378492
3	theta	.	0.100000	.	141.833768	74.834057	142.985700
3	theta	.	0.050000	.	142.512603	65.793205	142.985700
3	theta	.	0.010000	.	142.967407	48.123372	142.985700

Example 7.7: Simple Pooling Problem

The following optimization problem is discussed in Haverly (1978) and in Liebman et al. (1986, pp. 127–128). Two liquid chemicals, *X* and *Y*, are produced by the pooling and blending of three input liquid chemicals, *A*, *B*, and *C*. You know the sulfur impurity amounts of the input chemicals, and you have to respect upper limits of the sulfur impurity amounts of the output chemicals. The sulfur concentrations and the prices of the input and output chemicals are:

- Chemical *A*: Concentration = 3%, Price= \$6
- Chemical *B*: Concentration = 1%, Price= \$16
- Chemical *C*: Concentration = 2%, Price= \$10
- Chemical *X*: Concentration $\leq$ 2.5%, Price= \$9
- Chemical *Y*: Concentration $\leq$ 1.5%, Price= \$15

The problem is complicated by the fact that the two input chemicals *A* and *B* are available only as a mixture (they are either shipped together or stored together). Because the amounts of *A* and *B* are unknown, the sulfur concentration of the mixture is also unknown.



You know customers will buy no more than 100 units of X and 200 units of Y. The problem is determining how to operate the pooling and blending of the chemicals to maximize the profit. The objective function for the profit is

$$\begin{aligned} \mathit{profit} &= \mathit{cost}(x) \times \mathit{amount}(x) + \mathit{cost}(y) \times \mathit{amount}(y) \\ &- \mathit{cost}(a) \times \mathit{amount}(a) - \mathit{cost}(b) \times \mathit{amount}(b) - \mathit{cost}(c) \times \mathit{amount}(c) \end{aligned}$$

There are three groups of constraints:

1. The first group of constraint functions is the mass balance restrictions illustrated by the graph. These are four linear equality constraints:
   - $amount(a) + amount(b) = pool_to_x + pool_to_y$
   - $pool_to_x + c_to_x = amount(x)$
   - $pool_to_y + c_to_y = amount(y)$
   - $amount(c) = c_to_x + c_to_y$
2. You introduce a new variable, $pool_s$, that represents the sulfur concentration of the pool. Using $pool_s$ and the sulfur concentration of *C* (2%), you obtain two nonlinear inequality constraints for the sulfur concentrations of *X* and *Y*, one linear equality constraint for the sulfur balance, and lower and upper boundary restrictions for $pool_s$:
   - $pool_s \times pool_to_x + 2\, c_to_x \leq 2.5\, amount(x)$
   - $pool_s \times pool_to_y + 2\, c_to_y \leq 1.5\, amount(y)$
   - $3\, amount(a) + 1\, amount(b) = pool_s \times (amount(a) + amount(b))$
   - $1 \leq pool_s \leq 3$

3. The last group assembles the remaining boundary constraints. First, you do not want to produce more than you can sell; and finally, all variables must be nonnegative:

   - $amount(x) \leq 100, \quad amount(y) \leq 200$
   - $amount(a), amount(b), amount(c), amount(x), amount(y) \geq 0$
   - $pool_to_x, pool_to_y, c_to_x, c_to_y \geq 0$

There exist several local optima to this problem that can be found by specifying different starting points. Using the starting point with all variables equal to 1 (specified with a PARMS statement), PROC NLP finds a solution with $profit = 400$:

```
proc nlp all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools = 1;
               bounds 0 <= amountx amounty amounta amountb amountc,
               amountx <= 100,
               amounty <= 200,
          0 <= pooltox pooltoy ctox ctoy,
          1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
          pooltox + ctox = amountx,
          pooltoy + ctoy = amounty,
          ctox + ctoy    = amountc;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

The specified starting point was not feasible with respect to the linear equality constraints; therefore, a starting point is generated that satisfies linear and boundary constraints. Output 7.7.1 gives the starting parameter estimates.

Output 7.7.1 Starting Estimates

PROC NLP: Nonlinear Maximization

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Lower Bound Constraint	Upper Bound Constraint
1	amountx	1.363636	9.000000	-0.843698	0	100.000000
2	amounty	1.363636	15.000000	-0.111882	0	200.000000
3	amounta	0.818182	-6.000000	-0.430733	0	.
4	amountb	0.818182	-16.000000	-0.542615	0	.
5	amountc	1.090909	-10.000000	0.017768	0	.
6	pooltox	0.818182	0	-0.669628	0	.
7	pooltoy	0.818182	0	-0.303720	0	.
8	ctox	0.545455	0	-0.174070	0	.
9	ctoy	0.545455	0	0.191838	0	.
10	pools	2.000000	0	0.068372	1.000000	3.000000

Value of Objective Function = 3.8181818182

Value of Lagrange Function = -2.866739915

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Active Bound Constraint	
1	amountx	-1.40474E-11	9.000000	0	Lower BC	
2	amounty	200.000000	15.000000	-8.88178E-16	Upper BC	
3	amounta	5.561161E-16	-6.000000	0	Lower BC	
4	amountb	100.000000	-16.000000	1.065814E-14		
5	amountc	100.000000	-10.000000	-1.77636E-15		
6	pooltox	7.024225E-12	0	0	Lower BC	
7	pooltoy	100.000000	0	1.776357E-15		
8	ctox	-2.10716E-11	0	1.776357E-15	Lower BC	LinDep
9	ctoy	100.000000	0	5.329071E-15		
10	pools	1.000000	0	4.973799E-14	Lower BC	LinDep

The starting point satisfies the four equality constraints, as shown in Output 7.7.2. The nonlinear constraints are given in Output 7.7.3.

Output 7.7.2 Linear Constraints

PROC NLP: Nonlinear Maximization

Linear Constraints

1	2.2204E-16	:	ACT	0	==	+	1.0000	*	amounta	+	1.0000	*	amountb	- 1.0000 * pooltox - 1.0000 * pooltoy
2	0	:	ACT	0	==	-	1.0000	*	amountx	+	1.0000	*	pooltox	+ 1.0000 * ctox
3	-1.11E-16	:	ACT	0	==	-	1.0000	*	amounty	+	1.0000	*	pooltoy	+ 1.0000 * ctoy
4	-1.11E-16	:	ACT	0	==	-	1.0000	*	amountc	+	1.0000	*	ctox	+ 1.0000 * ctoy

Output 7.7.3 Nonlinear Constraints

PROC NLP: Nonlinear Maximization

Values of Nonlinear Constraints							
	Constraint		Value	Residual	Lagrange Multiplier		
[5]	nlc3		0	0	4.9441	Active	NLEC
[6]	nlc1_G		0.6818	0.6818	.		
[7]	nlc2_G		-0.6818	-0.6818	-9.8046	Violat.	NLIC

PROC NLP: Nonlinear Maximization

Values of Nonlinear Constraints								
	Constraint		Value	Residual	Lagrange Multiplier			
[5]	nlc3		1.11E-15	1.11E-15	6.0000	Active	NLEC	
[6]	nlc1_G		4.31E-16	4.31E-16	.	Active	NLIC	LinDep
[7]	nlc2_G		0	0	-6.0000	Active	NLIC	

Output 7.7.4 shows the settings of some important PROC NLP options.

Output 7.7.4 Options

PROC NLP: Nonlinear Maximization

Minimum Iterations	0
Maximum Iterations	200
Maximum Function Calls	500
Iterations Reducing Constraint Violation	20
ABSGCONV Gradient Criterion	0.00001
GCONV Gradient Criterion	1E-8
ABSFCONV Function Criterion	0
FCONV Function Criterion	2.220446E-16
FCONV2 Function Criterion	1E-6
FSIZE Parameter	0
ABSXCONV Parameter Change Criterion	0
XCONV Parameter Change Criterion	0
XSIZE Parameter	0
ABSCONV Function Criterion	1.340781E154
Line Search Method	2
Starting Alpha for Line Search	1
Line Search Precision LSPRECISION	0.4
DAMPSTEP Parameter for Line Search	.
FD Derivatives: Accurate Digits in Obj.F	15.653559775
FD Derivatives: Accurate Digits in NLCon	15.653559775
Singularity Tolerance (SINGULAR)	1E-8
Constraint Precision (LCEPS)	1E-8
Linearly Dependent Constraints (LCSING)	1E-8
Releasing Active Constraints (LCDEACT)	.

The iteration history, given in Output 7.7.5, does not show any problems.

Output 7.7.5 Iteration History

PROC NLP: Nonlinear Maximization

Dual Quasi-Newton Optimization

Modified VMCWD Algorithm of Powell (1978, 1982)

Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

Lagrange Multiplier Update of Powell(1982)

Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Maximum Gradient Element of the Lagrange Function
1	0	19	-1.42400	0.00962	6.9131	1.000	0.783
2 '	0	20	2.77026	0.0166	5.3770	1.000	2.629
3	0	21	7.08706	0.1409	7.1965	1.000	9.452
4 '	0	22	11.41264	0.0583	15.5769	1.000	23.390
5 '	0	23	24.84607	1.78E-15	496.1	1.000	147.6
6	0	24	378.22829	147.4	3316.8	1.000	840.4
7 '	0	25	307.56787	50.9338	607.9	1.000	27.143
8 '	0	26	347.24475	1.8328	21.9882	1.000	28.482
9 '	0	27	349.49273	0.00915	7.1826	1.000	28.289
10 '	0	28	356.58291	0.1083	50.2554	1.000	27.479
11 '	0	29	388.70709	2.4280	24.7997	1.000	21.114
12 '	0	30	389.30094	0.0157	10.0473	1.000	18.647
13 '	0	31	399.19199	0.7996	11.1866	1.000	0.416
14 '	0	32	400.00000	0.0128	0.1534	1.000	0.00087
15 '	0	33	400.00000	7.38E-11	2.43E-10	1.000	365E-12

Optimization Results			
Iterations	15	**Function Calls**	34
Gradient Calls	18	**Active Constraints**	10
Objective Function	400	**Maximum Constraint Violation**	7.381118E-11
Maximum Projected Gradient	0	**Value Lagrange Function**	-400
Maximum Gradient of the Lagran Func	4.973799E-14	**Slope of Search Direction**	-2.43334E-10

FCONV2 convergence criterion satisfied.

The optimal solution in Output 7.7.6 shows that to obtain the maximum profit of \$400, you need only to produce the maximum 200 units of blending Y and no units of blending X.

Output 7.7.6 Optimization Solution

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Active Bound Constraint	
1	amountx	-1.40474E-11	9.000000	0	Lower BC	
2	amounty	200.000000	15.000000	-8.88178E-16	Upper BC	
3	amounta	5.561161E-16	-6.000000	0	Lower BC	
4	amountb	100.000000	-16.000000	1.065814E-14		
5	amountc	100.000000	-10.000000	-1.77636E-15		
6	pooltox	7.024225E-12	0	0	Lower BC	
7	pooltoy	100.000000	0	1.776357E-15		
8	ctox	-2.10716E-11	0	1.776357E-15	Lower BC	LinDep
9	ctoy	100.000000	0	5.329071E-15		
10	pools	1.000000	0	4.973799E-14	Lower BC	LinDep

Value of Objective Function = 400

Value of Lagrange Function = 400

The constraints are satisfied at the solution, as shown in Output 7.7.7

Output 7.7.7 Linear and Nonlinear Constraints at the Solution

PROC NLP: Nonlinear Maximization

Linear Constraints Evaluated at Solution

1	ACT	0	=	0	+	1.0000	*	amounta	+	1.0000	*	amountb	-	1.0000	*	pooltox	- 1.0000 * pooltoy
2	ACT	3.8975E-17	=	0	-	1.0000	*	amountx	+	1.0000	*	pooltox	+	1.0000	*	ctox	
3	ACT	0	=	0	-	1.0000	*	amounty	+	1.0000	*	pooltoy	+	1.0000	*	ctoy	
4	ACT	0	=	0	-	1.0000	*	amountc	+	1.0000	*	ctox	+	1.0000	*	ctoy	

Values of Nonlinear Constraints

Constraint			Value	Residual	Lagrange Multiplier			
[5]	nlc3	1.11E-15	1.11E-15	6.0000	Active	NLEC		
[6]	nlc1_G	4.31E-16	4.31E-16	.	Active	NLIC	LinDep	
[7]	nlc2_G	0	0	-6.0000	Active	NLIC		

Linearly Dependent Active Boundary Constraints

Parameter	N	Kind
ctox	8	Lower BC
pools	10	Lower BC

Output 7.7.7 *continued*

Linearly Dependent Gradients of Active Nonlinear Constraints	
Parameter	N
nlc3	6

The same problem can be specified in many different ways. For example, the following specification uses an INEST= data set containing the values of the starting point and of the constants COST, COSTB, COSTC, COSTX, COSTY, CA, CB, CC, and CD:

```
data init1(type=est);
   input _type_ $ amountx amounty amounta amountb
         amountc pooltox pooltoy ctox ctoy pools
         _rhs_ costa costb costc costx costy
         ca cb cc cd;
   datalines;
parms  1 1 1 1 1 1 1 1 1 1
       . 6 16 10 9 15  2.5 1.5 2. 3.
;

proc nlp inest=init1 all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
   bounds 0 <= amountx amounty amounta amountb amountc,
              amountx <= 100,
              amounty <= 200,
          0 <= pooltox pooltoy ctox ctoy,
          1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
          pooltox + ctox = amountx,
          pooltoy + ctoy = amounty,
          ctox + ctoy    = amountc;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = ca * amountx - pools * pooltox - cc * ctox;
   nlc2 = cb * amounty - pools * pooltoy - cc * ctoy;
   nlc3 = cd * amounta + amountb - pools * (amounta + amountb);
run;
```

The third specification uses an INEST= data set containing the boundary and linear constraints in addition to the values of the starting point and of the constants. This specification also writes the model specification into an OUTMOD= data set:

```
data init2(type=est);
   input _type_ $ amountx amounty amounta amountb amountc
                  pooltox pooltoy ctox ctoy pools
                  _rhs_ costa costb costc costx costy;
```

```
   datalines;
parms      1   1  1  1  1    1   1   1  1  1
           .   6 16 10  9   15 2.5 1.5  2  3
lowerbd    0   0  0  0  0    0   0   0  0  1
           .   .  .  .  .    .   .   .  .  .
upperbd  100 200  .  .  .    .   .   .  .  3
           .   .  .  .  .    .   .   .  .  .
eq         .   .  1  1  .   -1  -1   .  .  .
           0   .  .  .  .    .   .   .  .  .
eq         1   .  .  .  .   -1   .  -1  .  .
           0   .  .  .  .    .   .   .  .  .
eq         .   1  .  .  .    .  -1   . -1  .
           0   .  .  .  .    .   .   .  .  .
eq         .   .  .  .  1    .   .  -1 -1  .
           0   .  .  .  .    .   .   .  .  .
;

proc nlp inest=init2 outmod=model all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

The fourth specification not only reads the INEST=INIT2 data set, it also uses the model specification from the MODEL data set that was generated in the last specification. The PROC NLP call now contains only the defining variable statements:

```
proc nlp inest=init2 model=model all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
run;
```

All four specifications start with the same starting point of all variables equal to 1 and generate the same results. However, there exist several local optima to this problem, as is pointed out in Liebman et al. (1986, p. 130).

```
proc nlp inest=init2 model=model all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy = 0,
         pools = 2;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
run;
```

This starting point with all variables equal to 0 is accepted as a local solution with $profit = 0$, which minimizes rather than maximizes the profit.

Example 7.8: Chemical Equilibrium

The following example is used in many test libraries for nonlinear programming and was taken originally from Bracken and McCormick (1968).

The problem is to determine the composition of a mixture of various chemicals satisfying its chemical equilibrium state. The second law of thermodynamics implies that a mixture of chemicals satisfies its chemical equilibrium state (at a constant temperature and pressure) when the free energy of the mixture is reduced to a minimum. Therefore the composition of the chemicals satisfying its chemical equilibrium state can be found by minimizing the function of the free energy of the mixture.

Notation:

m	number of chemical elements in the mixture
n	number of compounds in the mixture
x_j	number of moles for compound j, $j = 1, \ldots, n$
s	total number of moles in the mixture ($s = \sum_{i=1}^{n} x_j$)
a_{ij}	number of atoms of element i in a molecule of compound j
b_i	atomic weight of element i in the mixture

Constraints for the Mixture:

- The number of moles must be positive:

$$x_j > 0, \quad j = 1, \ldots, n$$

- There are m mass balance relationships,

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m$$

Objective Function: Total Free Energy of Mixture

$$f(x) = \sum_{j=1}^{n} x_j \left[c_j + \ln\left(\frac{x_j}{s}\right) \right]$$

with

$$c_j = \left(\frac{F^\circ}{RT}\right)_j + \ln P$$

where F°/RT is the model standard free energy function for the jth compound (found in tables) and P is the total pressure in atmospheres.

Minimization Problem:

Determine the parameters x_j that minimize the objective function $f(x)$ subject to the nonnegativity and linear balance constraints.

Numeric Example:

Determine the equilibrium composition of compound $\frac{1}{2}N_2H_4 + \frac{1}{2}O_2$ at temperature $T = 3500°\mathrm{K}$ and pressure $P = 750\mathrm{psi}$.

				a_{ij}		
				$i=1$	$i=2$	$i=3$
j	Compound	$(F°/RT)_j$	c_j	H	N	O
1	H	-10.021	-6.089	1		
2	H_2	-21.096	-17.164	2		
3	H_2O	-37.986	-34.054	2		1
4	N	-9.846	-5.914		1	
5	N_2	-28.653	-24.721		2	
6	NH	-18.918	-14.986	1	1	
7	NO	-28.032	-24.100		1	1
8	O	-14.640	-10.708			1
9	O_2	-30.594	-26.662			2
10	OH	-26.111	-22.179	1		1

Example Specification:

```
proc nlp tech=tr pall;
      array c[10] -6.089 -17.164 -34.054  -5.914 -24.721
               -14.986 -24.100 -10.708 -26.662 -22.179;
   array x[10] x1-x10;
   min y;
   parms x1-x10 = .1;
   bounds 1.e-6 <= x1-x10;
   lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
          1. = x4 + 2. * x5 + x6 + x7,
          1. = x3 + x7  + x8 + 2. * x9 + x10;
   s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
   y = 0.;
   do j = 1 to 10;
      y = y + x[j] * (c[j] + log(x[j] / s));
   end;
run;
```

Displayed Output:

The iteration history given in Output 7.8.1 does not show any problems.

Output 7.8.1 Iteration History

PROC NLP: Nonlinear Minimization

Trust Region Optimization

Without Parameter Scaling

Iteration	Restarts	Function Calls	Active Constraints		Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Trust Region Radius
1	0	2	3	'	-47.33412	2.2790	6.0765	2.456	1.000
2	0	3	3	'	-47.70043	0.3663	8.5592	0.908	0.418
3	0	4	3		-47.73074	0.0303	6.4942	0	0.359
4	0	5	3		-47.73275	0.00201	4.7606	0	0.118
5	0	6	3		-47.73554	0.00279	3.2125	0	0.0168
6	0	7	3		-47.74223	0.00669	1.9552	110.6	0.00271
7	0	8	3		-47.75048	0.00825	1.1157	102.9	0.00563
8	0	9	3		-47.75876	0.00828	0.4165	3.787	0.0116
9	0	10	3		-47.76101	0.00224	0.0716	0	0.0121
10	0	11	3		-47.76109	0.000083	0.00238	0	0.0111
11	0	12	3		-47.76109	9.609E-8	2.733E-6	0	0.00248

Optimization Results			
Iterations	11	**Function Calls**	13
Hessian Calls	12	**Active Constraints**	3
Objective Function	-47.76109086	**Max Abs Gradient Element**	1.8637498E-6
Lambda	0	**Actual Over Pred Change**	0
Radius	0.0024776027		

GCONV convergence criterion satisfied.

Output 7.8.2 lists the optimal parameters with the gradient.

Output 7.8.2 Optimization Results

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	x1	0.040668	-9.785055
2	x2	0.147730	-19.570110
3	x3	0.783153	-34.792170
4	x4	0.001414	-12.968921
5	x5	0.485247	-25.937841
6	x6	0.000693	-22.753976
7	x7	0.027399	-28.190984
8	x8	0.017947	-15.222060
9	x9	0.037314	-30.444120
10	x10	0.096871	-25.007115

Value of Objective Function = -47.76109086

The three equality constraints are satisfied at the solution, as shown in Output 7.8.3.

Output 7.8.3 Linear Constraints at Solution

PROC NLP: Nonlinear Minimization

Linear Constraints Evaluated at Solution

1 ACT 4.8572E-16 = 2.0000 - 1.0000 * x1 - 2.0000 * x2 - 2.0000 * x3 - 1.0000 * x6 - 1.0000 * x10

2 ACT 2.8796E-16 = 1.0000 - 1.0000 * x4 - 2.0000 * x5 - 1.0000 * x6 - 1.0000 * x7

3 ACT 1.1102E-16 = 1.0000 - 1.0000 * x3 - 1.0000 * x7 - 1.0000 * x8 - 2.0000 * x9 - 1.0000 * x10

The Lagrange multipliers are given in Output 7.8.4.

Output 7.8.4 Lagrange Multipliers

PROC NLP: Nonlinear Minimization

First Order Lagrange Multipliers	
Active Constraint	Lagrange Multiplier
Linear EC [1]	9.785055
Linear EC [2]	12.968921
Linear EC [3]	15.222060

The elements of the projected gradient must be small to satisfy a necessary first-order optimality condition. The projected gradient is given in Output 7.8.5.

Output 7.8.5 Projected Gradient

PROC NLP: Nonlinear Minimization

Projected Gradient	
Free Dimension	Projected Gradient
1	4.5770097E-9
2	6.868334E-10
3	-7.283017E-9
4	-0.000001864
5	-0.000001434
6	-0.000001361
7	-0.000000294

The projected Hessian matrix shown in Output 7.8.6 is positive definite, satisfying the second-order optimality condition.

Output 7.8.6 Projected Hessian Matrix

PROC NLP: Nonlinear Minimization

Projected Hessian Matrix							
	X1	X2	X3	X4	X5	X6	X7
X1	20.903196985	-0.122067474	2.6480263467	3.3439156526	-1.373829641	-1.491808185	1.1462413516
X2	-0.122067474	565.97299938	106.54631863	-83.7084843	-37.43971036	-36.20703737	-16.635529
X3	2.6480263467	106.54631863	1052.3567179	-115.230587	182.89278895	175.97949593	-57.04158208
X4	3.3439156526	-83.7084843	-115.230587	37.529977667	-4.621642366	-4.574152161	10.306551561
X5	-1.373829641	-37.43971036	182.89278895	-4.621642366	79.326057844	22.960487404	-12.69831637
X6	-1.491808185	-36.20703737	175.97949593	-4.574152161	22.960487404	66.669897023	-8.121228758
X7	1.1462413516	-16.635529	-57.04158208	10.306551561	-12.69831637	-8.121228758	14.690478023

The following PROC NLP call uses a specified analytical gradient and the Hessian matrix is computed by finite-difference approximations based on the analytic gradient:

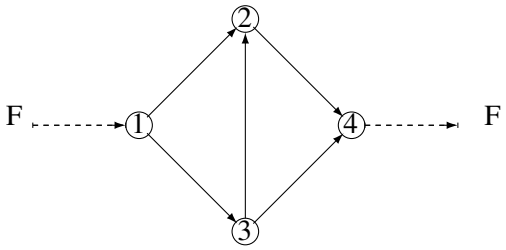
```
proc nlp tech=tr fdhessian all;
   array c[10] -6.089 -17.164 -34.054  -5.914 -24.721
               -14.986 -24.100 -10.708 -26.662 -22.179;
   array x[10] x1-x10;
   array g[10] g1-g10;
   min y;
   parms x1-x10 = .1;
   bounds 1.e-6 <= x1-x10;
   lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
          1. = x4 + 2. * x5 + x6 + x7,
          1. = x3 + x7  + x8 + 2. * x9 + x10;
   s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
   y = 0.;
   do j = 1 to 10;
      y = y + x[j] * (c[j] + log(x[j] / s));
      g[j] = c[j] + log(x[j] / s);
   end;
run;
```

The results are almost identical to those of the previous run.

Example 7.9: Minimize Total Delay in a Network

The following example is taken from the user's guide of GINO (Liebman et al. 1986). A simple network of five roads (arcs) can be illustrated by the path diagram:

Figure 7.13 Simple Road Network



The five roads connect four intersections illustrated by numbered nodes. Each minute F vehicles enter and leave the network. Arc (i, j) refers to the road from intersection i to intersection j, and the parameter x_{ij} refers to the flow from i to j. The law that traffic flowing into each intersection j must also flow out is described by the linear equality constraint

$$\sum_i x_{ij} = \sum_i x_{ji} \ , \quad j = 1, \ldots, n$$

In general, roads also have an upper capacity, which is the number of vehicles which can be handled per minute. The upper limits c_{ij} can be enforced by boundary constraints

$$0 \le x_{ij} \le c_{ij} \ , \quad i, j = 1, \ldots, n$$

Finding the maximum flow through a network is equivalent to solving a simple linear optimization problem, and for large problems, PROC LP or PROC NETFLOW can be used. The objective function is

$$\max \quad f = x_{24} + x_{34}$$

and the constraints are

$$\begin{aligned} x_{13} &= x_{32} + x_{34} \\ x_{12} + x_{32} &= x_{24} \\ x_{12} + x_{13} &= x_{24} + x_{34} \\ 0 \le x_{12}, x_{32}, x_{34} &\le 10 \\ 0 \le x_{13}, x_{24} &\le 30 \end{aligned}$$

The three linear equality constraints are linearly dependent. One of them is deleted automatically by the PROC NLP subroutines. Even though the default technique is used for this small example, any optimization subroutine can be used.

```
proc nlp all initial=.5;
   max y;
   parms x12 x13 x32 x24 x34;
   bounds x12 <= 10,
          x13 <= 30,
          x32 <= 10,
          x24 <= 30,
          x34 <= 10;
   /* what flows into an intersection must flow out */
   lincon x13 = x32 + x34,
          x12 + x32 = x24,
          x24 + x34 = x12 + x13;
   y = x24 + x34 + 0*x12 + 0*x13 + 0*x32;
run;
```

The iteration history is given in Output 7.9.1, and the optimal solution is given in Output 7.9.2.

Output 7.9.1 Iteration History

PROC NLP: Nonlinear Maximization

Newton-Raphson Ridge Optimization

Without Parameter Scaling

Iteration		Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Between Actual and Predicted Change
1	*	0	2	4	20.25000	19.2500	0.5774	0.0313	0.860
2	*	0	3	5	30.00000	9.7500	0	0.0313	1.683

Optimization Results			
Iterations	2	Function Calls	4
Hessian Calls	3	Active Constraints	5
Objective Function	30	Max Abs Gradient Element	0
Ridge	0	Actual Over Pred Change	1.6834532374

All parameters are actively constrained. Optimization cannot proceed.

Output 7.9.2 Optimization Results

PROC NLP: Nonlinear Maximization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	x12	10.000000	0	Upper BC
2	x13	20.000000	0	
3	x32	10.000000	0	Upper BC
4	x24	20.000000	1.000000	
5	x34	10.000000	1.000000	Upper BC

Value of Objective Function = 30

Finding a traffic pattern that minimizes the total delay to move F vehicles per minute from node 1 to node 4 introduces nonlinearities that, in turn, demand nonlinear optimization techniques. As traffic volume increases, speed decreases. Let t_{ij} be the travel time on arc (i, j) and assume that the following formulas describe the travel time as decreasing functions of the amount of traffic:

$$\begin{aligned} t_{12} &= 5 + 0.1x_{12}/(1 - x_{12}/10) \\ t_{13} &= x_{13}/(1 - x_{13}/30) \\ t_{32} &= 1 + x_{32}/(1 - x_{32}/10) \\ t_{24} &= x_{24}/(1 - x_{24}/30) \\ t_{34} &= 5 + 0.1x_{34}/(1 - x_{34}/10) \end{aligned}$$

These formulas use the road capacities (upper bounds), assuming $F = 5$ vehicles per minute have to be moved through the network. The objective function is now

$$\min \quad f = t_{12}x_{12} + t_{13}x_{13} + t_{32}x_{32} + t_{24}x_{24} + t_{34}x_{34}$$

and the constraints are

$$\begin{aligned} x_{13} &= x_{32} + x_{34} \\ x_{12} + x_{32} &= x_{24} \\ x_{24} + x_{34} &= F = 5 \\ 0 \le x_{12}, x_{32}, x_{34} &\le 10 \\ 0 \le x_{13}, x_{24} &\le 30 \end{aligned}$$

Again, the default algorithm is used:

```
proc nlp all initial=.5;
   min y;
   parms x12 x13 x32 x24 x34;
   bounds x12 x13 x32 x24 x34 >= 0;
   lincon x13 = x32 + x34,  /* flow in = flow out */
          x12 + x32 = x24,
          x24 + x34 = 5;    /* = f = desired flow */
   t12 = 5 + .1 * x12 / (1 - x12 / 10);
   t13 = x13 / (1 - x13 / 30);
   t32 = 1 + x32 / (1 - x32 / 10);
   t24 = x24 / (1 - x24 / 30);
   t34 = 5 + .1 * x34 / (1 - x34 / 10);
   y = t12*x12 + t13*x13 + t32*x32 + t24*x24 + t34*x34;
run;
```

The iteration history is given in Output 7.9.3, and the optimal solution is given in Output 7.9.4.

Output 7.9.3 Iteration History

PROC NLP: Nonlinear Minimization

Newton-Raphson Ridge Optimization

Without Parameter Scaling

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Between Actual and Predicted Change
1	0	2	4	40.30303	0.3433	4.44E-16	0	0.508

Optimization Results			
Iterations	1	Function Calls	3
Hessian Calls	2	Active Constraints	4
Objective Function	40.303030303	Max Abs Gradient Element	4.440892E-16
Ridge	0	Actual Over Pred Change	0.5083585587

ABSGCONV convergence criterion satisfied.

Output 7.9.4 Optimization Results

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	x12	2.500000	5.777778	
2	x13	2.500000	5.702479	
3	x32	1.114018E-17	1.000000	Lower BC
4	x24	2.500000	5.702479	
5	x34	2.500000	5.777778	

Value of Objective Function = 40.303030303

The active constraints and corresponding Lagrange multiplier estimates (costs) are given in Output 7.9.5 and Output 7.9.6, respectively.

Output 7.9.5 Linear Constraints at Solution

PROC NLP: Nonlinear Minimization

Linear Constraints Evaluated at Solution

1	ACT	4.4409E-16	=	0	+	1.0000	*	x13	-	1.0000	*	x32 - 1.0000 * x34
2	ACT	4.4409E-16	=	0	+	1.0000	*	x12	+	1.0000	*	x32 - 1.0000 * x24
3	ACT	0	=	-5.0000	+	1.0000	*	x24	+	1.0000	*	x34

Output 7.9.6 Lagrange Multipliers at Solution

PROC NLP: Nonlinear Minimization

First Order Lagrange Multipliers

Active Constraint		Lagrange Multiplier
Lower BC	x32	0.924702
Linear EC	[1]	5.702479
Linear EC	[2]	5.777778
Linear EC	[3]	11.480257

Output 7.9.7 shows that the projected gradient is very small, satisfying the first-order optimality criterion.

Output 7.9.7 Projected Gradient at Solution

PROC NLP: Nonlinear Minimization

Projected Gradient	
Free Dimension	Projected Gradient
1	4.440892E-16

The projected Hessian matrix (shown in Output 7.9.8) is positive definite, satisfying the second-order optimality criterion.

Output 7.9.8 Projected Hessian at Solution

PROC NLP: Nonlinear Minimization

Projected Hessian Matrix	
	X1
X1	1.535309013

References

Abramowitz, M. and Stegun, I. A. (1972), *Handbook of Mathematical Functions*, New York: Dover Publications.

Al-Baali, M. and Fletcher, R. (1985), "Variational Methods for Nonlinear Least Squares," *Journal of the Operations Research Society*, 36, 405–421.

Al-Baali, M. and Fletcher, R. (1986), "An Efficient Line Search for Nonlinear Least Squares," *Journal of Optimization Theory and Applications*, 48, 359–377.

Bard, Y. (1974), *Nonlinear Parameter Estimation*, New York: Academic Press.

Beale, E. M. L. (1972), "A Derivation of Conjugate Gradients," in F. A. Lootsma, ed., *Numerical Methods for Nonlinear Optimization*, London: Academic Press.

Betts, J. T. (1977), "An Accelerated Multiplier Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, 21, 137–174.

Bracken, J. and McCormick, G. P. (1968), *Selected Applications of Nonlinear Programming*, New York: John Wiley & Sons.

Chamberlain, R. M., Powell, M. J. D., Lemarechal, C., and Pedersen, H. C. (1982), "The Watchdog Technique for Forcing Convergence in Algorithms for Constrained Optimization," *Mathematical Programming*, 16, 1–17.

Cramer, J. S. (1986), *Econometric Applications of Maximum Likelihood Methods*, Cambridge: Cambridge University Press.

Dennis, J. E., Gay, D. M., and Welsch, R. E. (1981), "An Adaptive Nonlinear Least-Squares Algorithm," *ACM Transactions on Mathematical Software*, 7, 348–368.

Dennis, J. E. and Mei, H. H. W. (1979), "Two New Unconstrained Optimization Algorithms Which Use Function and Gradient Values," *Journal of Optimization Theory and Applications*, 28, 453–482.

Dennis, J. E. and Schnabel, R. B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall.

Eskow, E. and Schnabel, R. B. (1991), "Algorithm 695: Software for a New Modified Cholesky Factorization," *ACM Transactions on Mathematical Software*, 17, 306–312.

Fletcher, R. (1987), *Practical Methods of Optimization*, 2nd Edition, Chichester, UK: John Wiley & Sons.

Fletcher, R. and Powell, M. J. D. (1963), "A Rapidly Convergent Descent Method for Minimization," *Computer Journal*, 6, 163–168.

Fletcher, R. and Xu, C. (1987), "Hybrid Methods for Nonlinear Least Squares," *Journal of Numerical Analysis*, 7, 371–389.

Gallant, A. R. (1987), *Nonlinear Statistical Models*, New York: John Wiley & Sons.

Gay, D. M. (1983), "Subroutines for Unconstrained Minimization," *ACM Transactions on Mathematical Software*, 9, 503–524.

George, J. A. and Liu, J. W. (1981), *Computer Solutions of Large Sparse Positive Definite Systems*, Englewood Cliffs, NJ: Prentice-Hall.

Gill, E. P., Murray, W., Saunders, M. A., and Wright, M. H. (1983), "Computing Forward-Difference Intervals for Numerical Optimization," *SIAM Journal on Scientific and Statistical Computing*, 4, 310–321.

Gill, E. P., Murray, W., Saunders, M. A., and Wright, M. H. (1984), "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints," *ACM Transactions on Mathematical Software*, 10, 282–298.

Gill, P. E., Murray, W., and Wright, M. H. (1981), *Practical Optimization*, New York: Academic Press.

Goldfeld, S. M., Quandt, R. E., and Trotter, H. F. (1966), "Maximisation by Quadratic Hill-Climbing," *Econometrica*, 34, 541–551.

Hambleton, R. K., Swaminathan, H., and Rogers, H. J. (1991), *Fundamentals of Item Response Theory*, Newbury Park, CA: Sage Publications.

Hartmann, W. M. (1992a), *Applications of Nonlinear Optimization with PROC NLP and SAS/IML Software*, Technical report, SAS Institute Inc., Cary, NC.

Hartmann, W. M. (1992b), *Nonlinear Optimization in IML, Releases 6.08, 6.09, 6.10*, Technical report, SAS Institute Inc., Cary, NC.

Haverly, C. A. (1978), "Studies of the Behavior of Recursion for the Pooling Problem," *SIGMAP Bulletin, Association for Computing Machinery*, 25, 19–28.

Hock, W. and Schittkowski, K. (1981), *Test Examples for Nonlinear Programming Codes*, volume 187 of *Lecture Notes in Economics and Mathematical Systems*, Berlin: Springer-Verlag.

Jennrich, R. I. and Sampson, P. F. (1968), "Application of Stepwise Regression to Nonlinear Estimation," *Technometrics*, 10, 63–72.

Lawless, J. F. (1982), *Statistical Methods and Methods for Lifetime Data*, New York: John Wiley & Sons.

Liebman, J., Lasdon, L., Schrage, L., and Waren, A. (1986), *Modeling and Optimization with GINO*, Redwood City, CA: Scientific Press.

Lindström, P. and Wedin, P. A. (1984), "A New Line-Search Algorithm for Nonlinear Least-Squares Problems," *Mathematical Programming*, 29, 268–296.

Moré, J. J. (1978), "The Levenberg-Marquardt Algorithm: Implementation and Theory," in G. A. Watson, ed., *Lecture Notes in Mathematics*, volume 30, 105–116, Berlin: Springer-Verlag.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1981), "Testing Unconstrained Optimization Software," *ACM Transactions on Mathematical Software*, 7, 17–41.

Moré, J. J. and Sorensen, D. C. (1983), "Computing a Trust-Region Step," *SIAM Journal on Scientific and Statistical Computing*, 4, 553–572.

Moré, J. J. and Wright, S. J. (1993), *Optimization Software Guide*, Philadelphia: SIAM.

Murtagh, B. A. and Saunders, M. A. (1983), *MINOS 5.0 User's Guide*, Technical Report SOL 83-20, Stanford University.

Nelder, J. A. and Mead, R. (1965), "A Simplex Method for Function Minimization," *Computer Journal*, 7, 308–313.

Polak, E. (1971), *Computational Methods in Optimization*, New York: Academic Press.

Powell, M. J. D. (1977), "Restart Procedures for the Conjugate Gradient Method," *Mathematical Programming*, 12, 241–254.

Powell, M. J. D. (1978a), "Algorithms for Nonlinear Constraints That Use Lagrangian Functions," *Mathematical Programming*, 14, 224–248.

Powell, M. J. D. (1978b), "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," in G. A. Watson, ed., *Lecture Notes in Mathematics*, volume 630, 144–175, Berlin: Springer-Verlag.

Powell, M. J. D. (1982a), "Extensions to Subroutine VF02AD," in R. F. Drenick and F. Kozin, eds., *Systems Modeling and Optimization, Lecture Notes in Control and Information Sciences*, volume 38, 529–538, Berlin: Springer-Verlag.

Powell, M. J. D. (1982b), *VMCWD: A Fortran Subroutine for Constrained Optimization*, Technical Report DAMTP 1982/NA4, Cambridge University.

Powell, M. J. D. (1992), "A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation," *DAMTP/NA5*.

Rosenbrock, H. H. (1960), "An Automatic Method for Finding the Greatest or Least Value of a Function," *Computer Journal*, 3, 175–184.

Schittkowski, K. (1980), *Nonlinear Programming Codes—Information, Tests, Performance*, volume 183 of *Lecture Notes in Economics and Mathematical Systems*, Berlin: Springer-Verlag.

Schittkowski, K. (1987), *More Test Examples for Nonlinear Programming Codes*, volume 282 of *Lecture Notes in Economics and Mathematical Systems*, Berlin: Springer-Verlag.

Schittkowski, K. and Stoer, J. (1979), "A Factorization Method for the Solution of Constrained Linear Least Squares Problems Allowing Subsequent Data Changes," *Numerische Mathematik*, 31, 431–463.

Stewart, G. W. (1967), "A Modification of Davidon's Minimization Method to Accept Difference Approximations of Derivatives," *Journal of the Association for Computing Machinery*, 14, 72–83.

Wedin, P. A. and Lindström, P. (1987), *Methods and Software for Nonlinear Least Squares Problems*, Technical Report Report No. UMINF 133.87, University of Umeå.

Whitaker, D., Triggs, C. M., and John, J. A. (1990), "Construction of Block Designs Using Mathematical Programming," *Journal of the Royal Statistical Society, Series B*, 52, 497–503.

Wolfe, P. (1982), "Checking the Calculation of Gradients," *ACM Transactions on Mathematical Software*, 8, 337–343.

Subject Index

active set methods, 611
    quadratic programming, 575, 601
affine step, 44, 46
arc capacity
    INTPOINT procedure, 92
    NETFLOW procedure, 340
arc names
    INTPOINT procedure, 56, 79, 123
    NETFLOW procedure, 335, 343

backtracking rules
    LP procedure, 186, 214
balancing network problems
    INTPOINT procedure, 84
    NETFLOW procedure, 340
Bard function, 642
Bartels-Golub decomposition, 334, 360, 398
BFGS update method, 576
big-M method, 356
blending constraints
    INTPOINT procedure, 50
    NETFLOW procedure, 307
boundary constraints
    NLP procedure, 578
branch-and-bound, 210
    branching variable, 214, 215
    breadth-first search, 213
    control options, 186, 213
    depth-first search, 213
bypass arc
    INTPOINT procedure, 75
    NETFLOW procedure, 330

case sensitivity
    INTPOINT procedure, 80, 81, 111
    NETFLOW procedure, 336, 349, 392
centering step, 45, 47
central path
    INTPOINT procedure, 43
    NETFLOW procedure, 431
Cholesky factor, 603
COBYLA algorithm, 605, 606, 610
coefficients
    INTPOINT procedure, 93
    LP procedure, 192
    NETFLOW procedure, 341
columns
    INTPOINT procedure, 93
    LP procedure, 192
    NETFLOW procedure, 341
complementarity
    INTPOINT procedure, 42, 46, 124
    NETFLOW procedure, 430
complete pricing
    LP procedure, 191
computational problems
    NLP procedure, 615–617
computational resources, *see* See also memory requirements
    NLP procedure, 632, 633
confidence intervals, 589
    output options, 627
    profile confidence limits, 561
conjugate-descent update method, 576
conjugate gradient methods, 575, 605
constraint summary
    LP procedure, 228, 237
converting NPSC to LP
    INTPOINT procedure, 106
costs
    INTPOINT procedure, 94
    NETFLOW procedure, 342
covariance matrix, 561, 618, 626
    displaying, 572
crossproduct Jacobian matrix, 579, 626
    definition, 596
    displaying, 572
    saving, 571
current tableau
    LP procedure, 207
cycling
    NETFLOW procedure, 357, 358, 396

data compression
    LP procedure, 188
data sets, *see* input data sets, *see* output data sets
demands
    INTPOINT procedure, 94
    NETFLOW procedure, 342
dense input format
    INTPOINT procedure, 56, 63, 96, 99, 101
    LP procedure, 170
    NETFLOW procedure, 314, 379, 380, 411, 414
derivatives, 596
    computing, 545, 596
    finite differences, 607
devex method, 190

DFP update method, 576
displayed output
    LP procedure, 184, 193, 226–229
    NETFLOW procedure, 344
    NLP procedure, 629, 630
distribution problem, 48
double dogleg method, 575, 604
dual activities
    LP procedure, 206
dual BFGS update method, 576
dual DFP update method, 576
dual problem
    INTPOINT procedure, 42
    NETFLOW procedure, 429
dual variables
    INTPOINT procedure, 42
    NETFLOW procedure, 396, 429
duality gap
    INTPOINT procedure, 42, 124
    NETFLOW procedure, 430
dynamic pricing
    LP procedure, 191

efficiency
    INTPOINT procedure, 119–123
    NETFLOW procedure, 410–414
embedded networks
    INTPOINT procedure, 106
    NETFLOW procedure, 385
examples, *see* INTPOINT examples, *see* LP examples, *see* NETFLOW examples, *see* NLP examples, *see* INTPOINT examples, *see* LP examples, *see* NETFLOW examples, *see* NLP examples
excess node
    INTPOINT procedure, 118
    NETFLOW procedure, 405

feasible region, 595
feasible solution, 595
finite-difference approximations
    central differences, 607
    computation of, 563
    forward differences, 607
    NLP procedure, 562, 563
    second-order derivatives, 563
first-order conditions
    local minimum, 595
Fletcher-Reeves update method, 576
flow conservation constraints
    INTPOINT procedure, 39, 51, 54
    NETFLOW procedure, 304, 312, 419
function convergence
    NLP procedure, 559
functional summary
    INTPOINT procedure, 71
    LP procedure, 179
    NETFLOW procedure, 321
    NLP procedure, 556

global solution, 595
goal-programming model, 190
Goldstein conditions, 603, 605, 614
gradient vector
    checking correctness, 609
    convergence, 560
    definition, 596
    local optimality conditions, 595
    projected gradient, 612
    specifying, 580
grid points, 560, 570, 571

Hessian matrix, 626
    definition, 596
    displaying, 573
    finite-difference approximations, 563, 607
    initial estimate, 566
    local optimality conditions, 595
    projected, 612
    saving, 571
    scaling, 565, 609
    specifying, 581
    update method, 576
hybrid quasi-Newton methods, 575, 576, 606

infeasibility
    INTPOINT procedure, 42, 46, 113
    LP procedure, 184, 209, 228
    NETFLOW procedure, 399
infeasible information summary
    LP procedure, 228
infinity
    INTPOINT procedure, 78
    LP procedure, 190
    NETFLOW procedure, 334
initial basic feasible solution
    NETFLOW procedure, 329
input data sets
    INTPOINT procedure, 74, 100
    LP procedure, 183, 226
    NETFLOW procedure, 328, 380
    NLP procedure, 545, 621, 622
integer iteration log, 212
integer programs, 168, 210
interactive processing
    LP procedure, 175, 185, 221, 222
    NETFLOW procedure, 326
interior point algorithm
    INTPOINT procedure, 41

network problems, 418
options (NETFLOW), 364
intermediate variable, 639
INTPOINT examples, 126
altering arc data, 134, 146
converting PROC INTPOINT format to MPS format, 160
linear program, 158
MPS format, 158
nonarc variables, 151
production, inventory, distribution problem, 127
production, inventory, distribution problem (OPTMODEL), 162
side constraints, 139, 146, 151
INTPOINT procedure
affine step, 44, 46
arc names, 79, 123
balancing supply and demand, 84, 118
blending constraints, 50
bypass arc, 75
case sensitivity, 80, 81, 83, 111
centering step, 45, 47
central path, 43
coefficients, 93
columns, 93
complementarity, 42, 46, 124
converting NPSC to LP, 106
costs, 94
data set options, 74
default arc capacity, 76
default arc cost, 77
default constraint type, 76
default lower bound, 77
default objective function, 77
default options, 122
default upper bounds, 76
demands, 94
dense format, 56, 63, 96, 99, 101
details, 100
distribution problem, 48
dual problem, 42
dual variables, 42
duality gap, 42, 124
efficiency, 119–123
embedded networks, 53, 106
excess node, 118
flow conservation constraints, 39, 51, 54
functional summary, 71
general options, 75
infeasibility, 42, 46, 113
input data sets, 74, 100
interior point algorithm, 41
introductory LP example, 63
introductory NPSC example, 57
inventory problem, 48
Karush-Kuhn-Tucker conditions, 42, 45
linear programming problems, 41, 62
loop arcs, 112
maximum cost flow, 79
maximum flow problem, 78
memory requirements, 69, 76, 79, 119–123
missing supply and missing demand, 114
missing values, 114
MPS file conversion, 158
multicommodity problems, 52
multiple arcs, 112
multiprocess, multiproduct example, 53
network problems, 48
nonarc variables, 54
NPSC, 39
options classified by function, 71
output data sets, 74, 109
overview, 38
preprocessing, 42, 56, 63, 87
Primal-Dual with Predictor-Corrector algorithm, 41, 44
primal problem, 42
production-inventory-distribution problem, 48
proportionality constraints, 49
scaling input data, 82
shortest path problem, 83
side constraints, 39, 54
sparse format, 56, 63, 93, 102, 106
step length, 43
stopping criteria, 89, 123
supply-chain problem, 48
symbolic factorization, 44
syntax skeleton, 71
table of syntax elements, 71
termination criteria, 89, 123
TYPE variable, 98
upper bounds, 45
inventory problem, 48
iteration log
integer iteration log (LP), 212
LP procedure, 184, 230

Jacobian matrix, 626
constraint functions, 582
definition, 596
displaying, 573
objective functions, 582
saving, 571, 572
scaling, 609

Karush-Kuhn-Tucker conditions, 595, 611
INTPOINT procedure, 42, 45
NETFLOW procedure, 436

Kuhn-Tucker conditions, *see* Karush-Kuhn-Tucker conditions

labels
  assigning to decision variables, 583
Lagrange multipliers, 595, 612, 626
Lagrangian function, 595
least squares problems
  definition of, 544
  optimization algorithms, 600
legacy procedures, 13
  migration from, 14
Levenberg-Marquardt minimization, 575
  least squares method, 606
line-search methods, 567, 613
  step length, 561, 569
linear complementarity problem, 575
  quadratic programming, 601
linear constraints
  NLP procedure, 584
linear programming problems, 168
  Bartels-Golub update, 169
  INTPOINT procedure, 41, 62
  NETFLOW procedure, 310
linearly constrained optimization, 602
local minimum
  first-order conditions, 595
  second-order conditions, 595
local solution, 595
loop arcs
  INTPOINT procedure, 112
  NETFLOW procedure, 392
lower bounds
  NETFLOW procedure, 343
LP examples, 232
  assignment problem, 277
  assignment problem (OPTMODEL), 294
  blending problem, 171, 233
  branch-and-bound search, 272
  converting PROC LP format to MPS format, 292
  fixed charges, 289, 296
  goal programming, 254
  infeasibilities, 263
  integer program, 260
  introductory example, 171
  mixed-integer program, 176
  MPS file conversion, 177
  multicommodity transshipment problem, 289
  multicommodity transshipment problem (OPTMODEL), 296
  preprocessing, 176
  price parametric programming, 247
  price sensitivity analysis, 241
  product mix problem, 254
  range analysis, 244
  restarting a problem, 244
  restarting an integer program, 266
  scheduling problem, 282
  sparse data format, 238
  special ordered sets, 251
LP procedure
  backtracking rules, 186, 214
  branch-and-bound, 186, 210, 213
  coefficients, 192
  columns, 192
  complete pricing, 191
  constraint summary, 228, 237
  current tableau, 207
  customizing search heuristics, 215
  data compression, 188
  data set options, 183
  dense format, 170
  details, 201
  devex method, 190
  displayed output, 184, 193, 226–230
  dual activities, 206
  dynamic pricing, 191
  functional summary, 179
  infeasible information summary, 228
  input data sets, 183, 226
  integer iteration log, 212
  integer programs, 210
  interactive processing, 175, 185, 221, 222
  introductory example, 171
  iteration log, 184, 230
  memory limit, 231
  memory requirements, 223
  missing values, 201
  mixed-integer programs, 210
  MPS file conversion, 177, 204
  multiple pricing, 191, 208
  ODS table names, 230
  ODS variable names, 230, 231
  options classified by function, 179
  ORLP macro variable, 207
  output data sets, 183, 223–225
  Output Delivery System (ODS), 230
  overview, 168
  parametric programming, 189, 220, 222
  partial pricing, 191
  pause processing, 185
  preprocessing, 176, 185, 209, 210
  price parametric programming, 221
  price sensitivity analysis, 218, 229
  pricing strategies, 208
  problem definition statements, 174
  problem input, 173
  problem summary, 227, 234

projected objective value, 213
projected pseudocost, 214
range analysis, 189, 219
range coefficient, 195
reduced costs, 206
reset options, 196
right-hand-side constants, 197
right-hand-side parametric programming, 220
right-hand-side sensitivity analysis, 217, 229
rows, 193, 197
scaling input data, 192, 209
sensitivity analysis, 189, 217, 222
simplex algorithm control options, 190
solution summary, 227, 234
sparse format, 171, 183, 201
suppress printing, 184
syntax skeleton, 178
table of syntax elements, 179
terminate processing, 195
tolerance, 184, 186, 187, 190–192
TYPE variable, 198, 202, 236
variables, 170, 171, 200, 224, 225, 227, 235

macro variable
_ORNETFL, 415
macro variable
ORLP, 207
ORLP, 207
matrix
definition (NLP), 585
generation, 27
maximum flow problem
INTPOINT procedure, 78
NETFLOW procedure, 335
maximum likelihood Weibull estimation
using PROC NLP, 653
memory requirements
INTPOINT procedure, 69, 76, 79, 119–123
LP procedure, 223
NETFLOW procedure, 330–332, 334, 335, 410–414
merit function, 603
migration to PROC OPTMODEL
from PROC INTPOINT, 162
from PROC LP, 294, 296
from PROC NETFLOW, 532, 535, 537, 539
from PROC NLP, 634
missing values
INTPOINT procedure, 114
LP procedure, 201
NETFLOW procedure, 401
NLP procedure, 570, 631
mixed-integer programs, 168
form of, 168
LP procedure, 210
model building, 18
Moore-Penrose conditions, 620
MPS file conversion
INTPOINT procedure, 158
LP procedure, 177, 204
multicommodity problems
INTPOINT procedure, 52
NETFLOW procedure, 308
multiple arcs
INTPOINT procedure, 112
NETFLOW procedure, 393
multiple pricing
LP procedure, 191, 208

Nelder-Mead simplex method, 575, 605
NETFLOW examples, 466
constrained solution, 496
converting PROC NETFLOW format to MPS format, 530
distribution problem, 473
distribution problem (OPTMODEL), 537
generalized networks (OPTMODEL), 532
inventory problem, 473
inventory problem (OPTMODEL), 537
maximum flow (OPTMODEL), 535
minimum cost flow, 468
nonarc variables, 503
production problem, 473
production problem (OPTMODEL), 537
shortest path problem, 466
shortest path problem (OPTMODEL), 539
side constraints, 488, 503
unconstrained solution, 482
warm start, 472, 482, 488, 496
NETFLOW procedure
arc capacity, 340
arc names, 314, 317, 335, 343
balancing supply and demand, 340, 405
Bartels-Golub decomposition, 334, 360, 398
big-M method, 356
blending constraints, 307
bypass arc, 330
case sensitivity, 336, 349, 392
central path, 431
coefficients, 341
columns, 341
complementarity, 430
costs, 342
cycling, 357, 358, 396
data set options, 328
default arc capacity, 331
default arc cost, 332
default constraint type, 332

default lower flow bound, 332
default options, 413
demands, 332, 342
dense format, 310, 314, 379, 380, 411, 414
details, 380
dual variables, 396, 429
duality gap, 430
efficiency, 410–414
embedded networks, 385
excess node, 405
flow conservation constraints, 304, 312, 419
functional summary, 321
infeasibility, 399
infinity, 334
initial basic feasible solution, 329
input data sets, 328, 380
interactive processing, 326
interior point algorithm, 417
interior point options, 364
introductory example, 315
Karush-Kuhn-Tucker conditions, 436
key arc, 359
linear programming, 310
loop arcs, 392
lower bounds, 343
macro variable _ORNETFL, 415
major iteration, 395
maximum cost flow, 335
maximum flow problem, 335
memory limit, 416
memory requirements, 330–332, 334, 335, 410–414
minor iteration, 395
missing supply and missing demand, 401
multicommodity problems, 308
multiple arcs, 393
network models, 304
network problems, 418
nonarc variables, 312
nonkey arc, 359
NPSC, 311
options classified by function, 321
output data sets, 328, 329, 354, 388
overview, 303
pivot, 344
preprocessing, 366
pricing strategies, 360, 393
printing cautions, 349
printing options, 344
production-inventory-distribution problem, 304
proportionality constraints, 306
ratio test, 357, 358
reduced costs, 396
scaling input data, 338
shortest path problem, 339
side constraints, 305, 311, 399
sink nodes, 339
source nodes, 339
sparse format, 310, 314, 339, 382, 386, 411, 414
stages, 351
status, 396
stopping criteria, 433
supplies, 340
syntax skeleton, 321
table of syntax elements, 321
termination criteria, 367, 433
tightening bounds, 399
TYPE variable, 378
warm starts, 313, 340, 406, 408, 414
working basis matrix, 332, 334, 340, 358, 397
wraparound search, 394
network models, 304
network problems
format, 24
interior point algorithm, 418
INTPOINT procedure, 48
Newton-Raphson method, 575
with line search, 601
with ridging, 602
NLP examples, 642
approximate standard errors, 648
Bard function, 642
blending problem, 660
boundary constraints, 548
chemical equilibrium, 669
covariance matrix, 648
Hock and Schittkowski problem, 645
introductory examples, 546–549, 551, 553
least squares problem, 547, 642
linear constraints, 549, 645
maximum likelihood Weibull estimation, 653
maximum-likelihood estimates, 553, 649
migration to PROC OPTMODEL, 634
nonlinear constraints, 550, 551
nonlinear network problem, 675
quadratic programming problem, 644
restarting an optimization, 647
Rosenbrock function, 647
starting point, 645
statistical analysis, 648
trust region method, 647
unconstrained optimization, 546
NLP procedure
active set methods, 601, 611
boundary constraints, 578
choosing an optimization algorithm, 599
computational problems, 615–617
computational resources, 632, 633

conjugate gradient methods, 605
convergence difficulties, 616, 617
convergence status, 631
covariance matrix, 561, 572, 618, 620, 626
crossproduct Jacobian, 596, 626
debugging options, 568, 591
derivatives, 596
display function values, 572, 573
displayed output, 572, 629, 630
double dogleg method, 604
eigenvalue tolerance, 561
feasible region, 595
feasible solution, 595
feasible starting point, 613
finite-difference approximations, 562, 563, 607
first-order conditions, 595
function convergence, 559, 562
functional summary, 556
global solution, 595
Goldstein conditions, 603, 605, 614
gradient, 564, 596
gradient convergence, 560, 564
grid points, 560
Hessian, 566, 596, 607, 626
Hessian scaling, 565, 609
Hessian update method, 576
initial values, 566, 573, 621
input data sets, 545, 621, 622
iteration history, 629
Jacobian, 596, 626
Karush-Kuhn-Tucker conditions, 595
Lagrange multipliers, 595, 626
Lagrangian function, 595
least squares problems, 600
Levenberg-Marquardt method, 606
limiting function calls, 569
limiting number of iterations, 569
line-search methods, 567, 613
linear complementarity, 601
linear constraints, 584, 602
local optimality conditions, 595
local solution, 595
memory limit, 634
memory requirements, 597
missing values, 570, 631
Nelder-Mead simplex method, 605
Newton-Raphson method, 601, 602
nonlinear constraints, 580, 603
optimality criteria, 594
optimization algorithms, 575, 597
optimization history, 573
options classified by function, 556
output data sets, 546, 570, 571, 623, 624, 628
overview, 544
parameter convergence, 560, 577
precision, 560, 563, 608, 618
predicted reduction convergence, 562
profile confidence limits, 561
program statements, 590
projected gradient, 626
projected Hessian matrix, 626
quadratic programming, 599, 600, 603
quasi-Newton methods, 602, 606
rank of covariance matrix, 626
restricting output, 573, 574
restricting step length, 614
second-order conditions, 595
singularity criterion, 560, 570, 577
stationary point, 617
step length, 561
storing model files, 628
suppress printing, 570
table of syntax elements, 556
termination criteria, 610
time limit, 569
trust region method, 601
TYPE variable, 621–623, 625, 627, 628
unconstrained optimization, 602
variables, 621
nonarc variables
INTPOINT procedure, 54
NETFLOW procedure, 312
nonlinear optimization, 544, *see* NLP procedure
algorithms, 597
computational problems, 615
conjugate gradient methods, 605
feasible starting point, 613
hybrid quasi-Newton methods, 606
Levenberg-Marquardt method, 606
Nelder-Mead simplex method, 605
Newton-Raphson method with line search, 601
Newton-Raphson method with ridging, 602
nonlinear constraints, 580, 588, 603
optimization algorithms, 599
quasi-Newton method, 602
trust region method, 601
NPSC
INTPOINT procedure, 39
NETFLOW procedure, 311

objective function
INTPOINT procedure, 40, 41, 94
LP procedure, 169
NETFLOW procedure, 311, 429
NLP procedure, 544, 554, 586
ODS variable names
LP procedure, 231
online documentation, 10

optimality criteria, 594
optimization
    double dogleg method, 604
    introduction, 14
    linear constraints, 602
    nonlinear constraints, 603
    unconstrained, 602
optimization algorithms
    least squares problems, 600
    NLP procedure, 575, 597
    nonlinear optimization, 599
    quadratic programming, 599, 600
options classified by function, *see* functional summary, *see* functional summary, *see* functional summary, *see* functional summary
ORLP macro variable, 207
_ORNETFL macro variable, 415
output data sets
    INTPOINT procedure, 74, 109
    LP procedure, 183, 223–225
    NETFLOW procedure, 328, 329, 354, 388
    NLP procedure, 546, 570, 571, 623, 624, 628
Output Delivery System (ODS)
    LP procedure, 230
overview
    INTPOINT procedure, 38
    LP procedure, 168
    NETFLOW procedure, 303
    NLP procedure, 544
    optimization, 14

parametric control options
    LP procedure, 189
parametric programming, 189, 220, 222
partial pricing
    LP procedure, 191
pause processing
    LP procedure, 185
Polak-Ribiere update method, 576
Powell-Beale update method, 576
precision
    nonlinear constraints, 560, 608
    objective function, 563, 608
preprocessing
    INTPOINT procedure, 42, 56, 63, 87
    LP procedure, 176, 185, 209, 210
    NETFLOW procedure, 366
price parametric programming, 221
price sensitivity analysis, 218, 229
pricing strategies
    LP procedure, 208
    NETFLOW procedure, 360, 393
Primal-Dual with Predictor-Corrector algorithm
    INTPOINT procedure, 41, 44
problem definition statements
    LP procedure, 174
problem specification
    dense format, 20
    network format, 24
    Sparse Format, 20
problem summary
    LP procedure, 227, 234
production-inventory-distribution problem, 304
    INTPOINT procedure, 48
profile confidence limits, 589
    parameters for, 561
program statements
    NLP procedure, 590
projected gradient, 612, 626
projected Hessian matrix, 612, 626
projected objective value
    LP procedure, 213
projected pseudocost
    LP procedure, 214
proportionality constraints
    INTPOINT procedure, 49
    NETFLOW procedure, 306

quadratic programming, 566, 603
    active set methods, 601
    definition, 544
    linear complementarity problem, 601
    optimization algorithms, 599, 600
    specifying the objective function, 586
quasi-Newton methods, 575, 576, 602

random numbers
    seed, 574
range analysis, 189, 219
range coefficient
    LP procedure, 195
ranging control options
    LP procedure, 189
ratio test
    NETFLOW procedure, 357, 358
reduced costs
    LP procedure, 206
    NETFLOW procedure, 396
report writing, 33, 34
right-hand-side constants
    LP procedure, 197
right-hand-side parametric programming, 220
right-hand-side sensitivity analysis, 217, 229
Rosenbrock function, 546, 579–581, 583
Rosen-Suzuki problem, 589
rows
    LP procedure, 193, 197

scaling input data

INTPOINT procedure, 82
LP procedure, 192, 209
NETFLOW procedure, 338
second-order conditions
local minimum, 595
second-order derivatives
finite-difference approximations, 563
sensitivity analysis, 217, 222
sensitivity control options
LP procedure, 189
shortest path problem
INTPOINT procedure, 83
NETFLOW procedure, 339
side constraints
INTPOINT procedure, 39, 54
NETFLOW procedure, 305, 311
simplex algorithm control options
LP procedure, 190
singularity, 574, 620
absolute singularity criterion, 560
relative singularity criterion, 570, 577
sink nodes
NETFLOW procedure, 339
solution summary
LP procedure, 227, 234
source nodes
NETFLOW procedure, 339
sparse input format
INTPOINT procedure, 56, 63, 93, 102
LP procedure, 171, 183, 201
NETFLOW procedure, 314, 339, 382, 411, 414
summary (INTPOINT), 106
summary (NETFLOW), 386
special ordered set, 199
standard errors
computing, 573
step length, 614
INTPOINT procedure, 43
supplies
NETFLOW procedure, 340
supply-chain problem, 48
symbolic factorization
INTPOINT procedure, 44
syntax skeleton
INTPOINT procedure, 71
LP procedure, 178
NETFLOW procedure, 321
NLP procedure, 556

table of syntax elements, *see* functional summary, *see* functional summary, *see* functional summary, *see* functional summary
tableau
display current, 207
termination criteria, 610
absolute function convergence, 559
absolute gradient convergence, 560
absolute parameter convergence, 560
INTPOINT procedure, 89, 123
NETFLOW procedure, 367, 433
number of function calls, 569
number of iterations, 569
predicted reduction convergence, 562
relative function convergence, 562
relative gradient convergence, 564
relative parameter convergence, 577
time limit, 569
tolerance
LP procedure, 184, 186, 187, 190–192
trust region methods, 575
TYPE variable
INTPOINT procedure, 98
LP procedure, 198, 202, 236
NETFLOW procedure, 378
NLP procedure, 621–623, 625, 627, 628

unconstrained optimization, 602
upper bounds
INTPOINT procedure, 45

variables
LP procedure, 200, 224, 225, 227, 235
VF02AD algorithm, 603
VMCWD algorithm, 603

Wald confidence limits, 589, 590
warm starts
NETFLOW procedure, 313, 340, 406, 408, 414
working basis matrix
NETFLOW procedure, 332, 334, 340, 358, 397
wraparound search
NETFLOW procedure, 394

Syntax Index

ABORT statement
    NLP program statements, 591
ABSCONV= option
    PROC NLP statement, 559
ABSFCONV= option
    PROC NLP statement, 559
ABSFTOL= option, *see* ABSFCONV= option
ABSGCONV= option
    PROC NLP statement, 560, 604, 611, 618
ABSGTOL= option, *see* ABSGCONV= option
ABSTOL= option, *see* ABSCONV= option
ABSXCONV= option
    PROC NLP statement, 560
ABSXTOL= option, *see* ABSXCONV= option
ACTBC keyword
    TYPE variable (NLP), 625
ACTIVEIN= option
    PROC LP statement, 183, 217, 226
ACTIVEOUT= option
    PROC LP statement, 183, 217, 224
ALL keyword
    FDINT= option (NLP), 563
ALL option, *see* PALL option
ALLART option
    PROC NETFLOW statement, 329
AND_KEEPGOING_C= option
    PROC INTPOINT statement, 92, 125
    RESET statement (NETFLOW), 370
AND_KEEPGOING_DG= option
    PROC INTPOINT statement, 92, 125
    RESET statement (NETFLOW), 370
AND_KEEPGOING_IB= option
    PROC INTPOINT statement, 92, 125
    RESET statement (NETFLOW), 370
AND_KEEPGOING_IC= option
    PROC INTPOINT statement, 92, 125
    RESET statement (NETFLOW), 370
AND_KEEPGOING_ID= option
    PROC INTPOINT statement, 92, 125
    RESET statement (NETFLOW), 370
AND_STOP_C= option
    PROC INTPOINT statement, 90, 125
    RESET statement (NETFLOW), 368
AND_STOP_DG= option
    PROC INTPOINT statement, 90, 125
    RESET statement (NETFLOW), 368
AND_STOP_IB= option
    PROC INTPOINT statement, 90, 125
    RESET statement (NETFLOW), 368
AND_STOP_IC= option
    PROC INTPOINT statement, 91, 125
    RESET statement (NETFLOW), 369
AND_STOP_ID= option
    PROC INTPOINT statement, 91, 125
    RESET statement (NETFLOW), 369
ANY keyword
    P*x*SCAN= option (NETFLOW), 395
AOUT= option, *see* ARCOUT= option, *see* ARCOUT= option
ARCDATA keyword
    GROUPED= option (INTPOINT), 77
    GROUPED= option (NETFLOW), 333
ARCDATA= option
    PROC INTPOINT statement, 55, 56, 62, 63, 69, 74, 100
    PROC NETFLOW statement, 314, 315, 328, 380
ARCNAME statement, *see* NAME statement, *see* NAME statement
ARCOUT= option
    PROC NETFLOW statement, 315, 328, 389
    RESET statement (NETFLOW), 354
ARCS option
    PRINT statement (NETFLOW), 346
ARC_SINGLE_OBS option
    PROC INTPOINT statement, 75
    PROC NETFLOW statement, 330
ARCS_ONLY_ARCDATA option
    PROC INTPOINT statement, 75, 122
    PROC NETFLOW statement, 329, 412
ARRAY statement
    NLP procedure, 577
ASING= option, *see* ASINGULAR= option
ASINGULAR= option
    PROC NLP statement, 560, 620
AUTO option
    PROC LP statement, 186, 216, 217

BACKTRACK= option
    PROC LP statement, 186, 214
BASIC keyword
    TYPE variable (LP), 200
BASIC option
    PRINT statement (NETFLOW), 347
BEST keyword
    P*x*SCAN= option (NETFLOW), 360, 394
    Q*x*FILLSCAN= option (NETFLOW), 361, 395

BEST option
PRINT statement (LP), 194
BEST= option
PROC NLP statement, 560, 580
BFGS keyword
UPDATE= option (NLP), 576, 603
BIGM1 option
RESET statement (NETFLOW), 356
BIGM2 option
RESET statement (NETFLOW), 358
BINARY keyword
TYPE variable (LP), 200
BINFST option
PROC LP statement, 186
BLAND keyword
PRICETYPE*x*= option (NETFLOW), 360, 393, 396
BOTH keyword
CLPARM= option (NLP), 561
GROUPED= option (INTPOINT), 78
GROUPED= option (NETFLOW), 333
SCALE= option (INTPOINT), 82
SCALE= option (LP), 192, 209
SCALE= option (NETFLOW), 339
BOUNDS statement
NLP procedure, 578, 588, 600
BPD= option, *see* BYPASSDIVIDE= option, *see* BYPASSDIVIDE= option
BY statement
NLP procedure, 578
BYPASSDIV= option, *see* BYPASSDIVIDE= option, *see* BYPASSDIVIDE= option
BYPASSDIVIDE= option
PROC INTPOINT statement, 75
PROC NETFLOW statement, 330
BYTES= option
PROC INTPOINT statement, 69, 76, 122
PROC NETFLOW statement, 330, 412

CANSELECT= option
PROC LP statement, 187, 213, 216, 217
CAPAC keyword
TYPE variable (INTPOINT), 98
TYPE variable (NETFLOW), 378
CAPAC statement, *see* CAPACITY statement, *see* CAPACITY statement
CAPACITY statement
INTPOINT procedure, 92
NETFLOW procedure, 340
CD keyword
UPDATE= option (NLP), 576, 605
CDIGITS= option
PROC NLP statement, 560, 608
CENTRAL keyword
FD= option (NLP), 562
FDHESSIAN= option (NLP), 563
CF= option, *see* COREFACTOR= option
CHI keyword
FORCHI= option (NLP), 590
CHOLTINYTOL= option
PROC INTPOINT statement, 87
RESET statement (NETFLOW), 365
CLOSE keyword
VARSELECT= option (LP), 189, 215
CLPARM= option
PROC NLP statement, 561, 627
COEF statement
INTPOINT procedure, 93
LP procedure, 192
NETFLOW procedure, 341
COEFS keyword
NON_REPLIC= option (INTPOINT), 82
NON_REPLIC= option (NETFLOW), 337
COL keyword
SCALE= option (INTPOINT), 82
SCALE= option (NETFLOW), 339
COL statement
LP procedure, 192
COLUMN keyword
SCALE= option (INTPOINT), 82
SCALE= option (LP), 192, 209
SCALE= option (NETFLOW), 339
COLUMN option
PRINT statement (LP), 194
COLUMN statement
INTPOINT procedure, 93
NETFLOW procedure, 341
COMPLETE keyword
PRICETYPE= option (LP), 191, 209
CON keyword
FDINT= option (NLP), 563
SCALE= option (INTPOINT), 82
SCALE= option (NETFLOW), 338
CON_ARCS option
PRINT statement (NETFLOW), 346
CONDATA keyword
GROUPED= option (INTPOINT), 77
GROUPED= option (NETFLOW), 333
CONDATA= option
PROC INTPOINT statement, 56, 62, 63, 69, 74, 101
PROC NETFLOW statement, 314, 315, 328, 380
CONGRA keyword
TECH= option (NLP), 575, 600, 605, 611
CON_NONARCS option
PRINT statement (NETFLOW), 346
CONOPT statement
NETFLOW procedure, 341

CONOUT= option
PROC INTPOINT statement, 56, 63, 69, 74, 109
PROC NETFLOW statement, 315, 328, 389
RESET statement (NETFLOW), 355
CON_SINGLE_OBS option
PROC INTPOINT statement, 76
PROC NETFLOW statement, 330
CONST keyword
TYPE variable (NLP), 622
CONSTRAINT keyword
SCALE= option (INTPOINT), 82
SCALE= option (NETFLOW), 338
CONSTRAINTS option
PRINT statement (NETFLOW), 346
CONTROL= option
PROC LP statement, 186, 187, 216, 217
CONTYPE statement, *see* TYPE statement, *see* TYPE statement
CON_VARIABLES option
PRINT statement (NETFLOW), 346
COREFACTOR= option
PROC NETFLOW statement, 331
COST keyword
TYPE variable (INTPOINT), 98
TYPE variable (NETFLOW), 378
COST statement
INTPOINT procedure, 94
NETFLOW procedure, 342
COUT= option, *see* CONOUT= option, *see* CONOUT= option, *see* CONOUT= option
COV= option
PROC NLP statement, 561, 618
COVARIANCE= option, *see* COV= option
COVRANK keyword
TYPE variable (NLP), 626
COVSING= option
PROC NLP statement, 561, 620
COV*x* keyword
TYPE variable (NLP), 626
CRPJAC keyword
TYPE variable (NLP), 626
CRPJAC statement
NLP procedure, 579, 599, 609
CYCLEMULT1= option
RESET statement (NETFLOW), 357

DAMPSTEP= option
PROC NLP statement, 561, 614, 615
DATA= option
PROC LP statement, 183
PROC NLP statement, 561, 621
DATASETS option
SHOW statement (NETFLOW), 374
DBFGS keyword
UPDATE= option (NLP), 576, 602, 604, 606
DBLDOG keyword
TECH= option (NLP), 575, 599, 600, 604, 615
DC= option, *see* DEFCAPACITY= option, *see* DEFCAPACITY= option
DCT= option, *see* DEFCONTYPE= option, *see* DEFCONTYPE= option
DDFP keyword
UPDATE= option (NLP), 576, 602, 604, 606
DECVAR statement
NLP procedure, 580, 617
DEFCAPACITY= option
PROC INTPOINT statement, 76, 122
PROC NETFLOW statement, 331, 413
DEFCONTYPE= option
PROC INTPOINT statement, 76, 122
PROC NETFLOW statement, 332, 413
DEFCOST= option
PROC INTPOINT statement, 77, 122
PROC NETFLOW statement, 332, 413
DEFMINFLOW= option
PROC INTPOINT statement, 77, 122
PROC NETFLOW statement, 332, 413
DEFTYPE= option, *see* DEFCONTYPE= option, *see* DEFCONTYPE= option
DELTAIT= option
PROC LP statement, 187
DEMAND statement
INTPOINT procedure, 94
NETFLOW procedure, 342
DEMAND= option
PROC INTPOINT statement, 77, 122
PROC NETFLOW statement, 332, 413
DENSETHR= option
PROC INTPOINT statement, 87
RESET statement (NETFLOW), 365
DESCENDING option
BY statement (NLP), 578
DETAIL keyword
GRADCHECK= option (NLP), 564, 609
DETERMIN keyword
TYPE variable (NLP), 626
DEVEX option
PROC LP statement, 190
DF keyword
VARDEF= option (NLP), 576
DFP keyword
UPDATE= option (NLP), 576, 603
DIAHES option
PROC NLP statement, 561, 579, 581
DMF= option, *see* DEFMINFLOW= option, *see* DEFMINFLOW= option
DO statement
NLP program statements, 591

DOBJECTIVE= option
PROC LP statement, 187
DOUT= option, *see* DUALOUT= option, *see* DUALOUT= option
DS=option, *see* DAMPSTEP= option
DUALFREQ= option
RESET statement (NETFLOW), 395
DUALIN= option, *see* NODEDATA= option
DUALOUT= option
PROC LP statement, 183, 225
PROC NETFLOW statement, 315, 328, 390
RESET statement (NETFLOW), 355
DWIA= option
PROC NETFLOW statement, 332
DYNAMIC keyword
PRICETYPE= option (LP), 191, 209

ENDPAUSE option
PROC LP statement, 185, 222
ENDPAUSE1 option
RESET statement (NETFLOW), 355
EPSILON= option
PROC LP statement, 190
EQ keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 199
TYPE variable (NETFLOW), 378
TYPE variable (NLP), 622, 623, 625
ERROR keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 214
ESTDATA= option, *see* INEST= option
EVERYOBS option
NLINCON statement (NLP), 588
EXCESS= option
PROC NETFLOW statement, 332

F keyword
FORCHI= option (NLP), 590
FACT_METHOD= option
PROC INTPOINT statement, 86
RESET statement (NETFLOW), 364
FAR keyword
VARSELECT= option (LP), 189, 215
FAST keyword
GRADCHECK= option (NLP), 564, 609
FCONV= option
PROC NLP statement, 562
FCONV2= option
PROC NLP statement, 562, 604
FD= option
PROC NLP statement, 562, 607, 609
FDH= option, *see* FDHESSIAN= option
FDHES= option, *see* FDHESSIAN= option
FDHESSIAN= option
PROC NLP statement, 563, 607, 609
FDIGITS= option
PROC NLP statement, 563, 608
FDINT= option
PROC NLP statement, 563, 608, 617
FEASIBLEPAUSE option
PROC LP statement, 185, 222
FEASIBLEPAUSE1 option
RESET statement (NETFLOW), 356
FEASIBLEPAUSE2 option
RESET statement (NETFLOW), 356
FEASRATIO= option
PROFILE statement (NLP), 590
FFACTOR= option
PROFILE statement (NLP), 590
FIFO keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 213
FIRST keyword
P*x*SCAN= option (NETFLOW), 360, 394, 395
Q*x*FILLSCAN= option (NETFLOW), 361, 395
FIXED keyword
TYPE variable (LP), 199, 210
FLOW option
PROC LP statement, 184
FORCHI= option
PROFILE statement (NLP), 590
FORWARD keyword
FD= option (NLP), 562
FDHESSIAN= option (NLP), 563
FP1 option, *see* FEASIBLEPAUSE1 option
FP2 option, *see* FEASIBLEPAUSE2 option
FR keyword
UPDATE= option (NLP), 576, 605
FREE keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 200, 210
TYPE variable (NETFLOW), 378
FROM statement, *see* TAILNODE statement, *see* TAILNODE statement
FROMNODE statement, *see* TAILNODE statement, *see* TAILNODE statement
FSIZE= option
PROC NLP statement, 564
FTOL= option, *see* FCONV= option
FTOLL2= option, *see* FTOL2= option
FUTURE1 option
PROC NETFLOW statement, 406
RESET statement (NETFLOW), 361
FUTURE2 option
PROC NETFLOW statement, 406, 407
RESET statement (NETFLOW), 361
FUZZ= option

PROC LP statement, 184

G4= option
PROC NLP statement, 564, 620
GAIN keyword
TYPE variable (NETFLOW), 378
GC= option, *see* GRADCHECK= option
GCONV= option
PROC NLP statement, 564, 604, 611, 618
GCONV2= option
PROC NLP statement, 564
GE keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 199
TYPE variable (NETFLOW), 378
TYPE variable (NLP), 622, 625
GENNET option
PROC NETFLOW statement, 333
GOALPROGRAM option
PROC LP statement, 190
GRAD keyword
TYPE variable (NLP), 625
GRADCHECK= option
PROC NLP statement, 564, 609
GRADIENT statement
NLP procedure, 580, 583, 599, 609, 610
GRIDPNT keyword
TYPE variable (NLP), 625
GROUPED= option
PROC INTPOINT statement, 77, 121
PROC NETFLOW statement, 333, 411
GTOL= option, *see* GCONV= option
GTOLL2= option, *see* GTOL2= option

HEAD statement, *see* HEADNODE statement, *see* HEADNODE statement
HEADNODE statement
INTPOINT procedure, 94
NETFLOW procedure, 342
HESCAL= option
PROC NLP statement, 565, 609
HESSIAN keyword
TYPE variable (NLP), 626
HESSIAN statement
NLP procedure, 581, 599, 609
HS= option, *see* HESCAL= option
HYQUAN keyword
TECH= option (NLP), 575, 600, 606, 614

ID statement
INTPOINT procedure, 95
LP procedure, 193
NETFLOW procedure, 342
IEPSILON= option
PROC LP statement, 187

IFEASIBLEPAUSE= option
PROC LP statement, 185, 222
IFP option, *see* INFEASIBLE option
IMAXIT= option
PROC LP statement, 187
IMAXITERB= option, *see* MAXITERB= option, *see* MAXITERB= option
INCLUDE statement
NLP procedure, 570, 581, 628
INEST= option
PROC NLP statement, 565, 583, 600, 621, 622, 627
INF= option, *see* INFINITY= option, *see* INFINITY= option
INFEASIBLE option
PROC NLP statement, 565, 580
INFINITY= option
PROC INTPOINT statement, 78
PROC LP statement, 190
PROC NETFLOW statement, 334
INHESS= option, *see* INHESSIAN= option
INHESSIAN= option
PROC NLP statement, 566, 604, 616
INITIAL keyword
TYPE variable (NLP), 625
INITIAL= option
PROC NLP statement, 566
INQUAD= option
PROC NLP statement, 566, 600, 622, 623
INSTEP= option
PROC NLP statement, 566, 603, 615, 616
INTEGER keyword
TYPE variable (LP), 199, 210
INTEGER_NONZEROS option
PRINT statement (LP), 194
INTEGER option
PRINT statement (LP), 194
INTEGER_ZEROS option
PRINT statement (LP), 194
INTFIRST option
RESET statement (NETFLOW), 359
INTPOINT option
PROC NETFLOW statement, 418, 419
RESET statement (NETFLOW), 334
INTPOINT procedure, 71
CAPACITY statement, 92
COEF statement, 93
COLUMN statement, 93
COST statement, 94
DEMAND statement, 94
HEADNODE statement, 94
ID statement, 95
LO statement, 95
NAME statement, 95

NODE statement, 96
PROC INTPOINT statement, 73
QUIT statement, 96
RHS statement, 96
ROW statement, 96
RUN statement, 97
SUPDEM statement, 97
SUPPLY statement, 97
TAILNODE statement, 97
TYPE statement, 98
VAR statement, 99
INVAR= option, *see* INEST= option
INVD_2D option
PROC NETFLOW statement, 334, 398
INVFREQ= option
PROC LP statement, 190
RESET statement (NETFLOW), 358
INVTOL= option
PROC LP statement, 190
IOBJECTIVE= option
PROC LP statement, 187
IPAUSE= option
PROC LP statement, 185, 222
IPIVOT statement
LP procedure, 193, 222
IPRSLTYPE= option, *see* PRSLTYPE= option, *see* PRSLTYPE= option

JACNLC statement
NLP procedure, 582, 599
JACOBIAN keyword
TYPE variable (NLP), 626
JACOBIAN statement
NLP procedure, 582, 599, 609, 610

KEEPGOING_C= option
PROC INTPOINT statement, 91, 124
RESET statement (NETFLOW), 369
KEEPGOING_DG= option
PROC INTPOINT statement, 91, 124
RESET statement (NETFLOW), 369
KEEPGOING_IB= option
PROC INTPOINT statement, 91, 124
RESET statement (NETFLOW), 369
KEEPGOING_IC= option
PROC INTPOINT statement, 91, 124
RESET statement (NETFLOW), 369
KEEPGOING_ID= option
PROC INTPOINT statement, 91, 124
RESET statement (NETFLOW), 369

LABEL statement
NLP procedure, 583
LAGRANGE keyword
TYPE variable (NLP), 626
LB keyword
TYPE variable (NLP), 621, 622, 625
LCD= option, *see* LCDEACT= option
LCDEACT= option
PROC NLP statement, 567, 604, 613
LCE= option, *see* LCEPSILON= option
LCEPS= option, *see* LCEPSILON= option
LCEPSILON= option
PROC NLP statement, 567, 601, 604, 612
LCS= option, *see* LCSINGULAR= option
LCSING= option, *see* LCSINGULAR= option
LCSINGULAR= option
PROC NLP statement, 567, 604, 612
LE keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 199
TYPE variable (NETFLOW), 378
TYPE variable (NLP), 622, 625
LEVMAR keyword
TECH= option (NLP), 575, 600, 606, 615
LICOMP keyword
TECH= option (NLP), 575, 599, 601, 622
LIFO keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 213, 216
LIFOTYPE= option
PROC LP statement, 188
LINCON statement
NLP procedure, 600, 605
LINEAR keyword
TYPE variable (NLP), 622, 623
LINESEARCH= option
PROC NLP statement, 567, 605, 606, 614
LIS= option, *see* LINESEARCH= option
LIST option
PROC NLP statement, 568, 630
LISTCODE option
PROC NLP statement, 568, 630
LO statement
INTPOINT procedure, 95
NETFLOW procedure, 343
LONG option
PRINT statement (NETFLOW), 347
LOSS keyword
TYPE variable (NETFLOW), 378
LOW keyword
TYPE variable (INTPOINT), 98
TYPE variable (NETFLOW), 378
LOWER= option
RESET statement (LP), 196
LOWERBD keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 199
TYPE variable (NETFLOW), 378

TYPE variable (NLP), 621, 622, 625
LOWERBD statement, *see* LO statement, *see* LO statement
LP procedure, 178
COEF statement, 192
COL statement, 192
ID statement, 193
IPIVOT statement, 193
PIVOT statement, 193
PRINT statement, 193
PROC LP statement, 183
QUIT statement, 195
RANGE statement, 195
RESET statement, 196
RHS statement, 197
RHSSEN statement, 197
ROW statement, 197
RUN statement, 198
SHOW statement, 198
TYPE statement, 198
VAR statement, 200
LRATIO1 option
RESET statement (NETFLOW), 357
LRATIO2 option
RESET statement (NETFLOW), 357
LSP= option, *see* LSPRECISION= option
LSPRECISION= option
PROC NLP statement, 568, 614
LSQ statement
NLP procedure, 586

MATRIX option
PRINT statement (LP), 194
MATRIX statement
NLP procedure, 584
MAX keyword
TYPE variable (LP), 198
MAX option, *see* MAXIMIZE option, *see* MAXIMIZE option
MAX statement
NLP procedure, 586
MAXARRAYBYTES= option
PROC NETFLOW statement, 334, 412
MAXFLOW option
PROC INTPOINT statement, 78
PROC NETFLOW statement, 335
MAXFU= option, *see* MAXFUNC= option
MAXFUNC= option
PROC NLP statement, 569, 632
MAXIMIZE keyword
TYPE variable (NETFLOW), 378
MAXIMIZE option
PROC INTPOINT statement, 79
PROC NETFLOW statement, 335
MAXIT1= option
PROC LP statement, 190
RESET statement (NETFLOW), 356
MAXIT2= option
PROC LP statement, 191
RESET statement (NETFLOW), 356
MAXIT3= option
PROC LP statement, 191
MAXIT= option, *see* NLP procedure, MAXITER= option
PROC LP statement, 190
MAXITER= option
PROC NLP statement, 569, 632
MAXITERB= option
PROC INTPOINT statement, 89, 123
PROC NETFLOW statement, 433
RESET statement (NETFLOW), 367
MAXL= option
RESET statement (NETFLOW), 359
MAXLUUPDATES= option
RESET statement (NETFLOW), 359
MAXQUAD statement
NLP procedure, 586, 600
MAXSTEP= option
PROC NLP statement, 569, 615
MAXTIME= option
PROC NLP statement, 569, 632
MAZIMIZE keyword
TYPE variable (INTPOINT), 98
MEMREP option
PROC INTPOINT statement, 79, 122
PROC NETFLOW statement, 335, 412
MF option, *see* MAXFLOW option, *see* MAXFLOW option
MIN keyword
TYPE variable (LP), 198
MIN statement
NLP procedure, 586
MINBLOCK1= option
RESET statement (NETFLOW), 357
MINFLOW statement, *see* LO statement, *see* LO statement
MINIMIZE keyword
TYPE variable (INTPOINT), 98
TYPE variable (NETFLOW), 378
MINIT= option, *see* MINITER= option
MINITER= option
PROC NLP statement, 569
MINQUAD statement
NLP procedure, 586, 600
MISC option
SHOW statement (NETFLOW), 376
MLUU= option, *see* MAXLUUPDATES= option
MOD= option, *see* MODEL= option

MODEL= option
PROC NLP statement, 570, 628
MODFILE= option, *see* MODEL= option
MOREOPT option
RESET statement (NETFLOW), 361
MPSOUT= option
PROC INTPOINT statement, 74, 110, 111
PROC LP statement, 183, 203, 225
PROC NETFLOW statement, 329, 391
MSING= option, *see* MSINGULAR= option
MSINGULAR= option
PROC NLP statement, 570, 620
MULT keyword
TYPE variable (NETFLOW), 378
MULT statement
NETFLOW procedure, 343
MULTIPLIER statement, *see* MULT statement

N keyword
VARDEF= option (NLP), 576
NACTBC keyword
TYPE variable (NLP), 625
NACTLC keyword
TYPE variable (NLP), 625
NAME statement
INTPOINT procedure, 95
NETFLOW procedure, 343
NAMECTRL= option
PROC INTPOINT statement, 79
PROC NETFLOW statement, 335
NARCS= option
PROC INTPOINT statement, 81, 121
PROC NETFLOW statement, 337, 412
NCOEFS= option
PROC INTPOINT statement, 81, 121
PROC NETFLOW statement, 337, 412
NCONS= option
PROC INTPOINT statement, 81, 121
PROC NETFLOW statement, 337, 412
NEIGNEG keyword
TYPE variable (NLP), 626
NEIGPOS keyword
TYPE variable (NLP), 626
NEIGZER keyword
TYPE variable (NLP), 626
NETFLOW procedure, 321
CAPACITY statement, 340
COEF statement, 341
PROC NETFLOW statement, 328
RESET statement, 350
RHS statement, 370
ROW statement, 371
RUN statement, 371
SAVE statement, 371
SHOW statement, 373
SUPDEM statement, 377
SUPPLY statement, 377
TAILNODE statement, 377
TYPE statement, 378
VAR statement, 379
NEWRAP keyword
TECH= option (NLP), 575, 599, 601
NLC statement, *see* NLINCON statement
NLDACTLC keyword
TYPE variable (NLP), 625
NLINCON statement
NLP procedure, 588, 605
NLP procedure
ARRAY statement, 577
BOUNDS statement, 578, 588, 600
BY statement, 578
CRPJAC statement, 579, 599, 609
DECVAR statement, 580
GRADIENT statement, 580, 583, 599, 609, 610
HESSIAN statement, 581, 599, 609
INCLUDE statement, 570, 581, 628
JACNLC statement, 582, 599
JACOBIAN statement, 582, 599, 609, 610
LABEL statement, 583
LINCON statement, 584, 600, 605
LSQ statement, 586
MATRIX statement, 584
MAX statement, 586
MAXQUAD statement, 586, 600
MIN statement, 586
MINQUAD statement, 586, 600
NLINCON statement, 588, 605
PROC NLP statement, 559
PROFILE statement, 589, 627
NMSIMP keyword
TECH= option, 605
TECH= option (NLP), 575, 599, 600
NNAS= option
PROC INTPOINT statement, 81, 121
PROC NETFLOW statement, 337, 412
NNODES= option
PROC INTPOINT statement, 82, 121
PROC NETFLOW statement, 337, 412
NOAUTO option
PROC LP statement, 188
NOBIGM1 option, *see* TWOPHASE1 option
NOBIGM2 option, *see* TWOPHASE2 option
NOBINFST option
PROC LP statement, 188
NOBS keyword
TYPE variable (NLP), 625
NODE statement
INTPOINT procedure, 96

NETFLOW procedure, 344
NODEDATA= option
PROC INTPOINT statement, 55, 56, 69, 74
PROC NETFLOW statement, 314, 315, 329
NODEOUT= option
PROC NETFLOW statement, 315, 329, 390
RESET statement (NETFLOW), 355
NODEVEX option
PROC LP statement, 191
NOEIGNUM option
PROC NLP statement, 570
NOENDPAUSE option
PROC LP statement, 185
NOENDPAUSE1 option
RESET statement (NETFLOW), 356
NOEP1 option, *see* NOENDPAUSE1 option
NOFEASIBLEPAUSE option
PROC LP statement, 185
NOFEASIBLEPAUSE1 option
RESET statement (NETFLOW), 356
NOFEASIBLEPAUSE2 option
RESET statement (NETFLOW), 356
NOFLOW option
PROC LP statement, 184
NOFP1 option, *see* NOFEASIBLEPAUSE1 option
NOFP2 option, *see* NOFEASIBLEPAUSE2 option
NOFUTURE1 option
RESET statement (NETFLOW), 362
NOFUTURE2 option
RESET statement (NETFLOW), 362
NOINTFIRST option
RESET statement (NETFLOW), 360
NOLRATIO1 option
RESET statement (NETFLOW), 358
NOLRATIO2 option
RESET statement (NETFLOW), 358
NOMISS option
PROC NLP statement, 570, 621, 631
NONARC keyword
SCALE= option (INTPOINT), 82
SCALE= option (NETFLOW), 339
NONARCS option
PRINT statement (NETFLOW), 346
NONBASIC option
PRINT statement (NETFLOW), 347
NONE keyword
GRADCHECK= option (NLP), 564
GROUPED= option (INTPOINT), 78
GROUPED= option (NETFLOW), 333
NON_REPLIC= option (INTPOINT), 82
NON_REPLIC= option (NETFLOW), 337
PRICETYPE= option (LP), 191
SCALE= option (INTPOINT), 82
SCALE= option (LP), 192, 209
SCALE= option (NETFLOW), 339
TECH= option (NLP), 575
NONINTEGER_NONZEROS option
PRINT statement (LP), 194
NONINTEGER option
PRINT statement (LP), 194
NON_REPLIC= option
PROC INTPOINT statement, 82
PROC NETFLOW statement, 337
NONZERO option
PRINT statement (NETFLOW), 347
NONZEROS option
PRINT statement (LP), 195
NOP option, *see* NOPRINT option
NOPARAPRINT option
PROC LP statement, 184
NOPERTURB1 option
RESET statement (NETFLOW), 357
NOPOSTPROCESS option
PROC LP statement, 188
NOPREPROCESS option
PROC LP statement, 185
NOPRINT option
PROC LP statement, 184
PROC NLP statement, 570, 630
NOQ keyword
PRICETYPE*x*= option (NETFLOW), 360, 393, 394
NORANGEPRICE option
PROC LP statement, 189
NORANGERHS option
PROC LP statement, 189
NOSCRATCH option
RESET statement (NETFLOW), 362
NOTABLEAUPRINT option
PROC LP statement, 184
NOTSORTED option
BY statement (NLP), 578
NOTWOPHASE1 option, *see* BIGM1 option
NOTWOPHASE2 option, *see* BIGM2 option
NOUT= option, *see* NODEOUT= option
NOZTOL1 option
RESET statement (NETFLOW), 362
NOZTOL2 option
RESET statement (NETFLOW), 362
NRRIDG keyword
TECH= option (NLP), 575, 599, 602

OBJ keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 213, 216
FDINT= option (NLP), 563
OBJECTIVE keyword
TYPE variable (INTPOINT), 98

TYPE variable (NETFLOW), 378
OBJFN statement, *see* COST statement, *see* COST statement
OPTCHECK= option
PROC NLP statement, 570, 617
OPTIM_TIMER option
PROC INTPOINT statement, 82
RESET statement (NETFLOW), 362
OPTIONS option
SHOW statement (LP), 198
OTHERWISE statement
NLP program statements, 592
OUT= option
PROC NLP statement, 570, 623
OUTALL option
PROC NLP statement, 571
OUTCRPJAC option
PROC NLP statement, 571
OUTDER= option
PROC NLP statement, 571, 623
OUTEST= option
PROC NLP statement, 571, 583, 624, 627
OUTGRID option
PROC NLP statement, 571
OUTHES option, *see* OUTHESSIAN option
OUTHESSIAN option
PROC NLP statement, 571
OUTITER option
PROC NLP statement, 571
OUTJAC option
PROC NLP statement, 571
OUTM= option, *see* OUTMODEL= option
OUTMOD= option, *see* OUTMODEL= option
OUTMODEL= option
PROC NLP statement, 571, 581, 628
OUTNCJAC option
PROC NLP statement, 572
OUTTABLE option
PROFILE statement (NLP), 590
OUTTIME option
PROC NLP statement, 572
OUTVAR= option, *see* OUTEST= option

PALL option
PROC NLP statement, 572, 630
PARAMETERS statement, *see* DECVAR statement
PARAPRINT option
PROC LP statement, 184, 220, 221
PARARESTORE option
PROC LP statement, 191
PARMS keyword
TYPE variable (NLP), 621, 622, 625
PARMS statement, *see* DECVAR statement
PARTIAL keyword
PRICETYPE= option (LP), 191, 209
P*x*SCAN= option (NETFLOW), 360, 394, 395
Q*x*FILLSCAN= option (NETFLOW), 361, 395
PAUSE option
SHOW statement (NETFLOW), 375
PAUSE1= option
RESET statement (NETFLOW), 356
PAUSE2= option
RESET statement (NETFLOW), 356
PAUSE= option
PROC LP statement, 185, 222
PB keyword
UPDATE= option (NLP), 576, 605
PCOV option
PROC NLP statement, 572, 630
PCRPJAC option
PROC NLP statement, 572, 630
PDGAPTOL= option
PROC INTPOINT statement, 89, 124
PROC NETFLOW statement, 433
RESET statement (NETFLOW), 367
PDSTEPMULT= option
PROC INTPOINT statement, 87
RESET statement (NETFLOW), 366
PEIGVAL option
PROC NLP statement, 572, 620, 630
PENALTY keyword
VARSELECT= option (LP), 188, 189, 215
PENALTYDEPTH= option
PROC LP statement, 188, 215
PEPSILON= option
PROC LP statement, 186
PERROR option
PROC NLP statement, 572
PERTURB1 option
RESET statement (NETFLOW), 357
PFUNCTION option
PROC NLP statement, 572, 630
PGRID option
PROC NLP statement, 573, 630
PHASEMIX= option
PROC LP statement, 191
PHES option, *see* PHESSIAN option
PHESSIAN option
PROC NLP statement, 573, 630
PHIS option, *see* PHISTORY option
PHISTORY option
PROC NLP statement, 573, 630
PICTURE option
PRINT statement (LP), 194
PIN option, *see* PINIT option
PINIT option
PROC NLP statement, 573, 630
PIVOT statement

LP procedure, 193, 222
NETFLOW procedure, 344
PJAC option, *see* PJACOBI option
PJACOBI option
PROC NLP statement, 573, 630
PJTJ option, *see* PCRPJAC option
PL keyword
CLPARM= option (NLP), 561
PL_CL keyword
TYPE variable (NLP), 628
PLC_LOW keyword
TYPE variable (NLP), 627
PLC_UPP keyword
TYPE variable (NLP), 627
PMAXIT= option
PROC LP statement, 186, 210
PNLCJAC option
PROC NLP statement, 573, 630
POBJECTIVE= option
PROC LP statement, 188
POSTPROCESS option
PROC LP statement, 188
PR keyword
UPDATE= option (NLP), 576, 605
PREPROCESS option
PROC LP statement, 186
PRICE keyword
VARSELECT= option (LP), 189, 215
PRICE= option
PROC LP statement, 191, 208
PRICEPHI= option
PROC LP statement, 189, 195, 221, 222
PRICESEN keyword
TYPE variable (LP), 200, 219
PRICESEN option
PRINT statement (LP), 195
PRICETYPE= option
PROC LP statement, 191, 208
PRICETYPE*x*= option
RESET statement (NETFLOW), 360, 376, 393, 394, 396
PRICING option
SHOW statement (NETFLOW), 375
PRIMALIN= option
PROC LP statement, 183, 217, 226
PRIMALOUT= option
PROC LP statement, 183, 217, 224
PRINT option
PROC LP statement, 184
PRINT statement
LP procedure, 193, 222
NETFLOW procedure, 344
PRINTFREQ= option
PROC LP statement, 184
PRINTLEVEL= option
PROC LP statement, 184
PRINTLEVEL2= option
PROC INTPOINT statement, 88, 124
PROC NETFLOW statement, 433
RESET statement (NETFLOW), 367
PRIOR keyword
VARSELECT= option (LP), 189, 215
PROBLEM option
PRINT statement (NETFLOW), 346
PROC INTPOINT statement, *see* INTPOINT procedure
data set options, 74
general options, 75
PROC LP statement, *see* LP procedure
branch-and-bound control options, 186
data set options, 183
display control options, 184
interactive control options, 185
parametric control options, 189
preprocessing control options, 185
ranging control options, 189
sensitivity control options, 189
simplex algorithm control options, 190
PROC NETFLOW statement, *see* NETFLOW procedure
data set options, 328
general options, 329
PROC NLP statement, *see* NLP procedure
statement options, 559
PROFILE keyword
TYPE variable (NLP), 628
PROFILE statement
NLP procedure, 589, 627
PROJCRPJ keyword
TYPE variable (NLP), 626
PROJECT keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 213
PROJGRAD keyword
TYPE variable (NLP), 626
PROJHESS keyword
TYPE variable (NLP), 626
PROXIMITYPAUSE= option
PROC LP statement, 185, 193, 222
PRSLTYPE= option
INTPOINT procedure, 87
RESET statement (NETFLOW), 366
PSEUDOC keyword
BACKTRACK= option (LP), 186
CANSELECT= option (LP), 187, 214
VARSELECT= option (LP), 189, 215
PSH option, *see* PSHORT option
PSHORT option

PROC NLP statement, 573, 630
PSTDERR option
PROC NLP statement, 573, 630
PSUMMARY option
PROC NLP statement, 574, 630
PTIME option
PROC NLP statement, 574
PTYPE*x*= option, *see* [
PUT statement
NLP program statements, 591
PWOBJECTIVE= option
PROC LP statement, 188
P*x*NPARTIAL= option
RESET statement (NETFLOW), 360, 376, 394, 395
P*x*SCAN= option
PROC NETFLOW statement, 394
RESET statement (NETFLOW), 360, 376, 394, 395

Q keyword
PRICETYPE*x*= option (NETFLOW), 360, 393, 394
QSIZE*x*= option
RESET statement (NETFLOW), 361, 376, 394, 395
QUAD keyword
TYPE variable (NLP), 622, 623
QUADAS keyword
TECH= option (NLP), 575, 599, 601, 622
QUANEW keyword
TECH= option (NLP), 575, 599, 600, 602–604, 611, 616
QUIT statement
INTPOINT procedure, 96
LP procedure, 195, 222
NETFLOW procedure, 350
Q*x*= option, *see* [
Q*x*FILLNPARTIAL= option
RESET statement (NETFLOW), 361, 376, 395
Q*x*FILLSCAN= option
RESET statement (NETFLOW), 361, 376, 395

RANDOM= option
PROC NLP statement, 574
RANDOMPRICEMULT= option
PROC LP statement, 191
RANGE keyword
TYPE variable (LP), 200
RANGE statement
LP procedure, 195
RANGEPRICE option
PRINT statement (LP), 195
PROC LP statement, 189, 220
RANGERHS option
PRINT statement (LP), 195
PROC LP statement, 189, 220
RCHOLTINYTOL= option, *see* CHOLTINYTOL= option, *see* CHOLTINYTOL= option
RDENSETHR= option, *see* DENSETHR= option, *see* DENSETHR= option
READPAUSE option
PROC LP statement, 185, 222
REDUCEQSIZE*x*= option
RESET statement (NETFLOW), 361, 376, 395
REDUCEQ*x*= option
RESET statement (NETFLOW), 361
REFACTFREQ= option
RESET statement (NETFLOW), 360
REFRESHQ*x*= option
RESET statement (NETFLOW), 361, 376, 395
RELEVANT option
SHOW statement (NETFLOW), 376
REPSILON= option
PROC LP statement, 191
RESET statement
LP procedure, 196, 222
NETFLOW procedure, 350
REST= option, *see* RESTART= option
RESTART= option
PROC NLP statement, 574
RFF= option, *see* REFACTFREQ= option
RHS keyword
TYPE variable (INTPOINT), 98
TYPE variable (LP), 200
TYPE variable (NETFLOW), 378
RHS statement
INTPOINT procedure, 96
LP procedure, 197
NETFLOW procedure, 370
RHSOBS= option
PROC INTPOINT statement, 82
PROC NETFLOW statement, 338
RHSPHI= option
PROC LP statement, 189, 195, 220, 222
RHSSEN keyword
TYPE variable (LP), 200
RHSSEN option
PRINT statement (LP), 195
RHSSEN statement
LP procedure, 197, 218
ROW keyword
SCALE= option (INTPOINT), 82
SCALE= option (LP), 192, 209
SCALE= option (NETFLOW), 338
ROW option
PRINT statement (LP), 195
ROW statement

INTPOINT procedure, 96
LP procedure, 197
NETFLOW procedure, 371
RPDGAPTOL= option, *see* PDGAPTOL= option, *see* PDGAPTOL= option
RPDSTEPMULT= option, *see* PDSTEPMULT= option, *see* PDSTEPMULT= option
RTOLDINF= option, *see* TOLDINF= option, *see* TOLDINF= option
RTOLPINF= option, *see* TOLPINF= option, *see* TOLPINF= option
RTOLTOTDINF= option, *see* TOLTOTDINF= option, *see* TOLTOTDINF= option
RTOLTOTPINF= option, *see* TOLTOTPINF= option, *see* TOLTOTPINF= option
RTTOL= option
PROC INTPOINT statement, 89
RUN statement
INTPOINT procedure, 97
LP procedure, 198, 222
NETFLOW procedure, 371

SAME_NONARC_DATA option
PROC NETFLOW statement, 338, 412, 414
SASMPSXS macro function, 177, 204
SAVE option
QUIT statement (LP), 195
SAVE statement
NETFLOW procedure, 371
SCALE= option
PROC INTPOINT statement, 82
PROC LP statement, 192, 209
PROC NETFLOW statement, 338
SCDATA option, *see* SPARSECONDATA option, *see* SPARSECONDATA option
SCRATCH option
RESET statement (NETFLOW), 362
SE option, *see* PSTDERR option
SELECT statement
NLP program statements, 592
SENSITIVITY option
PRINT statement (LP), 194, 195, 222
SET statement, *see* RESET statement
SHORT option, *see* NLP procedure,PSHORT option
PRINT statement (NETFLOW), 347
SHORTPATH option
PROC INTPOINT statement, 83
PROC NETFLOW statement, 339
SHOW statement
LP procedure, 198, 222
NETFLOW procedure, 373
SIGSQ keyword
TYPE variable (NLP), 626
SIGSQ= option
PROC NLP statement, 574, 619
SIMPLEX option
SHOW statement (NETFLOW), 375
SING= option, *see* SINGULAR= option
SINGULAR= option
PROC NLP statement, 574
SINK= option
PROC INTPOINT statement, 83, 122
PROC NETFLOW statement, 339, 413
SINKNODE= option, *see* SINK= option, *see* SINK= option
SMALL= option
PROC LP statement, 192
SND option, *see* SAME_NONARC_DATA option
SOLUTION option
PRINT statement (LP), 195
SOME_ARCS option
PRINT statement (NETFLOW), 346
SOME_CONS option
PRINT statement (NETFLOW), 346
SOME_NONARCS option
PRINT statement (NETFLOW), 346
SOME_VARIABLES option
PRINT statement (NETFLOW), 346
SOSEQ keyword
TYPE variable (LP), 199
SOSLE keyword
TYPE variable (LP), 199
SOURCE= option
PROC INTPOINT statement, 83, 122
PROC NETFLOW statement, 339, 413
SOURCENODE= option, *see* SOURCE= option, *see* SOURCE= option
SP option, *see* SHORTPATH option, *see* SHORTPATH option
SP2 option, *see* SPARSEP2 option
SPARSECONDATA option
PROC INTPOINT statement, 83, 102
PROC NETFLOW statement, 339, 382
SPARSEDATA option
PROC LP statement, 183
SPARSEP2 option
PROC NETFLOW statement, 340
STAGE option
SHOW statement (NETFLOW), 376
STATUS option
SHOW statement (LP), 198
SHOW statement (NETFLOW), 373
STDERR keyword
TYPE variable (NLP), 625
STDERR option, *see* PSTDERR option
STOP_C= option
PROC INTPOINT statement, 89, 124
RESET statement (NETFLOW), 368

STOP_DG= option
PROC INTPOINT statement, 90, 124
RESET statement (NETFLOW), 368
STOP_IB= option
PROC INTPOINT statement, 90, 124
RESET statement (NETFLOW), 368
STOP_IC= option
PROC INTPOINT statement, 90, 124
RESET statement (NETFLOW), 368
STOP_ID= option
PROC INTPOINT statement, 90, 124
RESET statement (NETFLOW), 368
SUM option
PSUMMARY option, 574
SUMMARY option, *see* PSUMMARY option
SUMOBS option
NLINCON statement (NLP), 588
SUPDEM statement
INTPOINT procedure, 97
NETFLOW procedure, 377
SUPPLY statement
INTPOINT procedure, 97
NETFLOW procedure, 377
SUPPLY= option
PROC INTPOINT statement, 84, 122
PROC NETFLOW statement, 340, 413

TABLEAU option
PRINT statement (LP), 195, 207
TABLEAUOUT= option
PROC LP statement, 183, 225
TABLEAUPRINT option
PROC LP statement, 184, 207
TAIL statement, *see* TAILNODE statement, *see* TAILNODE statement
TAILNODE statement
INTPOINT procedure, 97
NETFLOW procedure, 377
TECH= option
PROC NLP statement, 575, 617
TECHNIQUE= option, *see* TECH= option
TERMINAT keyword
TYPE variable (NLP), 626
THRUNET option
PROC INTPOINT statement, 84, 118
PROC NETFLOW statement, 340, 405
TIME keyword
TYPE variable (NLP), 626
TIME= option
PROC LP statement, 192
TO statement, *see* HEADNODE statement, *see* HEADNODE statement
TOLDINF= option
PROC INTPOINT statement, 86
RESET statement (NETFLOW), 364
TOLPINF= option
PROC INTPOINT statement, 86
RESET statement (NETFLOW), 365
TOLTOTDINF= option
PROC INTPOINT statement, 86
RESET statement (NETFLOW), 365
TOLTOTPINF= option
PROC INTPOINT statement, 86
RESET statement (NETFLOW), 365
TONODE statement, *see* HEADNODE statement, *see* HEADNODE statement
TREETYPE= option
PROC LP statement, 188
TRUREG keyword
TECH= option (NLP), 575, 599, 601, 615
TWOPHASE1 option
RESET statement (NETFLOW), 356
TWOPHASE2 option
RESET statement (NETFLOW), 358
TYPE keyword
TYPE variable (INTPOINT), 99
TYPE variable (NETFLOW), 378
TYPE statement
INTPOINT procedure, 98
LP procedure, 198
NETFLOW procedure, 378
TYPEOBS= option
PROC INTPOINT statement, 84
PROC NETFLOW statement, 340

U= option
PROC LP statement, 192
RESET statement (NETFLOW), 360
UB keyword
TYPE variable (NLP), 621, 622, 625
UNREST keyword
TYPE variable (INTPOINT), 99
TYPE variable (NETFLOW), 378
UNRSTRT keyword
TYPE variable (LP), 199
UPD= option, *see* UPDATE= option
UPDATE= option
PROC NLP statement, 576, 601, 602
UPPCOST keyword
TYPE variable (INTPOINT), 99
TYPE variable (NETFLOW), 378
UPPER keyword
TYPE variable (INTPOINT), 99
TYPE variable (NETFLOW), 378
UPPER= option
RESET statement (LP), 196
UPPERBD keyword
TYPE variable (LP), 199, 210

TYPE variable (NLP), 621, 622, 625
UPPERBD statement, *see* CAPACITY statement, *see* CAPACITY statement

VAR statement, *see* NLP procedure,DECVAR statement
INTPOINT procedure, 99
LP procedure, 200
NETFLOW procedure, 379
VARDEF= option
PROC NLP statement, 576, 618
VARIABLES option
PRINT statement (NETFLOW), 346
VARNAME statement, *see* NAME statement, *see* NAME statement
VARSELECT= option
PROC LP statement, 188, 189, 214
VERBOSE= option
PROC INTPOINT statement, 84
RESET statement (NETFLOW), 362
VERSION= option
PROC NLP statement, 576, 603, 606, 616
VS= option, *see* VERSION= option
VSING= option, *see* VSINGULAR= option
VSINGULAR= option
PROC NLP statement, 577, 620

WALD keyword
CLPARM= option (NLP), 561
WALD_CL keyword
TYPE variable (NLP), 627
WARM option
PROC NETFLOW statement, 340, 407, 414
WHEN statement
NLP program statements, 592
WOBJECTIVE= option
PROC LP statement, 188, 189

XCONV= option
PROC NLP statement, 577
XSIZE= option
PROC NLP statement, 577
XTOL= option, *see* XCONV= option

Z1= option, *see* ZERO1= option
Z2= option, *see* ZERO2= option, *see* ZERO2= option
ZERO option
PRINT statement (NETFLOW), 347
ZERO1= option
RESET statement (NETFLOW), 363
ZERO2= option
PROC INTPOINT statement, 85
RESET statement (NETFLOW), 363
ZEROS option
PRINT statement (LP), 195
ZEROTOL= option
PROC INTPOINT statement, 85
RESET statement (NETFLOW), 364
ZTOL1 option
RESET statement (NETFLOW), 364
ZTOL2 option
RESET statement (NETFLOW), 364