SAS/OR® 12.3 User's Guide
Mathematical Programming

Contents

Acknowledgments

Credits

Documentation

Writing	Ioannis Akrotirianakis, Hao Cheng, Philipp Christophel, Matthew Galati, Dmitry V. Golovashkin, Melanie Gratton, Joshua Griffin, Jennie Hu, Tao Huang, Trevor Kearney, Zhifeng Li, Richard Liu, Amar Narisetty, Michelle Opp, Imre Pólik, Girish Ramachandra, Jack Rouse, Ben-Hao Wang, Kaihong Xu, Yan Xu, Wenwen Zhou
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, Portia Parker, Girish Ramachandra, Daniel Underwood
Technical Review	Shahrzad Azizzadeh, Tonya Chapman, Donna Fulenwider, Bill Gjertsen, Tao Huang, Edward P. Hughes, John Jasperse, Rui Kang, Charles B. Kelly, Radhika Kulkarni, Yu-Min Lin, M. Muraleetharan, Bengt Pederson, Rob Pratt, Kaihong Xu, Lois Zhu

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.

OPTMODEL	Leo Lopes, Jack Rouse
LP Simplex Algorithms	Philipp Christophel, Matthew Galati, Imre Pólik, Ben-Hao Wang, Yan Xu
LP Interior Point Algorithm	Hao Cheng

MILP Solver	Philipp Christophel, Matthew Galati, Menal Guzelsoy, Amar Narisetty, Yan Xu
NLP Solver	Joshua Griffin, Tao Huang, Wenwen Zhou
QP Solver	Hao Cheng
OPTLP	Hao Cheng, Matthew Galati, Imre Pólik, Ben-Hao Wang, Yan Xu
OPTQP	Hao Cheng, Wenwen Zhou
OPTMILP	Philipp Christophel, Amar Narisetty, Yan Xu
Decomposition Algorithm	Matthew Galati
OPTMILP Option Tuner	Ben-Hao Wang
MPS-Format SAS Data Set	Hao Cheng, Amar Narisetty
ODS Output	Philipp Christophel
Linear Algebra Specialist	Alexander Andrianov
High-performance computing foundation	Steve E. Krueger
High-Performance Analytics foundation	Robert Cohen, Georges H. Guirguis, Trevor Kearney, Richard Knight, Gang Meng, Oliver Schabenberger, Charles Shorb, Tom P. Weber
Numerical routines	Alexander Andrianov, Georges H. Guirguis

Support Groups

Software Testing	Shahrzad Azizzadeh, Tim Carter, Girija Gavankar, Dright Ho, Wei Huang, Rui Kang, Cheryl LeSaint, Yu-Min Lin, Minghui Liu, Jim McKenzie, Jim Metcalf, M. Muraleetharan, Sanjeewa Naranpanawe, Bengt Pederson, Aysegul Peker, Rob Pratt, Jennifer Sloan, Jonathan Stephenson, Kaihong Xu, Wei Zhang, 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 12.1, 12.2, and 12.3

Contents

Overview

SAS/OR 12.1 delivers a broad range of new capabilities and enhanced features, encompassing optimization, constraint programming, and discrete-event simulation. SAS/OR 12.1 enhancements significantly improve performance and expand your tool set for building, analyzing, and solving operations research models.

In previous years, SAS/OR software was updated only with new releases of Base SAS software, but this is no longer the case. This means that SAS/OR software can be released to customers when enhancements are ready, and the goal is to update SAS/OR every 12 to 18 months. To mark this newfound independence, the release numbering scheme for SAS/OR changed starting with SAS/OR 12.1. This new numbering scheme will be maintained when new versions of Base SAS and SAS/OR are shipped at the same time.

SAS/OR 12.2 is a maintenance release that does not contain any new features. SAS/OR 12.3 is another maintenance release that includes two new features that are now production, as described in the next section.

Highlights of Enhancements in SAS/OR 12.3

In SAS/OR 12.3, two important distributed-computing features become production: the option tuner for the OPTMILP procedure and the nonlinear optimization multistart algorithm for the NLP solver. The option tuner helps determine the most productive combinations of option settings for the OPTMILP procedure, and the NLP multistart algorithm is instrumental in addressing nonconvex nonlinear optimization problems.

SAS/OR 12.3 also adds the OPTLSO procedure, which performs parallel hybrid derivative-free optimization for optimization problems in which any or all of the functions involved can be nonsmooth, discontinuous, or computationally expensive to evaluate directly. The OPTLSO procedure permits both continuous and integer decision variables, and can operate in single-machine mode or distributed mode.

NOTE: Distributed mode requires SAS High-Performance Optimization.

Highlights of Enhancements in SAS/OR 12.1

Highlights of the SAS/OR enhancements include the following:

- multithreading is used to improve performance in these three areas:
    - PROC OPTMODEL problem generation
    - multistart for nonlinear optimization
    - option tuning for mixed integer linear optimization

- concurrent solve capability (experimental) for linear programming (LP) and nonlinear programming (NLP)

- improvements to all simplex LP algorithms and mixed integer linear programming (MILP) solver

- new decomposition (DECOMP) algorithm for LP and MILP

- new option for controlling MILP cutting plane strategy

- new conflict search capability for MILP solver

- option tuning for PROC OPTMILP

- new procedure, PROC OPTNET, for network optimization and analysis

- new SUBMIT block for invoking SAS code within PROC OPTMODEL

- SAS Simulation Studio improvements:
    - one-click connection of remote blocks in large models
    - autoscrolling for navigating large models
    - new search capability for block types and label content
    - alternative Experiment window configuration for large experiments
    - selective animation capability
    - new submodel component (experimental)

The CLP Procedure

In SAS/OR 12.1, the CLP procedure adds two classes of constraints that expand its capabilities and can accelerate its solution process. The LEXICO statement imposes a lexicographic ordering between pairs of variable lists. Lexicographic order is essentially analogous to alphabetical order but expands the concept to include numeric values. One vector (list) of values is lexicographically less than another if the corresponding elements are equal up to a certain point and immediately after that point the next element of the first vector is numerically less than the second. Lexicographic ordering can be useful in eliminating certain types of symmetry that can arise among solutions to constraint satisfaction problems (CSPs). 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.

Another constraint class that is added to PROC CLP for SAS/OR 12.1 is the bin-packing constraint, imposed via the PACK statement. A bin-packing constraint directs that a specified number of items must be placed into a specified number of bins, subject to the capacities (expressed in numbers of items) of the bins. The PACK statement provides a compact way to express such constraints, which can often be useful components of larger CSPs or optimization problems.

The DTREE, GANTT, and NETDRAW Procedures

In SAS/OR 12.1 the DTREE, GANTT, and NETDRAW procedures each add procedure-specific graph styles that control fonts, line colors, bar and node fill colors, and background images.

Supporting Technologies for Optimization

The underlying improvements in optimization in SAS/OR 12.1 are chiefly related to multithreading, which denotes the use of multiple computational cores to enable computations to be executed in parallel rather than serially. Multithreading can provide dramatic performance improvements for optimization because these underlying computations are performed many times in the course of an optimization process.

The underlying linear algebra operations for the linear, quadratic, and nonlinear interior point optimization algorithms are now multithreaded. The LP, QP, and NLP solvers can be used by PROC OPTMODEL, PROC OPTLP, and PROC OPTQP in SAS/OR. For nonlinear optimization with PROC OPTMODEL, the evaluation of nonlinear functions is multithreaded for improved performance.

Finally, the process of creating an optimization model from PROC OPTMODEL statements has been multithreaded. PROC OPTMODEL contains powerful declarative and programming statements and is adept at enabling data-driven definition of optimization models, with the result that a rather small section of PROC OPTMODEL code can create a very large optimization model when it is executed. Multithreading can dramatically shorten the time that is needed to create an optimization model.

In SAS/OR 12.1 you can use the NTHREADS= option in the PERFORMANCE statement in PROC OPT-MODEL and other SAS/OR optimization procedures to specify the number of cores to be used. Otherwise, SAS detects the number of cores available and uses them.

PROC OPTMODEL: Nonlinear Optimization

The nonlinear optimization solver that PROC OPTMODEL uses builds on the introduction of multithreading for its two most significant improvements in SAS/OR 12.1. First, in addition to the nonlinear solver options ALGORITHM=ACTIVESET and ALGORITHM=INTERIORPOINT, SAS/OR 12.1 introduces the ALGORITHM=CONCURRENT option (experimental), with which you can invoke both the active set and interior point algorithms for the specified problem, running in parallel on separate threads. The solution process terminates when either of the algorithms terminates. For repeated solves of a number of similarly structured problems or simply for problems for which the best algorithm isn't readily apparent, ALGORITHM=CONCURRENT should prove useful and illuminating.

Second, multithreading is central to the nonlinear optimization solver's enhanced multistart capability, which now takes advantage of multiple threads to execute optimizations from multiple starting points in parallel. The multistart capability is essential for problems that feature nonconvex nonlinear functions in either or both of the objective and the constraints because such problems might have multiple locally optimal points. Starting optimization from several different starting points helps to overcome this difficulty, and multithreading this process helps to ensure that the overall optimization process runs as fast as possible.

Linear Optimization with PROC OPTMODEL and PROC OPTLP

Extensive improvements to the primal and dual simplex linear optimization algorithms produce better performance and better integration with the crossover algorithm, which converts solutions that are found by the interior point algorithm into more usable basic optimal solutions. The crossover algorithm itself has undergone extensive enhancements that improve its speed and stability.

Paralleling developments in nonlinear optimization, SAS/OR 12.1 linear optimization introduces a concurrent algorithm, invoked with the ALGORITHM=CONCURRENT option, in the SOLVE WITH LP statement for PROC OPTMODEL or in the PROC OPTLP statement. The concurrent LP algorithm runs a selection of linear optimization algorithms in parallel on different threads, with settings to suit the problem at hand. The optimization process terminates when the first algorithm identifies an optimal solution. As with nonlinear optimization, the concurrent LP algorithm has the potential to produce significant reductions in the time needed to solve challenging problems and to provide insights that are useful when you solve a large number of similarly structured problems.

Mixed Integer Linear Optimization with PROC OPTMODEL and PROC OPTMILP

Mixed integer linear optimization in SAS/OR 12.1 builds on and extends the advances in linear optimization. Overall, solver speed has increased by over 50% (on a library of test problems) compared to SAS/OR 9.3. The branch-and-bound algorithm has approximately doubled its ability to evaluate and solve component linear optimization problems (which are referred to as nodes in the branch-and-bound tree). These improvements have significantly reduced solution time for difficult problems.

The Decomposition Algorithm

The most fundamental change to both linear and mixed integer linear optimization in SAS/OR 12.1 is the addition of the decomposition (DECOMP) algorithm, which is invoked with a specialized set of options in the SOLVE WITH LP and SOLVE WITH MILP statements for PROC OPTMODEL or in the DECOMP statement for PROC OPTLP and PROC OPTMILP. For many linear and mixed integer linear optimization problems, most of the constraints apply only to a small set of decision variables. Typically there are many such sets of constraints, complemented by a small set of linking constraints that apply to all or most of the decision variables. Optimization problems with these characteristics are said to have a "block-angular" structure, because it is easy to arrange the rows of the constraint matrix so that the nonzero values, which correspond to the local sets of constraints, appear as blocks along the main diagonal.

The DECOMP algorithm exploits this structure, decomposing the overall optimization problem into a set of component problems that can be solved in parallel on separate computational threads. The algorithm repeatedly solves these component problems and then cycles back to the overall problem to update key information that is used the next time the component problems are solved. This process repeats until it produces a solution to the complete problem, with the linking constraints present. The combination of parallelized solving of the component problems and the iterative coordination with the solution of the overall problem can greatly reduce solution time for problems that were formerly regarded as too time-consuming to solve practically.

To use the DECOMP algorithm, you must either manually or automatically identify the blocks of the constraint matrix that correspond to component problems. The METHOD= option controls the means by which blocks are identified. METHOD=USER enables you to specify the blocks yourself, using the .block suffix to declare blocks. This is by far the most common method of defining blocks. If your problem has a significant or dominant network structure, you can use METHOD=NETWORK to identify the blocks in the problem automatically. Finally, if no linking constraints are present in your problem, then METHOD=AUTO identifies the blocks automatically.

The DECOMP algorithm uses a number of detailed options that specify how the solution processes for the component problems and the overall problem are configured and how they coordinate with each other. You can also specify the number of computational threads to make available for processing component problems and the level of detail in the information to appear in the SAS log. Options specific to the linear and mixed integer linear solvers that are used by the DECOMP algorithm are largely identical to those for the respective solvers.

Setting the Cutting Plane Strategy

Cutting planes are a major component of the mixed integer linear optimization solver, accelerating its progress by removing fractional (not integer feasible) solutions. SAS/OR 12.1 adds the CUTSTRATEGY= option in the PROC OPTMILP statement and in the SOLVE WITH MILP statement for PROC OPTMODEL, enabling you to determine the aggressiveness of your overall cutting plane strategy. This option complements the individual cut class controls (CUTCLQUE=, CUTGOMORY=, CUTMIR=, and so on), with which you can enable or disable certain cut types, and the ALLCUTS= option, which enables or disables all cutting planes. In contrast, the CUTSTRATEGY= option controls cuts at a higher level, creating a profile for cutting plane use. As the cut strategy becomes more aggressive, more effort is directed toward creating cutting planes and

more cutting planes are applied. The available values of the CUTSTRATEGY= option are AUTOMATIC, BASIC, MODERATE, and AGGRESSIVE; the default is AUTOMATIC. The precise cutting plane strategy that corresponds to each of these settings can vary from problem to problem, because the strategy is also tuned to suit the problem at hand.

Conflict Search

Another means of accelerating the solution process for mixed integer linear optimization takes information from infeasible linear optimization problems that are encountered during an initial exploratory phase of the branch-and-bound process. This information is analyzed and ultimately is used to help the branch-and-bound process avoid combinations of decision variable values that are known to lead to infeasibility. This approach, known as conflict analysis or conflict search, influences presolve operations on branch-and-bound nodes, cutting planes, computation of decision variable bounds, and branching. Although the approach is complex, its application in SAS/OR 12.1 is straightforward. The CONFLICTSEARCH= option in the PROC OPTMILP statement or the SOLVE WITH MILP statement in PROC OPTMODEL enables you to specify the level of conflict search to be performed. The available values for the CONFLICTSEARCH= option are NONE, AUTOMATIC, MODERATE, and AGGRESSIVE. A more aggressive search strategy explores more branch-and-bound nodes initially before the branch-and-bound algorithm is restarted with information from infeasible nodes included. The default value is AUTOMATIC, which enables the solver to choose the search strategy.

PROC OPTMILP: Option Tuning

The final SAS/OR 12.1 improvement to the mixed integer linear optimization solver is option tuning, which helps you determine the best option settings for PROC OPTMILP. There are many options and settings available, including controls on the presolve process, branching, heuristics, and cutting planes. The TUNER statement enables you to investigate the effects of the many possible combinations of option settings on solver performance and determine which should perform best. The PROBLEMS= option enables you to submit several problems for tuning at once. The OPTIONMODE= option specifies the options to be tuned. OPTIONMODE=USER indicates that you will supply a set of options and initial values via the OPTIONVALUES= data set, OPTIONMODE=AUTO (the default) tunes a small set of predetermined options, and OPTIONMODE=FULL tunes a much more extensive option set.

Option tuning starts by using an initial set of option values to solve the problem. The problem is solved repeatedly with different option values, with a local search algorithm to guide the choices. When the tuning process terminates, the best option values are output to a data set specified by the SUMMARY= option. You can control the amount of time used by this process by specifying the MAXTIME= option. You can multithread this process by using the NTHREADS= option in the PERFORMANCE statement for PROC OPTMILP, permitting analyses of various settings to occur simultaneously.

PROC OPTMODEL: The SUBMIT Block

In SAS/OR 12.1, PROC OPTMODEL adds the ability to execute other SAS code nested inside PROC OPTMODEL syntax. This code is executed immediately after the preceding PROC OPTMODEL syntax and before the syntax that follows. Thus you can use the SUBMIT block to, for example, invoke other SAS procedures to perform analyses, to display results, or for other purposes, as an integral part of the process of creating and solving an optimization model with PROC OPTMODEL. This addition makes it even easier to integrate the operation of PROC OPTMODEL with other SAS capabilities.

To create a SUBMIT block, use a SUBMIT statement (which must appear on a line by itself) followed by the SAS code to be executed, and terminate the SUBMIT block with an ENDSUBMIT statement (which also must appear on a line by itself). The SUBMIT statement enables you to pass PROC OPTMODEL parameters, constants, and evaluated expressions to the SAS code as macro variables.

Network Optimization with PROC OPTNET

PROC OPTNET, new in SAS/OR 12.1, provides several algorithms for investigating the characteristics of networks and solving network-oriented optimization problems. A network, sometimes referred to as a graph, consists of a set of nodes that are connected by a set of arcs, edges, or links. There are many applications of network structures in real-world problems, including supply chain analysis, communications, transportation, and utilities problems. PROC OPTNET addresses the following classes of network problems:

- biconnected components

- maximal cliques

- connected components

- cycle detection

- weighted matching

- minimum-cost network flow

- minimum cut

- minimum spanning tree

- shortest path

- transitive closure

- traveling salesman

PROC OPTNET syntax provides a dedicated statement for each problem class in the preceding list.

The formats of PROC OPTNET input data sets are designed to fit network-structured data, easing the process of specifying network-oriented problems. The underlying algorithms are highly efficient and can successfully

address problems of varying levels of detail and scale. PROC OPTNET is a logical destination for users who are migrating from some of the legacy optimization procedures in SAS/OR. Former users of PROC NETFLOW can turn to PROC OPTNET to solve shortest-path and minimum-cost network flow problems, and former users of PROC ASSIGN can instead use the LINEAR_ASSIGNMENT statement in PROC OPTNET to solve assignment problems.

SAS Simulation Studio 12.1

SAS Simulation Studio 12.1, a component of SAS/OR 12.1 for Windows environments, adds several features that improve your ability to build, explore, and work with large, complex discrete-event simulation models. Large models present a number of challenges to a graphical user interface such as that of SAS Simulation Studio. Connection of model components, navigation within a model, identification of objects or areas of interest, and management of different levels of modeling are all tasks that can become more difficult as the model size grows significantly beyond what can be displayed on one screen. An indirect effect of model growth is an increased number of factors and responses that are needed to parameterize and investigate the performance of the system being modeled.

Improvements in SAS Simulation Studio 12.1 address each of these issues. In SAS Simulation Studio, you connect blocks by dragging the cursor to create links between output and input ports on regular blocks and Connector blocks. SAS Simulation Studio 12.1 automatically scrolls the display of the Model window as you drag the link that is being created from its origin to its destination, thus enabling you to create a link between two blocks that are located far apart (additionally you can connect any two blocks by clicking on the OutEntity port of the first block and then clicking on the InEntity port of the second block). Automatic scrolling also enables you to navigate a large model more easily. To move to a new area in the Model window, you can simply hold down the left mouse button and drag the visible region of the model to the desired area. This works for simple navigation and for moving a block to a new, remote location in the model.

SAS Simulation Studio 12.1 also enables you to search among the blocks in a model and identify the blocks that have a specified type, a certain character string in their label, or both. From the listing of identified blocks, you can open the Properties dialog box for each identified block and edit its settings. Thus, if you can identify a set of blocks that need similar updates, then you can make these updates without manually searching through the model for qualifying blocks and editing them individually. For very large models, this capability not only makes the update process easier but also makes it more thorough because you can identify qualifying blocks centrally.

When you design experiments for large simulation models, you often need a large number of factors to parameterize the model and a large number of responses to track system performance in sufficient detail. This was a challenge prior to SAS Simulation Studio 12.1 because the Experiment window displayed factors and responses in the header row of a table, with design points and their replications' results displayed in the rows below. A very large number of factors and responses did not fit on one screen in this display scheme, and you had to scroll across the Experiment window to view all of them.

SAS Simulation Studio 12.1 provides you with two alternative configurations for the Experiment window. The Design Matrix tab presents the tabular layout described earlier. The Design Point tab presents each design point in its own display. Factors and responses (summarized over replications) are displayed in separate tables, each with the factor or response names appearing in one column and the respective values in a second column. This layout enables a large number of factors and responses to be displayed. Response values for each replication of the design point can be displayed in a separate window.

SAS Simulation Studio 12.1 enhances its multilevel model management features by introducing the submodel component (experimental). Like the compound block, the submodel encapsulates a group of SAS Simulation Studio blocks and their connections, but the submodel outpaces the compound block in some important ways. The submodel, when expanded, opens in its own window. This means a submodel in its collapsed form can be placed close to other blocks in the Model window without requiring space for its expanded form (as is needed for compound blocks). The most important property of the submodel is its ability to be copied and instantiated in several locations simultaneously, whether in the same model, in different models in the same project, or in different projects. Each such instance is a direct reference to the original submodel, not a disconnected copy. Thus you can edit the submodel by editing any of its instances; changes that are made to any instance are propagated to all current and future instances of the submodel. This feature enables you to maintain consistency across your models and projects.

Finally, SAS Simulation Studio 12.1 introduces powerful new animation controls that should prove highly useful in debugging simulation models. In the past, animation could be switched on or off and its speed controlled, but these choices were made for the entire model. If you needed to animate a particular segment of the model, perhaps during a specific time span for the simulation clock, you had to focus your attention on that area and pay special attention when the time period of interest arrived. In SAS Simulation Studio 12.1 you can select both the area of the model to animate (by selecting a block or a compound block) and the time period over which animation should occur (by specifying the start and end times for animation). You can also control simulation speed for each such selection. Multiple selections are supported so that you can choose to animate several areas of the model, each during its defined time period and at its chosen speed.

Chapter 2
Using This Book

Contents

Purpose

SAS/OR User's Guide: Mathematical Programming provides a complete reference 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 used by PROC OPTMODEL, and the MPS-format SAS data set specification.

This chapter describes the organization of this book and the conventions 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 14 refers to other documents that contain related information.

Organization

Chapter 3, "Introduction to Optimization," contains a brief overview of the mathematical programming procedures in SAS/OR software and provides an introduction to optimization and the use of the optimization tools in the SAS System. That chapter also describes the flow of data between the procedures and how the components of the SAS System fit together.

After the introductory chapter, the next chapter describes the OPTMODEL procedure. The four subsequent chapters describe the linear programming, mixed integer linear programming, nonlinear programming, and quadratic programming solvers, which are used by the OPTMODEL procedure. The next chapter is the specification of the newly introduced MPS-format SAS data set. The last three chapters describe the new OPTLP, OPTMILP, and OPTQP procedures for solving linear programming, mixed linear programming, and quadratic programming problems, respectively. Each procedure description is self-contained; you need to be familiar with only the basic features of the SAS System and SAS terminology to use most procedures. The

statements and syntax necessary to run each procedure are presented in a uniform format throughout this book.

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 using 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 produced. You can duplicate the examples by copying the statements and data and running the SAS program. The SAS Sample Library contains the code 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

The printed version of *SAS/OR User's Guide: Mathematical Programming* 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,* 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 provide the ability 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 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: Mathematical Programming Legacy Procedures
> provides documentation 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. Guidelines are also provided on migrating from these older procedures to the newer OPTMODEL family of procedures.

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/OR Software: Project Management Examples, Version 6

contains a series of examples that illustrate how to use SAS/OR software to manage projects. Each chapter contains a complete project management scenario and describes how to use PROC GANTT, PROC CPM, and PROC NETDRAW, in addition to other reporting and graphing procedures in the SAS System, to perform the necessary project management tasks.

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

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.

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 various types depending on the sets of values that the variables are restricted to (real, integer, or binary, or a combination) and the nature of functional form of the constraints and objectives (linear, quadratic, or general nonlinear). An expression of an optimization problem in mathematical form is called a mathematical program.

When the complete description of a mathematical program is supplied to an appropriate algorithm (such as one of the solvers described in this book), the algorithm determines the optimal values for the decision variables so the objective is either maximized or minimized, the optimal values that are assigned to decision variables are on or between allowable bounds, and the constraints are obeyed. This process of solving mathematical programs is called mathematical programming, mathematical optimization, or just optimization.

When the constraints in an optimization problem are linear and the objective is either linear or quadratic, the optimization problem can be encapsulated in SAS data sets and then solved using the appropriate SAS/OR procedure: the OPTLP, OPTMILP, or OPTQP procedure.

Often optimization problems, and especially those with nonlinear elements, are formalized in an algebraic model that represents the problem. When formulated in its most abstract form, such an algebraic model is independent of problem data. A specific optimization problem instance (including the original problem) is then just an instantiation of the algebraic model with the specific data associated with that instance. An optimization modeling language (also called an algebraic modeling language) is a programming environment that has syntax, structures, and operations that enable you to express a mathematical program in a form that corresponds in a natural and transparent way to its algebraic model. The syntax, structures, and operations

also enable you to populate an algebraic model with a specific data instance and then solve the resulting optimization problem instance with an appropriate solver. The OPTMODEL procedure is such an algebraic modeling language in SAS/OR software and can be viewed as a single, unified environment to formulate and solve mathematical programming problems of many different types.

Whether mathematical programs are represented in SAS data sets or in an algebraic model in PROC OPTMODEL, they can be saved, easily changed, and solved again. The SAS/OR procedures also output SAS data sets that contain the solutions. These data sets can then be used to produce customized reports or as input to other SAS procedures. This structure enables you to use the tools of operations research and other SAS tools as building blocks to build decision support systems.

This chapter describes how to use SAS/OR software to solve a wide variety of optimization problems. It describes various types of optimization problems, indicates which SAS/OR procedures you can use, and shows how you provide data, run the procedure, and obtain optimal solutions. For additional examples that demonstrate the features of the OPTMODEL procedure, see *SAS/OR User's Guide: Mathematical Programming Examples*.

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

Linear Programming Problems

The OPTLP Procedure

The OPTLP procedure solves linear programming problems that are submitted in a SAS data set that uses a mathematical programming system (MPS) format.

The MPS file format is a format commonly used for describing linear programming (LP) and integer programming (IP) problems (Murtagh 1981; IBM 1988). MPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format SAS data set corresponds closely to the MPS file format and is used to describe linear programming problems for PROC OPTLP. For more details, see Chapter 15, "The MPS-Format SAS Data Set."

PROC OPTLP provides three solvers to solve general LPs: primal simplex, dual simplex, and interior point. The simplex solvers implement a two-phase simplex method, and the interior point solver implements a primal-dual predictor-corrector algorithm. For pure network LPs or LPs with significant network structure and additional linear side constraints, PROC OPTLP also provides a network simplex based solver. For more details about solving LPs with PROC OPTLP, see Chapter 10, "The OPTLP Procedure."

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling linear programming problems. If an LP has special network structure, the structure is typically natural and evident in a well-formulated model of the problem in PROC OPTMODEL.

Within PROC OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a linear model in data set form for use by the OPTLP procedure. For more details, see Chapter 5, "The OPTMODEL Procedure."

Mixed Integer Linear Problems

The OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs) —linear programs in which a subset of the decision variables are constrained to be integers. The OPTMILP procedure solves MILPs with an LP-based branch-and-bound algorithm augmented by advanced techniques such as cutting planes and primal heuristics. For more details about the OPTMILP procedure, see Chapter 11, "The OPTMILP Procedure."

The OPTMILP procedure requires a MILP to be specified by a SAS data set that adheres to the MPS format. See Chapter 15, "The MPS-Format SAS Data Set," for details about the MPS-format data set.

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling mixed integer linear programming problems. In fact, except for the declaration of some subset of variables to be integer or binary, modeling these problems is quite analogous to modeling LPs. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a mixed integer linear model in data set form for use by PROC OPTMILP. For more details, see Chapter 5, "The OPTMODEL Procedure."

Quadratic Programming Problems

The OPTQP Procedure

The OPTQP procedure solves quadratic programs—problems with a quadratic objective function and a collection of linear constraints, including general linear constraints along with lower or upper bounds (or both) on the decision variables.

You can specify the problem input data in one SAS data set that uses a quadratic programming system (QPS) format. For details about the QPS-format data specification, see Chapter 15, "The MPS-Format SAS Data Set." For more details about the OPTQP procedure, see Chapter 12, "The OPTQP Procedure."

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling quadratic programming problems. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a quadratic model in data set form for use by PROC OPTQP. For more details, see Chapter 5, "The OPTMODEL Procedure."

Nonlinear Problems

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling nonlinear programming problems. Within OPTMODEL you can declare a nonlinear optimization model, pass it directly to various solvers, and review the solver result. For more details, see Chapter 5, "The OPTMODEL Procedure."

You can solve many different types of nonlinear programming problems with PROC OPTMODEL using its nonlinear solver functionality. For more details about the nonlinear programming solver, see Chapter 8, "The Nonlinear Programming Solver."

Model Building with PROC OPTMODEL

Model generation and maintenance are often difficult and expensive aspects of applying mathematical programming techniques. The richly expressive syntax and features of PROC OPTMODEL, in addition to the flexible data input and output capabilities, simplify this task considerably. Although PROC OPTMODEL offers almost unlimited latitude in how a particular optimization problem is formulated, the most effective use of OPTMODEL is achieved when the model is abstracted away from the data. This aspect makes PROC OPTMODEL somewhat unusual among SAS procedures and is important enough to illustrate with a simple example.

A small product-mix problem serves as a starting point for a discussion of two different ways of modeling with PROC OPTMODEL.

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

Four processes are used to manufacture the candy:

1. Process 1 combines and cooks the basic ingredients for both chocolate 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 chocolate, and then cools and cuts the bars.

4. Process 4 is packaging: chocolate is 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 chocolate can be wrapped in 12 seconds, whereas a pound of toffee requires 50 seconds. These data are summarized as follows:

```
                     Available        Required per Pound
                     Time             chocolate    toffee
     Process         (sec)            (sec)        (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: $0.25(chocolate) + 0.75(toffee)$

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: $15(chocolate) + 40(toffee) \leq 27{,}000$

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

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

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

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

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

This linear program illustrates an example of a product mix problem. The mix of products that maximizes the objective without violating the constraints is the solution.

First, the following statements demonstrate a way of representing the optimization model in PROC OPT-MODEL that is almost a verbatim translation of the mathematical model:

```
proc optmodel;
   /* declare variables */
   var choco >= 0, toffee >= 0;

   /* maximize objective function (profit) */
   maximize profit = 0.25*choco + 0.75*toffee;
```

```
    /* subject to constraints */
    con process1:    15*choco +    40*toffee <= 27000;
    con process2:               56.25*toffee <= 27000;
    con process3: 18.75*choco                 <= 27000;
    con process4:    12*choco +    50*toffee <= 27000;

    /* solve LP using primal simplex solver */
    solve with lp / solver = primal_spx;

    /* display solution */
    print choco toffee;
quit;
```

The optimal objective value and the optimal solution are displayed in Figure 3.1:

Figure 3.1 Solution Summary

```
                    The OPTMODEL Procedure

                      Solution Summary

            Solver                              LP
            Algorithm              Primal Simplex
            Objective Function             profit
            Solution Status               Optimal
            Objective Value                   475
            Iterations                          7

            Primal Infeasibility                0
            Dual Infeasibility                  0
            Bound Infeasibility                 0


                     choco     toffee

                      1000        300
```

You can observe from the preceding example that PROC OPTMODEL provides an easy and very direct way of modeling and solving mathematical programming models. Although this way of modeling, where the data are intertwined heavily with model elements, is correct, has significant practical limitations. The model is not easy to explain, it is hard to generalize, and clearly this approach does not scale to large problems of the same similar type. To overcome these issues, you need to separate the data from the essential algebraic structure of the model. Along those lines, you can make the reasonable assumption that you have the following two data sets (one for the products and one for processes that capture the parameters and data elements of this product mix problem):

```
data Products;
   length Name $10.;
   input Name $ Profit;
datalines;
Chocolate  0.25
Toffee     0.75
;
```

```
data Processes;
   length Name $15.;
   input Name $ Available_time Chocolate Toffee;
datalines;
Cooking            27000            15            40
Color/Flavor       27000             0            56.25
Condiments         27000         18.75             0
Packaging          27000            12            50
;
```

The following alternative model in PROC OPTMODEL can solve the same problem by taking these data sets as input:

```
proc optmodel;
   /* declare sets and data indexed by sets */
   set <string> Products;
   set <string> Processes;
   num Profit{Products};
   num AvailableTime{Processes};
   num RequiredTime{Products,Processes};

   /* declare the variable */
   var Amount{Products};

   /* maximize objective function (profit) */
   maximize TotalProfit = sum{p in Products} Profit[p]*Amount[p];
   /* subject to constraints */
   con Availability{r in Processes}:
      sum{p in Products} RequiredTime[p,r]*Amount[p] <= AvailableTime[r];

   /* abstract algebraic model that captures the structure of the */
   /*    optimization problem has been defined without referring  */
   /*    to a single data constant                                */

   /* populate model by reading in the specific data instance */
   read data Products into Products=[name] Profit;
   read data Processes into Processes=[name] AvailableTime=Available_time
      {p in Products} <RequiredTime[p,name]= col(p)>;

   /* solve LP using primal simplex solver */
   solve with lp / solver = primal_spx;
   /* display solution */
   print Amount;
quit;
```

The details of the syntax and elements of the PROC OPTMODEL language are discussed in Chapter 5, "The OPTMODEL Procedure." The key observation here is that the preceding version of the PROC OPTMODEL statements capture the essence of the optimization model concisely, but completely, and the model can be explained, modified, and maintained easily. It also achieves total separation of the data from the model in that the same PROC OPTMODEL statements can be applied to any other specific problem of this type (and of any size) by simply changing the data sets appropriately and rerunning the same PROC OPTMODEL statements. Also, because of PROC OPTMODEL's ability to read data very flexibly and from any number of data sets, the problem data can be in its most natural form, making the model easier to explain and understand.

References

IBM (1988), *Mathematical Programming System Extended/370 (MPSX/370) Version 2 Program Reference Manual*, volume SH19-6553-0, Armonk, NY: IBM.

Murtagh, B. A. (1981), *Advanced Linear Programming: Computation and Practice*, New York: McGraw-Hill.

Rosenbrock, H. H. (1960), "An Automatic Method for Finding the Greatest or Least Value of a Function," *Computer Journal*, 3, 175–184.

Chapter 4
Shared Concepts and Topics

Contents

Parallel Computing

Although the speed of a single-core processor has increased considerably over the decades, further gains in computing power are possible through the use of multiple cores or processors. Using multiple cores or processors is called parallel computing. In parallel computing, certain computations are partitioned into independent smaller subcomputations, which are then processed on separate cores or processors simultaneously. Consumer-grade PCs and servers are often equipped with multicore processors, and configurations that consist of multiple processors are becoming relatively common and inexpensive. As a result, parallel computing is becoming increasingly important.

One type of parallel computing is multithreaded computing on a single machine; this execution mode is called single-machine mode. In single-machine mode, several threads use the processors of a single computer to work concurrently on subtasks. These threads share the random access memory (RAM) of that computer.

Another type of parallel computing is distributed computing, in which computation is parallelized over a number of processors on different computers. This mode is called distributed mode. In distributed mode, each computer owns an independent memory allocation, and computation can be multithreaded on each computer.

Syntax

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement is available in the OPTMODEL, OPTLP, OPTMILP, OPTQP, and OPTLSO procedures. You can use this statement to control the parallel execution of parallel features, such as the concurrent LP algorithm and the OPTMILP option tuner. For an example that demonstrates the use of the PERFORMANCE statement in the OPTMODEL procedure, see Example 8.5 in Chapter 8, "The Nonlinear Programming Solver."

The PERFORMANCE statement is available in both single-machine mode and distributed mode. This section focuses on single-machine mode. For information about the PERFORMANCE statement in distributed mode, see Chapter 2, "Shared Concepts and Topics" (*Base SAS Procedures Guide: High-Performance Procedures*).

NOTE: Distributed mode requires SAS High-Performance Optimization.

In single-machine mode, the PERFORMANCE statement enables you to control the number of threads used and the output of the ODS table that reports procedure timing. When you specify the PERFORMANCE statement, the PerformanceInfo ODS table is produced. This table lists performance characteristics such as execution mode and number of threads. You can specify the following *performance-options* in the PERFORMANCE statement:

DETAILS

Produces the Timing ODS table. This table shows a breakdown of the time used in each step of the procedure.

NTHREADS=*number*

specifies the number of threads that the procedure can use. The value of *number* can be any integer between 1 and 256 inclusive. This option overrides the SAS system option THREADS | NOTHREADS and the SAS system option CPUCOUNT=. If you do not specify the NTHREADS= option, the number of threads is determined by the SAS sytem option CPUCOUNT=.

Setting the NTHREADS= option to a number greater than the actual number of available physical cores might result in reduced performance. Specifying a high NTHREADS= value does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms in the specified procedure. In some circumstances, the procedure might use fewer threads than the specified value of the NTHREADS= option because the procedure's internal algorithms have determined that a smaller number is better.

The SAS system options THREADS | NOTHREADS and CPUCOUNT= are documented in *SAS System Options: Reference*.

PARALLELMODE=*number | string*

specifies the parallel processing mode. This option affects the solution results that are obtained from running the same model and the same option values on the same platform multiple times.

The values of *number* and the corresponding values of *string* are listed in Table 4.1.

Table 4.1   Values for PARALLELMODE= Option

number	*string*	**Description**
0	DETERMINISTIC	Requires algorithms to produce the same results every time.
1	NONDETERMINISTIC	Permits algorithms to produce different solution results. This mode requires less synchronization in parallel processing and might attain better performance than DETERMINISTIC mode.

Some procedures support only one mode; the modes that a procedure supports are detailed in its chapter.

ODS Tables

Anytime you specify the PERFORMANCE statement in a procedure, the procedure generates an ODS table called PerformanceInfo that summarizes the performance characteristics of the procedure. The information comes from the actual characteristics used and does not necessarily match the option values specified in the PERFORMANCE statement. When you specify the DETAILS option in the PERFORMANCE statement, the procedure generates an additional ODS table called Timing.

Output 4.1 shows a typical PerformanceInfo table in multithreaded computing mode.

Figure 4.1   PerformanceInfo Table

```
                 The OPTLP Procedure

              Performance Information

        Execution Mode        Single-Machine
        Number of Threads                  4
```

If you specify the NOTHREADS system option and do not specify the NTHREADS= option in the PERFORMANCE statement, then the PerformanceInfo table contains the information shown in Output 4.2.

Figure 4.2   PerformanceInfo Table: NOTHREADS Option Specified

```
                 The OPTLP Procedure

              Performance Information

        Execution Mode        Single-Machine
        Number of Threads           Disabled
```

Output 4.3 demonstrates the contents of a typical Timing table.

Figure 4.3 Timing Table

```
              Procedure Task Timing
                        Time
    Task                (sec.)     % Time

    Presolve Time        0.00       0.00%
    Solver Time          0.03      22.40%
    Wait Time            0.10      77.60%
```

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory that the SAS System uses. 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 set by the SAS System, which is sufficient for many applications. However, the solution of many realistic optimization problems can require more memory than the default. It is therefore recommended that the memory limit be increased above the default when you are solving optimization problems. This reduces the chance of a procedure failing because of an out-of-memory error.

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 that all available memory can be used, but use this setting 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 error. When a problem has millions of variables, -MEMSIZE 2G 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.

No matter how much memory is installed, 32-bit Windows operating systems permit the SAS System to use at most 4 gigabytes of memory. This memory limit might be lower, depending on which version of Windows you are running. The limit is enforced by the Windows operating system, not the SAS System.

You can specify the MEMSIZE option 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 environment.

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 environment.

Numerical Difficulties

Extremely large or extremely small numerical values might cause computational difficulties (singularities, stalled solution progress, false infeasibilities, and so on) for optimization solvers, but the occurrence of such difficulties is hard to predict. For this reason, solvers issue a data error message when they detect model data that exceed a specific threshold number. The value of the threshold number depends on your operating environment and is printed in the log as part of the data error message.

The following conditions produce a data error:

- The absolute value of an objective coefficient, constraint coefficient, or range (difference between the upper and lower bounds on a constraint) is greater than the threshold number.

- A variable's lower bound, a ≥ or = constraint's right-hand side, or a range constraint's lower bound is greater than the threshold number.

- A variable's upper bound, a ≤ or = constraint's right-hand side, or a range constraint's upper bound is smaller than the negative threshold number.

If a variable's upper bound is greater than 1E20, then solvers treats the bound as ∞. Similarly, if a variable's lower bound is less than –1E20, then LP solver treats the bound as $-\infty$.

If a solver fails or experiences numerical difficulties when solving a problem, try one of the following remedies:

- Improve the input data: Rescale very large and very small numbers in constraints, objectives, right-hand sides, and variable bounds. It is recommended that the magnitudes of the largest and smallest constraint coefficients not exceed 1E6.

- Specify different algorithms or options (or both): For example, to solve a linear program, you can choose from the primal simplex, dual simplex, interior point, and network simplex algorithms. Using available options, you can tighten or relax feasibility or optimality tolerances.

References

Andrews, G. R. (1999), *Foundations of Multithreaded, Parallel, and Distributed Programming*, Reading, MA: Addison-Wesley.

Chapter 5
The OPTMODEL Procedure

Contents

Overview: OPTMODEL Procedure

The OPTMODEL procedure includes the powerful OPTMODEL modeling language and state-of-the-art solvers for several classes of mathematical programming problems. The problems and their solvers are listed in Table 5.1.

Table 5.1 Solvers in PROC OPTMODEL

Problem	Solver
Linear programming	**LP**
Mixed integer linear programming	**MILP**
Quadratic programming	**QP**
General nonlinear programming	**NLP**

The OPTMODEL modeling language provides a modeling environment tailored to building, solving, and maintaining optimization models. This makes the process of translating the symbolic formulation of an optimization model into OPTMODEL virtually transparent since the modeling language mimics the symbolic algebra of the formulation as closely as possible. The OPTMODEL language also streamlines and simplifies the critical process of populating optimization models with data from SAS data sets. All of this transparency produces models that are more easily inspected for completeness and correctness, more easily corrected, and more easily modified, whether through structural changes or through the substitution of new data for old.

In addition to invoking optimization solvers directly with PROC OPTMODEL as already mentioned, you can use the OPTMODEL language purely as a modeling facility. You can save optimization models built with the OPTMODEL language in SAS data sets that can be submitted to other SAS/OR optimization procedures. In general, the OPTMODEL language serves as a common point of access for many of the SAS/OR optimization capabilities, whether providing both modeling and solver access or acting as a modeling interface for other optimization procedures.

For details and examples of the problems addressed and corresponding solvers, please see the dedicated chapters in this book. This chapter aims to give you a comprehensive understanding of the OPTMODEL procedure by discussing the framework provided by the OPTMODEL modeling language. For additional examples that demonstrate the features of the OPTMODEL procedure, see *SAS/OR User's Guide: Mathematical Programming Examples*.

The OPTMODEL modeling language features automatic differentiation, advanced flow control, optimization-oriented syntax (parameters, variables, arrays, constraints, objective functions), dynamic model generation, model-data separation, and transparent access to SAS data sets.

Getting Started: OPTMODEL Procedure

Optimization or mathematical programming is a search for a maximum or minimum of an *objective function* (also called a *cost function*), where search variables are restricted to particular constraints. Constraints are said to define a *feasible region* (see Figure 5.1).

Figure 5.1 Examples of Feasible Regions
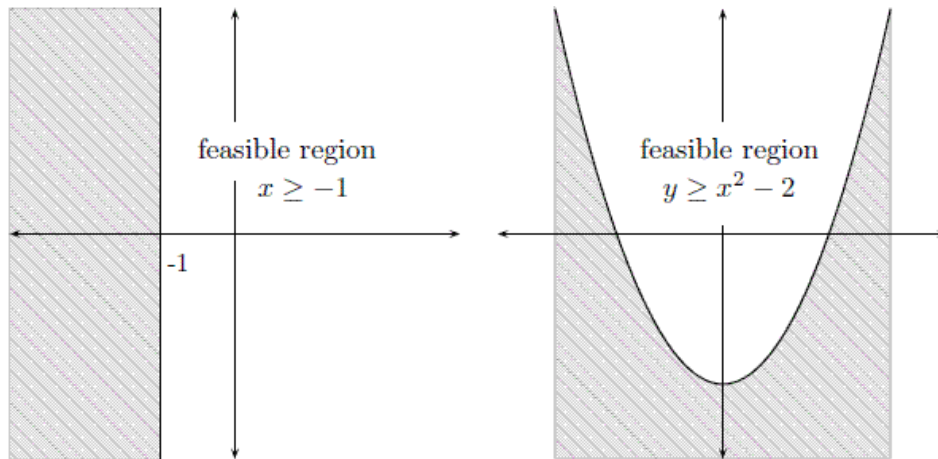
A more rigorous general formulation of such problems is as follows.

Let

$$f : S \to \mathbb{R}$$

be a real-valued function. Find x^* such that

- $x^* \in S$

- $f(x^*) \le f(x), \quad \forall x \in S$

Note that the formulation is for the minimum of f and that the maximum of f is simply the negation of the minimum of $-f$.

Here, function f is the *objective function*, and the variable in the objective function is called the optimization variable (or decision variable). S is the *feasible region*. Typically S is a subset of the Euclidean space $\mathbb{R}^n$ specified by the set of *constraints*, which are often a set of equalities ($=$) or inequalities ($\le, \ge$) that every element in S is required to satisfy simultaneously. For the special case where $S = \mathbb{R}^n$, the problem is an *unconstrained optimization*. An element x of S is called a *feasible solution* to the optimization problem, and the value $f(x)$ is called the *objective value*. A feasible solution x^* that minimizes the objective function is called an *optimal solution* to the optimization problem, and the corresponding objective value is called the *optimal value*.

In mathematics, special notation is used to denote an optimization problem. Generally, you can write an optimization problem as follows:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \end{array}$$

Normally, an empty body of constraint (the part after "subject to") implies that the optimization is un-constrained (that is, the feasible region is the whole space $\mathbb{R}^n$). The optimal solution (x^*) is denoted as

$$x^* = \underset{x \in S}{\operatorname{argmin}} f(x)$$

The optimal value ($f(x^*)$) is denoted as

$$f(x^*) = \underset{x \in S}{\min} f(x)$$

Optimization problems can be classified by the forms (linear, quadratic, nonlinear, and so on) of the functions in the objective and constraints. For example, a problem is said to be *linearly constrained* if the functions in the constraints are linear. A *linear programming* problem is a linearly constrained problem with a linear objective function. A nonlinear programming problem occurs where some function in the objective or constraints is nonlinear, and so on.

An Unconstrained Optimization Example

An unconstrained optimization problem formulation is simply

$$\text{minimize} \quad f(x)$$

For example, suppose you wanted to find the minimum value of this polynomial:

$$z(x, y) = x^2 - x - 2y - xy + y^2$$

You can compactly specify and solve the optimization problem by using the OPTMODEL modeling language. Here is the program:

```
/* invoke procedure */
proc optmodel;
   var x, y;   /* declare variables */

   /* objective function */
   min z=x**2 - x - 2*y - x*y + y**2;

   /* now run the solver */
   solve;

   print x y;
   quit;
```

This program produces the output in Figure 5.2.

Figure 5.2 Optimizing a Simple Polynomial

```
                        The OPTMODEL Procedure

                          Problem Summary

            Objective Sense              Minimization
            Objective Function                      z
            Objective Type                  Quadratic

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

            Number of Constraints                   0

            Constraint Coefficients                 0


                        Performance Information

             Execution Mode        Single-Machine
             Number of Threads                   4


                          Solution Summary

            Solver                                 QP
            Algorithm                   Interior Point
            Objective Function                      z
            Solution Status                   Optimal
            Objective Value              -2.333333333

            Primal Infeasibility                    0
            Dual Infeasibility          6.861556E-17
            Bound Infeasibility                     0
            Duality Gap                             0
            Complementarity                         0

            Iterations                              0
            Presolve Time                        0.00
            Solution Time                        0.01


                         x           y

                     1.3333     1.6667
```

In PROC OPTMODEL you specify the mathematical formulas that describe the behavior of the optimization problem that you want to solve. In the preceding example there were two independent variables in the polynomial, x and y. These are the *optimization variables* of the problem. In PROC OPTMODEL you declare optimization variables with the VAR statement. The formula that defines the quantity that you are seeking to optimize is called the *objective function*, or *objective*. The solver varies the values of the

optimization variables when searching for an optimal value for the objective.

In the preceding example the objective function is named z, declared with the MIN statement. The keyword MIN is an abbreviation for MINIMIZE. The expression that follows the equal sign (=) in the MIN statement defines the function to be minimized in terms of the optimization variables.

The VAR and MIN statements are just two of the many available PROC OPTMODEL declaration and programming statements. PROC OPTMODEL processes all such statements interactively, meaning that each statement is processed as soon as it is complete.

After PROC OPTMODEL has completed processing of declaration and programming statements, it processes the SOLVE statement, which submits the problem to a solver and prints a summary of the results. The PRINT statement displays the optimal values of the optimization variables x and y found by the solver.

It is worth noting that PROC OPTMODEL does not use a RUN statement but instead operates on an interactive basis throughout. You can continue to interact with PROC OPTMODEL even after invoking a solver. For example, you could modify the problem and issue another SOLVE statement (see the section "Model Update" on page 142).

The Rosenbrock Problem

You can use parameters to produce a clear formulation of a problem. Consider the Rosenbrock problem,

$$\text{minimize} \ \ f(x_1, x_2) = \alpha \, (x_2 - x_1^2)^2 + (1 - x_1)^2$$

where $\alpha = 100$ is a parameter (constant), x_1 and x_2 are optimization variables (whose values are to be determined), and $f(x_1, x_2)$ is an objective function.

Here is a PROC OPTMODEL program that solves the Rosenbrock problem:

```
proc optmodel;
   number alpha = 100; /* declare parameter */
   var x {1..2};        /* declare variables */
   /* objective function */
   min f = alpha*(x[2] - x[1]**2)**2 +
           (1 - x[1])**2;
   /* now run the solver */
   solve;

   print x;
   quit;
```

The PROC OPTMODEL output is shown in Figure 5.3.

Figure 5.3 Rosenbrock Function Results

```
                    The OPTMODEL Procedure

                       Problem Summary

    Objective Sense              Minimization
    Objective Function                      f
    Objective Type                  Nonlinear

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

    Number of Constraints                   0


                  Performance Information

     Execution Mode        Single-Machine
     Number of Threads                  4


                   Solution Summary

     Solver                           NLP
     Algorithm              Interior Point
     Objective Function                  f
     Solution Status               Optimal
     Objective Value         8.206033E-23

     Optimality Error        9.707102E-11
     Infeasibility                      0

     Iterations                        14
     Presolve Time                   0.00
     Solution Time                   0.01


                 [1]    x

                 1      1
                 2      1
```

A Transportation Problem

You can easily translate the symbolic formulation of a problem into the OPTMODEL procedure. Consider the transportation problem, which is mathematically modeled as the following linear programming problem:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in O, j \in D} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j \in D} x_{ij} = a_i, \quad \forall i \in O \quad \text{(SUPPLY)} \\
& \sum_{i \in O} x_{ij} = b_j, \quad \forall j \in D \quad \text{(DEMAND)} \\
& x_{ij} \geq 0, \quad \forall (i, j) \in O \times D
\end{aligned}
$$

where O is the set of origins, D is the set of destinations, c_{ij} is the cost to transport one unit from i to j, a_i is the supply of origin i, b_j is the demand of destination j, and x_{ij} is the decision variable for the amount of shipment from i to j.

Here is a very simple example. The cities in the set O of origins are Detroit and Pittsburgh. The cities in the set D of destinations are Boston and New York. The cost matrix, supply, and demand are shown in Table 5.2.

Table 5.2 A Transportation Problem

	Boston	New York	Supply
Detroit	30	20	200
Pittsburgh	40	10	100
Demand	150	150	

The problem is compactly and clearly formulated and solved by using the OPTMODEL procedure with the following statements:

```
proc optmodel;

   /* specify parameters */
   set O={'Detroit','Pittsburgh'};
   set D={'Boston','New York'};
   number c{O,D}=[30 20
                  40 10];
   number a{O}=[200 100];
   number b{D}=[150 150];
   /* model description */
   var x{O,D} >= 0;
   min total_cost = sum{i in O, j in D}c[i,j]*x[i,j];
   constraint supply{i in O}: sum{j in D}x[i,j]=a[i];
   constraint demand{j in D}: sum{i in O}x[i,j]=b[j];
   /* solve and output */
   solve;
   print x;
```

The output is shown in Figure 5.4.

Figure 5.4 Solution to the Transportation Problem

```
                   The OPTMODEL Procedure

                      Problem Summary

      Objective Sense              Minimization
      Objective Function             total_cost
      Objective Type                     Linear

      Number of Variables                     4
      Bounded Above                           0
      Bounded Below                           4
      Bounded Below and Above                 0
      Free                                    0
      Fixed                                   0

      Number of Constraints                   4
      Linear LE  (<=)                         0
      Linear EQ  (=)                          4
      Linear GE  (>=)                         0
      Linear Range                            0

      Constraint Coefficients                 8


                  Performance Information

        Execution Mode       Single-Machine
        Number of Threads                   1


                     Solution Summary

        Solver                             LP
        Algorithm                Dual Simplex
        Objective Function         total_cost
        Solution Status               Optimal
        Objective Value                  6500

        Primal Infeasibility                0
        Dual Infeasibility                  0
        Bound Infeasibility                 0

        Iterations                          0
        Presolve Time                    0.00
        Solution Time                    0.00


                        x
                                    New
                       Boston      York

            Detroit         150       50
            Pittsburgh        0      100
```

OPTMODEL Modeling Language: Basic Concepts

As seen from the examples in the previous section, a PROC OPTMODEL model consists of one or more declarations of variables, objectives, constraints, and parameters, in addition to possibly intermixed programming statements, which use the components that are created by the declarations. The declarations define the mathematical form of the problem to solve. The programming statements define data values, invoke the solver, or print the results. This section describes some basic concepts, such as variables, indices, and so on, which are used in the section "Syntax: OPTMODEL Procedure" on page 52.

Named Parameters

In the example described in the section "An Unconstrained Optimization Example" on page 37, all the numeric constants that describe the behavior of the objective function were specified directly in the objective expression. This is a valid way to formulate the objective expression. However, in many cases it is inconvenient to specify the numeric constants directly. Direct specification of numeric constants can also hide the structure of the problem that is being solved. The objective expression text would need to be modified when the numeric values in the problem change. This can be very inconvenient with large models.

In PROC OPTMODEL, you can create named numeric values that behave as constants in expressions. These named values are called *parameters*. You can write an expression by using mnemonic parameter names in place of numeric literals. This produces a clearer formulation of the optimization problem. You can easily modify the values of parameters, define them in terms of other parameters, or read them from a SAS data set.

The model from this same example can be reformulated in a more general polynomial form, as follows:

```
data coeff;
   input c_xx c_x c_y c_xy c_yy;
   datalines;
   1 -1  -2 -1 1
   ;
proc optmodel;
   var x, y;
   number c_xx, c_x, c_y, c_xy, c_yy;
   read data coeff into c_xx c_x c_y c_xy c_yy;
   min z=c_xx*x**2 + c_x*x + c_y*y + c_xy*x*y + c_yy*y**2;
   solve;
```

These statements read the coefficients from a data set, COEFF. The NUMBER statement declares the parameters. The READ DATA statement reads the parameters from the data set. You can apply this model easily to coefficients that you have generated by various means.

Indexing

Many models have large numbers of variables or parameters that can be categorized into families of similar purpose or behavior. Such families of items can be compactly represented in PROC OPTMODEL by using indexing. You can use indexing to assign each item in such families to a separate value location.

PROC OPTMODEL indexing is similar to array indexing in the DATA step, but it is more flexible. Index values can be numbers or strings, and are not required to fit into some rigid sequence. PROC OPTMODEL indexing is based on index sets, described further in the section "Index Sets" on page 52. For example, the following statement declares an indexed parameter:

```
number p{1..3};
```

The construct that follows the parameter name p, "{1..3}," is a simple index set that uses a range expression (see "Range Expression" on page 107). The index set contains the numeric members 1, 2, and 3. The parameter has distinct value locations for each of the index set members. The first such location is referenced as p[1], the second as p[2], and the third as p[3].

The following statements show an example of indexing:

```
proc optmodel;
   number p{1..3};
   p[1]=5;
   p[2]=7;
   p[3]=9;
   put p[*]=;
```

The preceding statements produce a line such as the one shown in Figure 5.5 in the log.

Figure 5.5 Indexed Parameter Output

```
p[1]=5 p[2]=7 p[3]=9
```

Index sets can also specify local dummy parameters. A dummy parameter can be used as an operand in the expressions that are controlled by the index set. For example, the assignment statements in the preceding statements could be replaced by an initialization in the parameter declaration, as follows:

```
number p{i in 1..3} init 3 + 2*i;
```

The initialization value of the parameter location p[1] is evaluated with the value of the local dummy parameter i equal to 1. So the initialization expression 3 + 2*i evaluates to 5. Similarly for location p[2], the value of i is 2 and the initialization expression evaluates to 7.

The OPTMODEL modeling language supports aggregation operators that combine values of an expression where a local dummy parameter (or parameters) ranges over the members of a set. For example, the SUM aggregation operator combines expression values by adding them together. The following statements output 21, since p[1] + p[2] + p[3] = 5 + 7 + 9 = 21:

```
proc optmodel;
   number p{i in 1..3} init 3 + 2*i;
   put (sum{i in 1..3} p[i]);
```

Aggregation operators like SUM are especially useful in objective expressions because they can combine a large number of similar expressions into a compact representation. As an example, the following statements define a trivial least squares problem:

```
proc optmodel;
   number n init 100000;
   var x{1..n};
   min z = sum{i in 1..n}(x[i] - log(i))**2;
   solve;
```

The objective function in this case is

$$z = \sum_{i=1}^{n} (x_i - \log i)^2$$

Effectively, the objective expression expands to the following large expression:

```
min z = (x[1] - log(1))**2
      + (x[2] - log(2))**2
      . . .
      + (x[99999] - log(99999))**2
      + (x[100000] - log(100000))**2;
```

Even though the problem has 100,000 variables, the aggregation operator SUM enables a compact objective expression.

NOTE: PROC OPTMODEL classifies as mathematically impure any function that returns a different value each time it is called. The RAND function, for example, falls into this category. PROC OPTMODEL disallows impure functions inside array index sets, objectives, and constraint expressions.

Types

In PROC OPTMODEL, parameters and expressions can have numeric or character values. These correspond to the elementary types named NUMBER and STRING, respectively. The NUMBER type is the same as the SAS data set numeric type. The NUMBER type includes support for missing values. The STRING type corresponds to the SAS character type, except that strings can have lengths up to a maximum of 65,534 characters (versus 32,767 for SAS character-type variables). The NUMBER and STRING types together are called the *scalar types*. You can abbreviate the type names as NUM and STR, respectively.

PROC OPTMODEL also supports set types for parameters and expressions. Sets represent collections of values of a member type, which can be a NUMBER, a STRING, or a vector of scalars (the latter is called a *tuple* and described in the following paragraphs). Members of a set all have the same member type. Members that have the same value are stored only once. For example, PROC OPTMODEL stores the set 2, 2, 2 as the set 2.

Specify a set of numbers with SET<NUMBER>. Similarly, specify a set of strings as SET<STRING>.

A set can also contain a collection of tuples, all of the same fixed length. A *tuple* is an ordered collection that contains a fixed number of elements. Each element in a tuple contains a scalar value. In PROC OPTMODEL, tuples of length 1 are equivalent to scalars. Two tuples have equal values if the elements at corresponding positions in each tuple have the same value. Within a set of tuples, the element type at a particular position in each tuple is the same for all set members. The element types are part of the set type. For example, the following statement declares `parts` as a set of tuples that have a string in the first element position and a

number in the second element position and then initializes its elements to be <R 1>, <R 2>, <C 1>, and <C 2>.

```
set<string,number> parts = /<R 1> <R 2> <C 1> <C 2>/;
```

To create a compact model, use sets to take advantage of the structure of the problem being modeled. For example, a model might contain various values that specify attributes for each member of a group of suppliers. You could create a set that contains members that represent each supplier. You can then model the attribute values by using arrays that are indexed by members of the set.

The section "Parameters" on page 46 has more details and examples.

Names

Names are used in the OPTMODEL modeling language to refer to various entities such as parameters or variables. Names must follow the usual rules for SAS names. Names can be up to 32 characters long and are not case sensitive. They must be declared before they are used.

Avoid declarations with names that begin with an underscore (_). These names can have special uses in PROC OPTMODEL.

Parameters

In the OPTMODEL modeling language, parameters are named locations that hold constant values. Parameter declarations specify the parameter type followed by a list of parameter names to declare. For example, the following statement declares numeric parameters named a and b:

```
number a, b;
```

Similarly, the following statements declare a set s of strings, a set n of numbers, and a set sn of tuples:

```
set<string> s;
set<number> n;
set<string, number> sn;
```

You can assign values to parameters in various ways. A parameter can be assigned a value with an assignment statement. For example, the following statements assign values to the parameter s, n, and sn in the preceding declaration:

```
s = {'a', 'b', 'c'};
n = {1, 2, 3};
sn = {<'a',1>, <'b',2>, <'c',3>};
```

Parameter values can also be assigned using a READ DATA statement (see the section "READ DATA Statement" on page 89).

A parameter declaration can provide an explicit value. To specify the value, follow the parameter name with an equal sign (=) and an expression. The value expression can be written in terms of other parameters. The declared parameter takes on a new value each time a parameter that is used in the expression changes. This automatic value update is shown in the following example:

```
proc optmodel;
   number pi=4*atan(1);
   number r;
   number circum=2*pi*r;
   r=1;
   put circum;        /* prints 6.2831853072 */
   r=2;
   put circum;        /* prints 12.566370614 */
```

The automatic update of parameter values makes it easy to perform "what if" analysis since, after the solver finds a solution, you can change parameters and reinvoke the solver. You can easily examine the effects of the changes on the optimal values.

If you declare a set parameter that has only the SET type specifier, then the element type is determined from the initialization expression. If the initialization expression is omitted or if the expression is an empty set, then the set type defaults to SET<NUMBER>. For example, the following statement implicitly declares s1 as a set of numbers:

```
set s1;
```

The following statement declares s2 as a set of strings:

```
set s2 = {'A'};
```

You can declare an array parameter by following the parameter name with an index set specification (see the section "Index Sets" on page 52). For example, declare an array of 10 numbers as follows:

```
number c{1..10};
```

Individual locations of a parameter array can be referred to with an indexing expression. For example, you can refer to the third location of parameter c as c[3]. Array index sets *cannot* be specified using a function such as RAND that returns a different value each time it is called.

Parameter names must be declared before they are used. Nonarray names become available at the end of the parameter declaration item. Array names become available after the index set specification. The latter case permits some forms of recursion in the optional initialization expression that can be supplied for a parameter.

You do not need to assign values to parameters before they are referenced. Most information in PROC OPTMODEL is stored symbolically and resolved when necessary. Values are resolved in certain statements. For example, PROC OPTMODEL resolves a parameter used in the objective during the execution of a SOLVE statement. If no value is available during resolution, then an error is diagnosed.

Expressions

Expressions are grouped into three categories based on the types of values they can produce: logical, set, and scalar (that is, numeric or character).

Logical expressions test for a Boolean (true or false) condition. As in the DATA step, logical operators produce a value equal to either 0 or 1. A value of 0 represents a false condition, while a value of 1 represents a true condition.

Logical expression operators are not allowed in certain contexts due to syntactic considerations. For example, in the VAR statement a logical operator might indicate the start of an option. Enclose a logical expression in parentheses to use it in such contexts. The difference is illustrated by the output (Figure 5.6) of the following statements, where two variables, x and y, are declared with initial values. The PRINT statement and the EXPAND statement are used to check the initial values and the variable bounds, respectively.

```
proc optmodel;
   var x init 0.5 >= 0 <= 1;
   var y init (0.5 >= 0) <= 1;
   print x y;
   expand;
```

Figure 5.6 Logical Expression in the VAR Statement

```
                                    x     y

                                  0.5     1



Var x >= 0 <= 1
Var y <= 1
```

Contexts that expect a logical expression also accept numeric expressions. In such cases zero or missing values are interpreted as false, and all nonzero nonmissing numeric values are interpreted as true.

Set expressions return a set value. PROC OPTMODEL supports a number of operators that create and manipulate sets. See the section "OPTMODEL Expression Extensions" on page 102 for a description of the various set expressions. Index-set syntax is described in the section "Index Sets" on page 52.

Scalar expressions are similar to the expressions in the DATA step except for PROC OPTMODEL extensions. PROC OPTMODEL provides an IF expression (described in the section "IF-THEN/ELSE Expression" on page 103). String lengths are assigned dynamically, so there is generally no padding or truncation of string values.

Table 5.3 shows the expression operators from lower to higher precedence (a higher precedence is given a larger number). Operators that have higher precedence are applied in compound expressions before operators that have lower precedence. The table also gives the order of evaluation that is applied when multiple operators of the same precedence are used together. Operators available in both PROC OPTMODEL and the DATA step have compatible precedences, except that in PROC OPTMODEL the NOT operator has a lower precedence than the relational operators. This means that, for example, NOT 1 < 2 is equal to NOT (1 < 2) (which is 0), rather than (NOT 1) < 2 (which is 1).

Table 5.3 Expression Operator Table

Precedence	Associativity	Operator	Alternates
Logic Expression Operators			
1	Left to right	`OR`	`|  !`
2	Unary	`OR{`*index-set*`}`	
		`AND{`*index-set*`}`	
3	Left to right	`AND`	`&`
4	Unary	`NOT`	`∼ ^ ¬`
5	Left to right	`<`	`LT`
		`>`	`GT`
		`<=`	`LE`
		`>=`	`GE`
		`=`	`EQ`
		`∼=`	`NE  ^= ¬=`
6	Left to right	`IN`	
		`NOT IN`	
7	Left to right	`WITHIN`	
		`NOT WITHIN`	
Set Expression Operators			
11		`IF l THEN s1 ELSE s2`	
12	Left to right	`UNION`	
		`DIFF`	
		`SYMDIFF`	
13	Unary	`UNION{`*index-set*`}`	
14	Left to right	`INTER`	
15	Unary	`INTER{`*index-set*`}`	
16	Left to right	`CROSS`	
17	Unary	`SETOF{`*index-set*`}`	
	Right to left	`..`	`TO`
		`..  e BY`	`TO e BY`
Scalar Expression Operators			
21		`IF l THEN e`	
		`IF l THEN e1 ELSE e2`	
22	Left to right	`||`	`!!`
23	Left to right	`+ -`	
24	Unary	`SUM{`*index-set*`}`	
		`PROD{`*index-set*`}`	
		`MIN{`*index-set*`}`	
		`MAX{`*index-set*`}`	
25	Left to right	`* /`	
26	Unary	`+ -`	
	Right to left	`><`	
		`<>`	
		`**`	`^`

Primary expressions are the individual operands that are combined using the expression operators. Simple primary expressions can represent constants or named parameter and variable values. More complex primary expressions can be used to call functions or construct sets.

Table 5.4   Primary Expression Table

Expression	Description
identifier-expression	Parameter/variable reference; see the section "Identifier Expressions" on page 50
name (*arg-list*)	Function call; *arg-list* is 0 or more expressions separated by commas
n	Numeric constant
. or **.***c*	Missing value constant
"*string*" or '*string*'	String constant
{ *member-list* **}**	Set constructor; *member-list* is 0 or more scalar expressions or tuple expressions separated by commas
{ *index-set* **}**	Index set expression; returns the set of all index set members
/ *members* /	Set literal expression; compactly specifies a simple set value
(*expression* **)**	Expression enclosed in parentheses
< *expr-list* **>**	Tuple expression; used with set operations; contains one or more scalar expressions separated by commas

Identifier Expressions

Use an *identifier-expression* to refer to a variable, objective, constraint, parameter or problem location in expressions or initializations. This is the syntax for *identifier-expressions*:

> *name* [**[** *expression-1* [, ... *expression-n*] **]**] [**.** *suffix*] **;**

To refer to a location in an array, follow the array *name* with a list of scalar expressions in square brackets ([ ]). The expression values are compared to the index set that was used to declare *name*. If there is more than one expression, then the values are formed into a tuple. The expression values for a valid array location must match a member of the array's index set. For example, the following statements define a parameter array A that has two valid indices that match the tuples <1,2> and <3,4>:

```
proc optmodel;
   set<number, number> ISET = {<1,2>, <3,4>};
   number A{ISET};
   a[1,2] = 0;  /* OK */
   a[3,2] = 0;  /* invalid index */
```

The first assignment is valid with this definition of the index set, but the second fails because <3,2> is not a member of the set parameter ISET.

Specify a *suffix* to refer to auxiliary locations for variables or objectives. See the section "Suffixes" on page 131 for more information.

Function Expressions

Most functions that can be invoked from the DATA step or the %SYSFUNC macro can be used in PROC OPTMODEL expressions. Certain functions are specific to the DATA step and cannot be used in PROC OPTMODEL. Functions specific to the DATA step include these:

- functions in the LAG, DIF, and DIM families

- functions that access the DATA step program data vector

- functions that access symbol attributes

The CALL statement can invoke SAS library subroutines. These subroutines can read and update the values of the parameters and variables that are used as arguments. See the section "CALL Statement" on page 68 for an example.

OPTMODEL arrays can be passed to SAS library functions and subroutines using the argument syntax:

> **OF** *array-name***[*]** [**.** *suffix*] **;**

The *array-name* is the name of an array symbol. The optional *suffix* allows auxiliary values to be referenced, as described in section "Suffixes" on page 131.

The OF argument form is resolved into a sequence of arguments, one for each index in the array. The array elements appear in order of the array's index set. The OF array form is a compact alternative to listing the array elements explicitly.

As an example, the following statements use the CALL SORTN function to sort the elements of a numeric array:

```
proc optmodel;
   number original{i in 1..8} = sin(i);
   number sorted{i in 1..8} init original[i];
   call sortn(of sorted[*]);
   print original sorted;
```

The output is shown in Figure 5.7. Eight arguments are passed to the SORTN routine. The original column shows the original order, and the sorted column has the sorted order.

Figure 5.7 Sorting Using an OF Array Argument

[1]	original	sorted
1	0.84147	−0.95892
2	0.90930	−0.75680
3	0.14112	−0.27942
4	−0.75680	0.14112
5	−0.95892	0.65699
6	−0.27942	0.84147
7	0.65699	0.90930
8	0.98936	0.98936

NOTE: OF array arguments cannot be used with function calls in declarations when any of the function arguments depend on variables, objectives, or implicit variables.

Index Sets

An index set represents a set of combinations of members from the component set expressions. The index set notation is used in PROC OPTMODEL to describe collections of valid array indices and to specify sets of values with which to perform an operation. Index sets can declare local dummy parameters and can further restrict the set of combinations by a selection expression.

In an index-set specification, the index set consists of one or more *index-set-items* that are separated by commas. Each *index-set-item* can include local dummy parameter declarations. An optional selection expression follows the list of *index-set-items*. The following syntax, which describes an index set, usually appears in braces ({ }):

> *index-set-item* [**,** ... *index-set-item*] [**:** *logic-expression*] **;**

index-set-item has these forms:

> *set-expression* **;**

> *name* **IN** *set-expression* **;**

> **<** *name-1* [**,** ... *name-n*] **>** **IN** *set-expression* **;**

Names that precede the IN keyword in *index-set-items* declare local dummy parameter names. Dummy parameters correspond to the dummy index variables in mathematical expressions. For example, the following statements output the number 385:

```
proc optmodel;
   put (sum{i in 1..10} i**2);
```

The preceding statements evaluate this summation:

$$\sum_{i=1}^{10} i^2 = 385$$

In both the statements and the summation, the index name is i.

The last form of *index-set-item* in the list can be modified to use the SLICE expression implicitly. See the section "More on Index Sets" on page 151 for details.

Array index sets cannot be defined using functions that return different values each time the functions are called. See the section "Indexing" on page 43 for details.

Syntax: OPTMODEL Procedure

PROC OPTMODEL statements are divided into three categories: the PROC statement, the declaration statements, and the programming statements. The PROC statement invokes the procedure and sets initial option values. The declaration statements declare optimization model components. The programming

statements read and write data, invoke the solver, and print results. In the following text, the statements are listed in the order in which they are grouped, with declaration statements first.

NOTE: Solver specific options are described in the individual chapters that correspond to the solvers.

PROC OPTMODEL *options* **;**

Declaration Statements:
CONSTRAINT *constraints* **;**
IMPVAR *optimization expression declarations* **;**
MAX *objective* **;**
MIN *objective* **;**
NUMBER *parameter declarations* **;**
PROBLEM *problem declaration* **;**
SET [*< types >*] *parameter declarations* **;**
STRING *parameter declarations* **;**
VAR *variable declarations* **;**

Programming Statements:
Assignment parameter = expression **;**
CALL *name* [**(** *expressions* **)**] **;**
CLOSEFILE *files* **;**
CONTINUE **;**
CREATE DATA *SAS-data-set* **FROM** *columns* **;**
DO **;** *statements* **;** **END** **;**
DO *variable* **=** *specifications* **;** *statements* **;** **END** **;**
DO UNTIL (*logic* **)** **;** *statements* **;** **END** **;**
DO WHILE (*logic* **)** **;** *statements* **;** **END** **;**
DROP *constraint* **;**
EXPAND *name* [*/ options*] **;**
FILE *file* **;**
FIX *variable* [**=** *expression*] **;**
FOR { *index-set* **}** *statement* **;**
IF *logic* **THEN** *statement* **;** [**ELSE** *statement*] **;**
LEAVE **;**
(null statement) **;**
PERFORMANCE *options* **;**
PRINT *print items* **;**
PUT *put items* **;**
QUIT **;**
READ DATA *SAS-data-set* **INTO** *columns* **;**
RESET OPTIONS *options* **;**
RESTORE *constraint* **;**
SAVE MPS *SAS-data-set* [**(OBJECTIVE | OBJ)** *name*] **;**
SAVE QPS *SAS-data-set* [**(OBJECTIVE | OBJ)** *name*] **;**
SOLVE [**WITH** *solver*] [**OBJECTIVE** *name*] [**RELAXINT**] [*/ options*] **;**
STOP **;**
SUBMIT *arguments* [*/ options*] **;**
UNFIX *variable* [**=** *expression*] **;**
USE PROBLEM *problem* **;**

Functional Summary

The statements and options available with PROC OPTMODEL are summarized by purpose in Table 5.5.

Table 5.5 Functional Summary

Description	Statement	Option
Declaration Statements:		
Declares a constraint	CONSTRAINT	
Declares optimization expressions	IMPVAR	
Declares a maximization objective	MAX	
Declares a minimization objective	MIN	
Declares a number type parameter	NUMBER	
Declares a problem	PROBLEM	
Declares a set type parameter	SET	
Declares a string type parameter	STRING	
Declares optimization variables	VAR	
Programming Statements:		
Assigns a value to a variable or parameter	=	
Invokes a library subroutine	CALL	
Closes the opened file	CLOSEFILE	
Terminates one iteration of a loop statement	CONTINUE	
Creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables	CREATE DATA	
Groups a sequence of statements together as a single statement	DO	
Executes statements repeatedly	DO (iterative)	
Executes statements repeatedly until some condition is satisfied	DO UNTIL	
Executes statements repeatedly as long as some condition is satisfied	DO WHILE	
Ignores the specified constraint	DROP	
Prints the specified constraint, variable, or objective declaration expressions after expanding aggregation operators, and so on	EXPAND	
Selects a file for the PUT statement	FILE	
Treats a variable as fixed in value	FIX	
Executes the statement repeatedly	FOR	
Executes the statement conditionally	IF	
Terminates the execution of the entire loop body	LEAVE	
Null statement	;	
Controls parallel execution	PERFORMANCE	
Outputs string and numeric data	PRINT	
Writes text data to the current output file	PUT	
Terminates the PROC OPTMODEL session	QUIT	

Description	Statement	Option
Reads data from a SAS data set into PROC OPT-MODEL parameters and variables	READ DATA	
Sets PROC OPTMODEL option values or restores them to their defaults	RESET OPTIONS	
Adds a constraint that was previously dropped back into the model	RESTORE	
Saves the structure and coefficients for a linear programming model into a SAS data set	SAVE MPS	
Saves the structure and coefficients for a quadratic programming model into a SAS data set	SAVE QPS	
Invokes a PROC OPTMODEL solver	SOLVE	
Halts the execution of all statements that contain it	STOP	
Submits SAS code for execution	SUBMIT	
Reverses the effect of FIX statement	UNFIX	
Selects the current problem	USE PROBLEM	

PROC OPTMODEL Options:

Description	Statement	Option
Specifies the accuracy for nonlinear constraints	PROC OPTMODEL	CDIGITS=
Specifies the maximum number of error messages displayed	PROC OPTMODEL	ERRORLIMIT=
Specifies the method used to approximate numeric derivatives	PROC OPTMODEL	FD=
Specifies the accuracy for the objective function	PROC OPTMODEL	FDIGITS=
Passes initial values for variables to the solver	PROC OPTMODEL	INITVAR/NOINITVAR
Specifies the tolerance for rounding the bounds on integer and binary variables	PROC OPTMODEL	INTFUZZ=
Specifies the maximum length for MPS row and column labels	PROC OPTMODEL	MAXLABLEN=
Checks missing values	PROC OPTMODEL	MISSCHECK/NOMISSCHECK
Specifies the maximum number of non-error messages displayed	PROC OPTMODEL	MSGLIMIT=
Specifies the number of digits to display	PROC OPTMODEL	PDIGITS=
Adjusts how two-dimensional array is displayed	PROC OPTMODEL	PMATRIX=
Specifies the type of presolve performed by the PROC OPTMODEL presolver	PROC OPTMODEL	PRESOLVER=
Specifies the tolerance, enabling the PROC OPT-MODEL presolver to remove slightly infeasible constraints	PROC OPTMODEL	PRESTOL=
Enables or disables printing summary	PROC OPTMODEL	PRINTLEVEL=
Specifies the width to display numeric columns	PROC OPTMODEL	PWIDTH=
Specifies the smallest difference that is permitted by the PROC OPTMODEL presolver between the upper and lower bounds of an unfixed variable	PROC OPTMODEL	VARFUZZ=

PROC OPTMODEL Statement

> **PROC OPTMODEL** [*options*] **;**

The PROC OPTMODEL statement invokes the OPTMODEL procedure. You can specify options to control how the optimization model is processed and how results are displayed. You can specify the following *options* (these options can also be specified in the RESET statement).

CDIGITS=*number*

specifies the expected number of decimal digits of accuracy for nonlinear constraints. The value can be fractional. PROC OPTMODEL uses this option to choose a step length when numeric derivative approximations are required to evaluate the Jacobian of nonlinear constraints. The default value depends on your operating environment. It is assumed that constraint values are accurate to the limits of machine precision.

See the section "Automatic Differentiation" on page 149 for more information about numeric derivative approximations.

ERRORLIMIT=*number* | **NONE**

specifies the maximum number of error messages that can be displayed. Specifying a value of *number* in the range 1 to $2^{31} - 1$ sets a specific limit. Specifying ERRORLIMIT=NONE removes any existing limit.

NOTE: Some errors abort processing immediately.

FD=FORWARD | CENTRAL

selects the method used to approximate numeric derivatives when analytic derivatives are unavailable. Most solvers require the derivatives of the objective and constraints. The methods available are as follows:

FD=FORWARD   use forward differences

FD=CENTRAL   use central differences

The default value is FORWARD. See the section "Automatic Differentiation" on page 149 for more information about numeric derivative approximations.

FDIGITS=*number*

specifies the expected number of decimal digits of accuracy for the objective function. The value can be fractional. PROC OPTMODEL uses the value to choose a step length when numeric derivatives are required. The default value depends on your operating environment. It is assumed that objective function values are accurate to the limits of machine precision.

See the section "Automatic Differentiation" on page 149 for more information about numeric derivative approximations.

INITVAR | NOINITVAR

selects whether or not to pass initial values for variables to the solver when the SOLVE statement is executed. INITVAR enables the current variable values to be passed. NOINITVAR causes the solver to be invoked without any specific initial values for variables. The INITVAR option is the default.

The LP and QP solvers always ignore initial values. The NLP solvers attempt to use specified initial values. The MILP solver uses initial values only if the PRIMALIN option is specified.

INTFUZZ=*number*

> specifies the tolerance for rounding the bounds on integer and binary variables to integer values. Bounds that differ from an integer by at most *number* are rounded to that integer. Otherwise lower boundsare rounded up to the next greater integer and upper bounds are rounded down to the next lesser integer. The value of *number* can range between 0 and 0.5. The default value is 0.00001.

MAXLABLEN=*number*

> specifies the maximum length for MPS row and column labels. The allowed range is 8 to 256, with 32 as the default. This option can also be used to control the length of row and column names displayed by solvers, such as those found in the LP solver iteration log. See also the description of the .label suffix in the section "Suffixes" on page 131.

MISSCHECK | NOMISSCHECK

> enables detailed checking of missing values in expressions. MISSCHECK requests that a message be produced each time PROC OPTMODEL evaluates an arithmetic operation or built-in function that has missing value operands (except when the operation or function specifically supports missing values). The MISSCHECK option can increase processing time. NOMISSCHECK turns off this detailed reporting. NOMISSCHECK is the default.

MSGLIMIT=*number* **| NONE**

> specifies the maximum number of non-error messages that can be displayed, including notes and warnings. Specifying a value of *number* in the range 0 to $2^{31} - 1$ sets a specific limit. Specifying MSGLIMIT=NONE removes any existing limit.

PDIGITS=*number*

> requests that the PRINT statement display *number* significant digitsfor numeric columns for which no format is specified. The value can range from 1 to 9. The default is 5.

PMATRIX=*number*

> adjusts the density evaluation of a two-dimensional array to affect how it is displayed. The value *number* scales the total number of nonempty array elements and is used by the PRINT statement to evaluate whether a two-dimensional array is "sparse" or "dense."Tables that contain a single two-dimensional array are printed in list form if they are sparse and in matrix form if they are dense. Any nonnegative value can be assigned to *number*; the default value is 1. Specifying a value for the PMATRIX= option that is less than 1 causes the list form to be used in more cases, while specifying a value greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. See the section "PRINT Statement" on page 84 for more information.

PRESOLVER=*number* **|** *string*

> specifies a presolve *string* or its corresponding value *number*, as listed in Table 5.6.

Table 5.6 Values for the PRESOLVER= Option

number	string	Description
−1	AUTOMATIC	Applies presolver using default setting.
0	NONE	Disables presolver.
1	BASIC	Performs minimal processing, only substituting fixed variables and removing empty feasible constraints.
2	MODERATE	Applies a higher level of presolve processing.
3	AGGRESSIVE	Applies the highest level of presolve processing.

The OPTMODEL presolver tightens variable bounds and eliminates redundant constraints. In general, this improves the performance of any solver. The AUTOMATIC option is intermediate between the MODERATE and AGGRESSIVE levels.

NOTE: The OPTMODEL presolver is bypassed when using the LP, QP, or MILP solvers and when saving problem data with the SAVE MPS and SAVE QPS statements.

PRESTOL=*number*

provides a tolerance so that slightly infeasible constraints can be eliminated by the OPTMODEL presolver. If the magnitude of the infeasibility is no greater than num$(|X| + 1)$, where X is the value of the original bound, then the empty constraint is removed from the presolved problem. OPTMODEL's presolver does not print messages about infeasible constraints and variable bounds when the infeasibility is within the PRESTOL tolerance. The value of PRESTOL can range between 0 and 0.1; the default value is 1E−12.

PRINTLEVEL=*number*

controls the level of listing output during a SOLVE command. The Output Delivery System (ODS) tables printed at each level are listed in Table 5.7. Some solvers can produce additional tables; see the individual solver chapters for more information.

Table 5.7 Values for the PRINTLEVEL= Option

number	**Description**
0	Disables all tables.
1	Prints Problem Summary and Solution Summary.
2	Prints Problem Summary, Solution Summary, Methods of Derivative Computation (for NLP solvers), Solver Options, Optimization Statistics, and solver-specific ODS tables.

For more details about the ODS tables produced by PROC OPTMODEL, see the section "ODS Table and Variable Names" on page 122.

PWIDTH=*number*

sets the width used by the PRINT statement to display numeric columns when no format is specified. The smallest value *number* can take is the value of the PDIGITS= option plus 7; the largest value *number* can take is 16. The default value is equal to the value of the PDIGITS= option plus 7.

VARFUZZ=*number*

specifies the smallest difference that is permitted by the OPTMODEL presolver between the upper and lower bounds of an unfixed variable. If the difference is smaller than *number*, then the variable is fixed to the average of the upper and lower bounds before it is presented to the solver. Any nonnegative value can be assigned to *number*; the default value is 0.

Declaration Statements

The declaration statements define the parameters, variables, constraints, and objectives that describe a PROC OPTMODEL optimization model. Declarations in the PROC OPTMODEL input are saved for later use.

Unlike programming statements, declarations cannot be nested in other statements. Declaration statements are terminated by a semicolon.

Many declaration attributes, such as variable bounds, are defined using expressions. Expressions in declarations are handled symbolically and are resolved as needed. In particular, expressions are generally reevaluated when one of the parameter values they use has been changed.

CONSTRAINT Declaration

CONSTRAINT *constraint* [**,** . . . *constraint*] **;**

CON *constraint* [**,** . . . *constraint*] **;**

The constraint declaration defines one or more constraints on expressions in terms of the optimization variables. You can specify multiple constraint declaration statements.

Constraints can have an upper bound, a lower bound, or both bounds. The allowed forms are as follows:

[*name* [**{** *index-set* **}**] **:**] *expression* **=** *expression*
> declares an equality constraint or, when an *index-set* is specified, a family of equality constraints. The solver attempts to assign values to the optimization variables to make the two expressions equal.

[*name* [**{** *index-set* **}**] **:**] *expression relation expression*
> declares an inequality constraint that has a single upper or lower bound. *index-set* declares a family of inequality constraints. *relation* is the <= or >= operator. When *relation* is the <= operator, the solver tries to assign optimization variable values so that the left *expression* has a value less than or equal to the right *expression*. When *relation* is the >= operator, the solver tries to assign optimization variable values so that the left *expression* has a value greater than or equal to the right *expression*.

[*name* [**{** *index-set* **}**] **:**] *bound relation body relation bound*
> declares an inequality constraint that is bounded on both sides, called a range constraint. *index-set* declares a family of range constraints. *relation* is the <= or >= operator. The same operator must be used in both positions. The first *bound* expression defines the lower bound (if the <= operator is used) or the upper bound (if the >= operator is used). The second *bound* defines the upper bound (if the <= operator is used) or the lower bound (if the >= operator is used). The solver tries to assign optimization variables so that the value of the *body* expression is in the range between the upper and lower bounds.

name defines the name for the constraint. Use the name to reference constraint attributes, such as the bounds, elsewhere in the PROC OPTMODEL model. If no name is provided, then a default name is created of the form _ACON_[*n*], where *n* is an integer. See the section "Constraints" on page 126 for more information.

Here is a simple example that defines a constraint with a lower bound:

```
proc optmodel;
   var x, y;
   number low;
   con a: x+y >= low;
```

The following example adds an upper bound:

```
var x, y;
number low;
con a: low <= x+y <= low+10;
```

Indexed families of constraints can be defined by specifying an *index-set* after the name. Any dummy parameters that are declared in the *index-set* can be referenced in the expressions that define the constraint. A particular member of an indexed family can be specified by using an *identifier-expression* with a bracketed index list, in the same fashion as array parameters and variables. For example, the following statements create an indexed family of constraints named incr:

```
proc optmodel;
   number n;
   var x{1..n}
   /* require nondecreasing x values */
   con incr{i in 1..n-1}: x[i+1] >= x[i];
```

The CON statement in the example creates constraints incr[1] through incr[$n-1$].

Constraint expressions cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 43 for details.

IMPVAR Declaration

> **IMPVAR** *impvar-decl* [, ... *impvar-decl*] ;

The IMPVAR statement declares one or more names that refer to optimization expressions in the model. The declared name is called an implicit variable. An implicit variable is useful for structuring models so that complex expressions do not need to be repeated each time they are used. The value of an implicit variable needs to be computed only once instead of at each place where the original expression is used, which helps reduce computational overhead. Implicit variables are evaluated without intervention from the solver.

Multiple IMPVAR statements are allowed. The names of implicit variables must be distinct from other model declarations, such as variables and constraints. Implicit variables can be used in model expressions in the same places where ordinary variables are allowed.

This is the syntax for an *impvar-decl*:

> *name* [{ *index-set* }] = *expression* ;

Each *impvar-decl* declares a name for an implicit variable. The name can be followed by an *index-set* specification to declare a family of implicit variables. The *expression* that the name refers to follows. Dummy parameters that are declared in the *index-set* specification can be used in the expression. The *expression* can refer to other model components, including variables, the current implicit variable, and other implicit variables.

As an example, in the following model statements the implicit variable total_weight is used in multiple constraints to set a limit on various product quantities, represented by locations in array x:

```
impvar total_weight = sum{p in PRODUCTS} Weight[p]*x[p];

con prod1_limit: Weight['Prod1'] * x['Prod1'] <= 0.3 * total_weight;
con prod2_limit: Weight['Prod2'] * x['Prod2'] <= 0.25 * total_weight;
```

MAX and MIN Objective Declarations

> **MAX** *name* [**{** *index-set* **}**] **=** *expression* **;**

> **MIN** *name* [**{** *index-set* **}**] **=** *expression* **;**

The MAX or MIN declaration specifies an objective for the solver. The *name* names the objective function for later reference. When a non-array objective declaration is read, the declaration becomes the new objective of the current problem, replacing any previous objective. The solver maximizes an objective that is specified with the MAX keyword and minimizes an objective that is specified with the MIN keyword. An objective is not allowed to have the same name as a parameter or variable. Multiple objectives are permitted, but the solver processes only one objective at a time.

expression specifies the numeric function to maximize or minimize in terms of the optimization-variables. Specify an *index-set* to declare a family of objectives. Dummy parameters declared in the *index-set* specification can be used in the following expression.

Objectives can also be used as implicit variables. When used in an expression, an objective name refers to the current value of the named objective function. The value of an unsuffixed objective name can depend on the value of optimization variables, so objective names cannot be used in constant expressions such as variable bounds. You can reference objective names in objective or constraint expressions. For example, the following statements declare two objective names, q and l, which are immediately referred to in the objective declaration of z and the declarations of the constraints.

```
proc optmodel;
   var x, y;
   min q=(x+y)**2;
   max l=x+2*y;
   min z=q+l;
   con c1: q<=4;
   con c2: l>=2;
```

Objectives cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 43 for details.

NUMBER, STRING, and SET Parameter Declarations

> **NUMBER** *parameter-decl* [**,** … *parameter-decl*] **;**

> **STRING** *parameter-decl* [**,** … *parameter-decl*] **;**

> **SET** [**<** *scalar-type***,** … *scalar-type* **>**] *parameter-decl* [**,** … *parameter-decl*] **;**

Parameters provide names for constants. Parameters are declared by specifying the parameter type followed by a list of parameter names. Declarations of parameters that have NUMBER or STRING types start with a *scalar-type* specification:

> **NUMBER** | **NUM** **;**

> **STRING** | **STR** **;**

The NUM and STR keywords are abbreviations for the NUMBER and STRING keywords, respectively.

The declaration of a parameter that has the set type begins with a *set-type* specification:

> **SET** [**<** *scalar-type***,** ... *scalar-type* **>**] **;**

In a *set-type* declaration, the SET keyword is followed by a list of *scalar-type* items that specify the member type. A set with scalar members is specified with a single *scalar-type* item. A set with tuple members has a *scalar-type* item for each tuple element. The *scalar-type* items specify the types of the elements at each tuple position.

If the SET keyword is not followed by a list of *scalar-type* items, then the set type is determined from the type of the initialization expression. The declared type defaults to SET<NUMBER> if no initialization expression is given or if the expression type cannot be determined.

For any parameter type, the type declaration is followed by a list of *parameter-decl* items that specify the names of the parameters to declare. In a *parameter-decl* item the parameter name can be followed by an optional index specification and any necessary options, as follows:

> *name* [**{** *index-set* **}**] [*parameter-options*] **;**

The parameter *name* and *index-set* can be followed by a list of *parameter-options*. Dummy parameters declared in the *index-set* can be used in the *parameter-options*. The parameter options can be specified with the following forms:

= *expression*

> provides an explicit value for each parameter location. In this case the parameter acts like an alias for the *expression* value.

INIT *expression*

> specifies a default value that is used when a parameter value is required but no other value has been supplied. For example:

```
number n init 1;
set s init {'a', 'b', 'c'};
```

> PROC OPTMODEL evaluates the expression for each parameter location the first time the parameter needs to be resolved. The expression is not used when the parameter already has a value.

= [*initializers* **]**

> provides a compact means to define the values for an array, in which each array location value can be individually specified by the *initializers*.

INIT [*initializers* **]**

> provides a compact means to define multiple default values for an array. Each array location value can be individually specified by the *initializers*. With this option the array values can still be updated outside the declaration.

The =*expression* parameter option defines a parameter value by using a formula. The formula can refer to other parameters. The parameter value is updated when the referenced parameters change. The following example shows the effects of the update:

```
proc optmodel;
   number n;
```

```
set<number> s = 1..n;
number a{s};
n = 3;
a[1] = 2;     /* OK */
a[7] = 19;    /* error, 7 is not in s */
n = 10;
a[7] = 19;    /* OK now */
```

In the preceding example the value of set s is resolved for each use of array a that has an index. For the first use of a[7], the value 7 is not a member of the set s. However, the value 7 is a member of s at the second use of a[7].

The INIT *expression* parameter option specifies a default value for a parameter. The following example shows the usage of this option:

```
proc optmodel;
   num a{i in 1..2} init i**2;
   a[1] = 2;
   put a[*]=;
```

When the value of a parameter is needed but no other value has been supplied, the default value specified by INIT *expression* is used, as shown in Figure 5.8.

Figure 5.8 INIT Option: Output

```
a[1]=2 a[2]=4
```

NOTE: Parameter values can also be read from files or specified with assignment statements. However, the value of a parameter that is assigned with the =*expression* or =[*initializers*] forms can be changed only by modifying the parameters used in the defining expressions. Parameter values specified by the INIT option can be reassigned freely.

Initializing Arrays

Arrays can be initialized with the =[*initializers*] or INIT [*initializers*] forms. These forms are convenient when array location values need to be individually specified. The forms behave the same way, except that the INIT [*initializers*] form allows the array values to be modified after the declaration. These forms of initialization are used in the following statements:

```
proc optmodel;
   number a{1..3} = [5 4 7];
   number b{1..3} INIT [5 4 7];
   put a[*]=;
   b[1] = 1;
   put b[*]=;
```

Each array location receives a different value, as shown in Figure 5.9. The displayed values for b are a combination of the default values from the declaration and the assigned value in the statements.

Figure 5.9 Array Initialization

```
a[1]=5 a[2]=4 a[3]=7
b[1]=1 b[2]=4 b[3]=7
```

Each *initializer* takes the following form:

[**[** *index* **]**] *value* ;

The *value* specifies the value of an array location and can be a numeric or string constant, a set literal, or an expression enclosed in parentheses.

In array initializers, string constants can be specified using quoted strings. When the string text follows the rules for a SAS name, the text can also be specified without quotation marks. String constants that begin with a digit, contain blanks, or contain other special characters must be specified with a quoted string.

As an example, the following statements define an array parameter that could be used to map numeric days of the week to text strings:

```
proc optmodel;
   string dn{1..5} =
          [Monday Tuesday Wednesday Thursday Friday];
```

The optional *index* in square brackets specifies the index of the array location to initialize. The index specifies one or more numeric or string subscripts. The subscripts allow the same syntactic forms as the *value* items. Commas can be used to separate index subscripts. For example, location a[1,'abc'] of an array a could be specified with the index [1 abc]. The following example initializes just the diagonal locations in a square array:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1 [2 2] 0.2 [3 3] 0.3];
```

An index does not need to specify all the subscripts of an array location. If the index begins with a comma, then only the rightmost subscripts of the index need to be specified. The preceding subscripts are supplied from the index that was used by the preceding *initializer*. This can simplify the initialization of arrays that are indexed by multiple subscripts. For example, you can add new entries to the matrix of the previous example by using the following statements:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1              [,3] 1
                                    [2 2] 0.2  [,3] 2
                                               [3 3] 0.3];
```

The spacing shows the layout of the example array. The previous example was updated by initializing two more values at m[1,3] and m[2,3].

If an index is omitted, then the next location in the order of the array's index set is initialized. If the index set has multiple *index-set-items*, then the rightmost indices are updated before indices to the left are updated. At

the beginning of the initializer list, the rightmost index is the first member of the index set. The index set must use a range expression to avoid unpredictable results when an index value is omitted.

The initializers can be followed by commas. The use of commas has no effect on the initialization. The comma can be used to clarify layout. For example, the comma could separate rows in a matrix.

Not every array location needs to be initialized. The locations without an explicit initializer are set to zero for numeric arrays, set to an empty string for string arrays, and set to an empty set for set arrays.

NOTE: An array location must not be initialized more than once during the processing of the initializer list.

PROBLEM Declaration

> **PROBLEM** *name* [**{** *index-set* **}**] [**FROM** *problem-id*] [**INCLUDE** *problem-items*] **;**

Problems are declared with the PROBLEM declaration. Problem declarations track an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. The problem name can optionally be followed by an *index-set* to create a family of problems. When a problem is first used (via the USE PROBLEM statement), the specifications from the optional FROM and INCLUDE clauses create the initial set of included variables, constraints, and the problem objective. An empty problem is created if neither clause is specified. The clauses are applied only when the problem is first used with the USE PROBLEM statement.

The FROM clause specifies an existing problem from which to copy the included symbols. The *problem-id* is an *identifier expression*. The dropped and fixed status for these symbols in the specified problem is also copied.

The INCLUDE clause specifies a list of variables, constraints, and objectives to include in the problem. These items are included with default status (unfixed and undropped) which overrides the status from the FROM clause, if it exists. Each item is specified with one of the following forms:

identifier-expression

> includes the specified items in the problem. The *identifier-expression* can be a symbol name or an array symbol with explicit index. If an array symbol is used without an index, then all array elements are included.

{ *index-set* **}** *identifier-expression*

> includes the specified subset of items in the problem. The item specified by the *identifier-expression* is added to the problem for each member of the *index-set*. The dummy parameters from the *index-set* can be used in the indexing of the *identifier-expression*. If the *identifier-expression* is an array symbol without indexing, then the *index-set* provides the indices for the included locations.

You can use the FROM and INCLUDE clauses to designate the initial objective for a problem. The objective is copied from the problem designated by the FROM clause, if present. Then the INCLUDE clause, if any, is applied, and the last objective specified becomes the initial objective.

The following statements declare some problems with a variable x and different objectives to illustrate some of the ways of including model components. Note that the statements use the predeclared problem _START_ to avoid resetting the objective in prob2 when the objective z3 is declared.

```
proc optmodel;
   problem prob1;
   use problem prob1;
   var x >= 0;              /* included in prob1 */
   min z1 = (x-1)**2;       /* included in prob1 */
   expand;                  /* prob1 contains x, z1 */

   problem prob2 from prob1;
   use problem prob2;       /* includes x, z1 */
   min z2 = (x-2)**2;       /* resets prob2 objective to z2 */
   expand;                  /* prob2 contains x, z2 */

   use problem _start_;     /* don't modify prob2 */
   min z3 = (x-3)**2;
   problem prob3 include x z3;
   use problem prob3;
   expand;                  /* prob3 contains x, z3 */
```

See the section "Multiple Subproblems" on page 146 for more details about problem processing.

VAR Declaration

> **VAR** *var-decl* [, ... *var-decl*] ;

The VAR statement declares one or more optimization variables. Multiple VAR statements are permitted. A variable is not allowed to have the same name as a parameter or constraint.

Each *var-decl* specifies a variable name. The name can be followed by an array *index-set* specification and then variable options. Dummy parameters declared in the index set specification can be used in the following variable options.

Here is the syntax for a *var-decl*:

> *name* [**{** *index-set* **}**] [*var-options*] ;

For example, the following statements declare a group of 100 variables, x[1]–x[100]:

```
proc optmodel;
   var x{1..100};
```

Here are the available variable options:

INIT *expression*

> sets an initial value for the variable. The expression is used only the first time the value is required. If no initial value is specified, then 0 is used by default.

>= *expression*

> sets a lower bound for the variable value. The default lower bound is $-\infty$.

<= *expression*

> sets an upper bound for the variable value. The default upper bound is ∞.

INTEGER

> requests that the solver assign the variable an integer value.

BINARY

requests that the solver assign the variable a value of either 0 or 1.

For example, the following statements declare a variable that has an initial value of 0.5. The variable is bounded between 0 and 1:

```
proc optmodel;
    var x init 0.5 >= 0 <= 1;
```

The values of the bounds can be determined later by using suffixed references to the variable. For example, the upper bound for variable x can be referred to as x.ub. In addition the bounds options can be overridden by explicit assignment to the suffixed variable name. Suffixes are described further in the section "Suffixes" on page 131.

When used in an expression, an unsuffixed variable name refers to the current value of the variable. Unsuffixed variables are not allowed in the expressions for options that define variable bounds or initial values. Such expressions have values that must be fixed during execution of the solver.

Programming Statements

PROC OPTMODEL supports several programming statements. You can perform various actions with these statements, such as reading or writing data sets, setting parameter values, generating text output, or invoking a solver.

Statements are read from the input and are executed immediately when complete. Certain statements can contain one or more substatements. The execution of substatements is held until the statements that contain them are submitted. Parameter values that are used by expressions in programming statements are resolved when the statement is executed; this resolution might cause errors to be detected. For example, the use of undefined parameters is detected during resolution of the symbolic expressions from declarations.

A statement is terminated by a semicolon. The positions at which semicolons are placed are shown explicitly in the following statement syntax descriptions.

The programming statements can be grouped into the categories shown in Table 5.8.

Table 5.8 Types of Programming Statements in PROC OPTMODEL

Control	Looping	General	Input/Output	Model
DO	CONTINUE	Assignment	CLOSEFILE	DROP
IF	FOR	CALL	CREATE DATA	EXPAND
Null (;)	DO Iterative	PERFORMANCE	FILE	FIX
QUIT	DO UNTIL	RESET OPTIONS	PRINT	RESTORE
STOP	DO WHILE	SUBMIT	PUT	SOLVE
	LEAVE		READ DATA	UNFIX
			SAVE MPS	USE PROBLEM
			SAVE QPS	

Assignment Statement

> *identifier-expression = expression* **;**

The assignment statement assigns a variable or parameter value. The type of the target *identifier-expression* must match the type of the right-hand-side expression.

For example, the following statements set the current value for variable x to 3:

```
proc optmodel;
   var x;
   x = 3;
```

NOTE: Parameters that were declared with the equal sign (=) initialization forms must not be reassigned a value with an assignment statement. If this occurs, PROC OPTMODEL reports an error.

CALL Statement

> **CALL** *name* **(** *argument-1* [**,** ... *argument-n*] **)** **;**

The CALL statement invokes the named library subroutine. The values that are determined for each argument expression are passed to the subroutine when the subroutine is invoked. The subroutine can update the values of PROC OPTMODEL parameters and variables when an argument is an *identifier-expression* (see the section "Identifier Expressions" on page 50). For example, the following statements set the parameter array a to a random permutation of 1 to 4:

```
proc optmodel;
   number a{i in 1..4} init i;
   number seed init -1;
   call ranperm(seed, a[1], a[2], a[3], a[4]);
```

See *SAS Functions and CALL Routines: Reference* for a list of CALL routines.

CLOSEFILE Statement

> **CLOSEFILE** *file-specifications* **;**

The CLOSEFILE statement closes files that were opened by the FILE statement. Each file is specified by a logical name, a physical filename in quotation marks, or an expression enclosed in parentheses that evaluates to a physical filename. See the section "FILE Statement" on page 79 for more information about file specifications.

The following example shows how the CLOSEFILE statement is used with a logical filename:

```
filename greet 'hello.txt';
proc optmodel;
   file greet;
   put 'Hi!';
   closefile greet;
```

Generally you must close a file with a CLOSEFILE statement before external programs can access the file. However, any open files are automatically closed when PROC OPTMODEL terminates.

CONTINUE Statement

> **CONTINUE ;**

The CONTINUE statement terminates the current iteration of the loop statement (iterative DO, DO UNTIL, DO WHILE, or FOR) that immediately contains the CONTINUE statement. Execution resumes at the start of the loop after checking WHILE or UNTIL tests. The FOR or iterative DO loops apply new iteration values.

CREATE DATA Statement

> **CREATE DATA** *SAS-data-set* **FROM** [**[** *key-columns* **]** [**=** *key-set*]] *columns* **;**

The CREATE DATA statement creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables. The CREATE DATA statement can create a data set with a single observation or a data set with observations for every location in one or more arrays. The data set is closed after the execution of the CREATE DATA statement.

The arguments to the CREATE DATA statement are as follows:

SAS-data-set
> specifies the output data set name and options.

key-columns
> declares index values and their corresponding data set variables. The values are used to index array locations in *columns*.

key-set
> specifies a set of index values for the *key-columns*.

columns
> specifies data set variables as well as the PROC OPTMODEL source data for the variables.

Each *column* or *key-column* defines output data set variables and a data source for a column. For example, the following statement generates the output SAS data set resdata from the PROC OPTMODEL array opt, which is indexed by the set indset:

```
create data resdata from [solns]=indset opt;
```

The output data set variable solns contains the index elements in indset.

Columns
Columns can have the following forms:

identifier-expression [/ *options*]
> transfers data from the PROC OPTMODEL parameter or variable specified by the *identifier-expression*. The output data set variable has the same name as the *name* part of the *identifier-expression* (see the section "Identifier Expressions" on page 50). If the *identifier-expression* refers to an array, then the index can be omitted when it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 71 for more information. The following example creates a data set with the variables m and n:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n;
```

name **=** *expression* [/ *options*]

transfers the value of a PROC OPTMODEL expression to the output data set variable *name*. The *expression* is reevaluated for each observation. If the *expression* contains any operators or function calls, then it must be enclosed in parentheses. If the *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 71 for more information. The following example creates a data set with the variable ratio:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from ratio=(m/n);
```

COL(*name-expression***) =** *expression* [/ *options*]

transfers the value of a PROC OPTMODEL expression to the output data set variable named by the string expression *name-expression*. The PROC OPTMODEL expression is reevaluated for each observation. If this expression contains any operators or function calls, then it must be enclosed in parentheses. If the PROC OPTMODEL *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 71 for more information. The following example uses the COL expression to form the variable s5:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from col("s"||n)=(m+n);
```

{ *index-set* **}** **<** *columns* **>**

performs the transfers by iterating each column specified by **<** *columns* **>** for each member of the *index set*. If there are n columns and m index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct output data set variable names in the iterated columns, using COL expressions. The columns are expanded when the CREATE DATA statement is executed, before any output is performed. This form of *columns* cannot be nested. In other words, the following form of *columns* is NOT allowed:

{ *index-set* **}** **<** **{** *index-set* **}** **<** *columns* **>** **>**

The following example demonstrates the use of the iterated *columns* form:

```
proc optmodel;
   set<string> alph = {'a', 'b', 'c'};
   var x{1..3, alph} init 2;
   create data example from [i]=(1..3)
      {j in alph}<col("x"||j)=x[i,j]>;
```

The data set created by these statements is shown in Figure 5.10.

Figure 5.10 CREATE DATA with COL Expression

```
Obs    i    xa    xb    xc

 1     1     2     2     2
 2     2     2     2     2
 3     3     2     2     2
```

NOTE: When no *key-columns* are specified, the output data set has a single observation.

The following statements incorporate several of the preceding examples to create and print a data set by using PROC OPTMODEL parameters:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n ratio=(m/n) col("s"||n)=(m+n);

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.11.

Figure 5.11 CREATE DATA for Single Observation

```
Obs    m    n    ratio    s5

 1     7    5     1.4     12
```

Column Options
Each *column* or *key-column* that defines a data set variable can be followed by zero or more of the following modifiers:

FORMAT=*format.*

associates a format with the current column.

INFORMAT=*informat.*

associates an informat with the current column.

LABEL=*'label'*

associates a label with the current column. The label can be specified by a quoted string or an expression in parentheses.

LENGTH=*length*

specifies a length for the current column. The length can be specified by a numeric constant or a parenthesized expression. The range for character variables is 1 to 32,767 bytes. The range for numeric variables depends on the operating environment and has a minimum of 2 or 3.

TRANSCODE=YES | **NO**

specifies whether character variables can be transcoded. The default value is YES. See

the TRANSCODE=option of the ATTRIB statement in *SAS Statements: Reference* for more information.

The following statements demonstrate the use of column options, including the use of multiple options for a single column:

```
proc optmodel;
   num sq{i in 1..10} = i*i;
   create data squares from [i/format=hex2./length=3] sq/format=6.2;

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.12.

Figure 5.12 CREATE DATA for Single Observation

```
               Obs      i        sq

                1      01       1.00
                2      02       4.00
                3      03       9.00
                4      04      16.00
                5      05      25.00
                6      06      36.00
                7      07      49.00
                8      08      64.00
                9      09      81.00
               10      0A     100.00
```

Key Columns

Key-columns declare index values that enable multiple observations to be written from array *columns*. An observation is created for each unique index value combination. The index values supply the index for array *columns* that do not have an explicit index.

Key-columns define the data set variables where the index value elements are written. They can also declare local dummy parameters for use in expressions in the *columns*. *Key-columns* are syntactically similar to *columns*, but are more restricted in form. The following forms of *key-columns* are allowed:

name [/ *options*]

transfers an index element value to the data set variable *name*. A local dummy parameter, *name*, is declared to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 71 for more information.

COL(name-expression**)** [= *index-name*] [/ *options*]

transfers an index element value to the data set variable named by the string-valued *name-expression*. The argument *index-name* optionally declares a local dummy parameter to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 71 for more information.

A *key-set* in the CREATE DATA statement explicitly specifies the set of index values. *key-set* can be specified as a set expression, although it must be enclosed in parentheses if it contains any function calls or operators. *key-set* can also be specified as an index set expression, in which case the *index-set* dummy parameters override any dummy parameters that are declared in the *key-columns* items. The following statements create a data set from the PROC OPTMODEL parameter m, a matrix whose only nonzero entries are located at (1, 1) and (4, 1):

```
proc optmodel;
   number m{1..5, 1..3} = [[1 1] 1 [4 1] 1];
   create data example
      from [i j] = {setof{i in 1..2}<i**2>, {1, 2}} m;

proc print data=example noobs;
run;
```

The dummy parameter i in the SETOF expression takes precedence over the dummy parameter i declared in the *key-columns* item. The output from these statements is shown in Figure 5.13.

Figure 5.13 CREATE: *key-set* with SETOF Aggregation Expression

i	j	m
1	1	1
1	2	0
4	1	1
4	2	0

If no *key-set* is specified, then the set of index values is formed from the union of the index sets of the implicitly indexed *columns*. The number of index elements for each implicitly indexed array must match the number of *key-columns*. The type of each index element (string versus numeric) must match the element of the same position in other implicit indices.

The arrays for implicitly indexed columns in a CREATE DATA statement do not need to have identical index sets. A missing value is supplied for the value of an implicitly indexed array location when the implied index value is not in the array's index set.

In the following statements, the *key-set* is unspecified. The set of index values is $\{1, 2, 3\}$, which is the union of the index sets of x and y. These index sets are not identical, so missing values are supplied when necessary. The results of these statements are shown in Figure 5.14.

```
proc optmodel;
   number x{1..2} init 2;
   var y{2..3} init 3;
   create data exdata from [keycol] x y;

proc print;
run;
```

Figure 5.14 CREATE: Unspecified *key-set*

```
                Obs    keycol    x    y

                 1        1       2    .
                 2        2       2    3
                 3        3       .    3
```

The types of the output data set variables match the types of the source values. The output variable type for a *key-columns* matches the corresponding element type in the index value tuple. A numeric element matches a NUMERIC data set variable, while a string element matches a CHAR variable. For regular *columns* the source expression type determines the output data set variable type. A numeric expression produces a NUMERIC variable, while a string expression produces a CHAR variable.

Lengths of character variables in the output data set are determined automatically. The length is set to accommodate the longest string value output in that column.

You can use the iterated *columns* form to output selected rows of multiple arrays, assigning a different data set variable to each column. For example, the following statements output the last two rows of the two-dimensional array, a, along with corresponding elements of the one-dimensional array, b:

```
proc optmodel;
   num m = 3;   /* number of rows/observations */
   num n = 4;   /* number of columns in a */
   num a{i in 1..m, j in 1..n} = i*j;   /* compute a */
   num b{i in 1..m} = i**2;   /* compute b */
   set<num> subset = 2..m;   /* used to omit first row */
   create data out
         from [i]=subset {j in 1..n}<col("a"||j)=a[i,j]> b;
```

The preceding statements create a data set out, which has $m - 1$ observations and $n + 2$ variables. The variables are named i, a1 through an, and b, as shown in Figure 5.15.

Figure 5.15 CREATE DATA Set: The Iterated Column Form

```
           Obs    i    a1    a2    a3    a4    b

            1     2     2     4     6     8     4
            2     3     3     6     9    12     9
```

See the section "Data Set Input/Output" on page 115 for more examples of using the CREATE DATA statement.

DO Statement

DO ; *statements* **; END ;**

The DO statement groups a sequence of statements together as a single statement. Each statement within the list is executed sequentially. The DO statement can be used for grouping with the IF and FOR statements.

DO Statement, Iterative

DO *name* = *specification-1* [, ... *specification-n*] **;** *statements* **; END ;**

The iterative DO statement assigns the values from the sequence of *specification* items to a previously declared parameter or variable, *name*. The specified statement sequence is executed after each assignment. This statement corresponds to the iterative DO statement of the DATA step.

Each *specification* provides either a single number or a single string value, or a sequence of such values. Each *specification* takes the following form:

expression [**WHILE(** *logic-expression* **)** | **UNTIL(** *logic-expression* **)**] **;**

The *expression* in the *specification* provides a single value or set of values to assign to the target *name*. Multiple values can be provided for the loop by giving multiple *specification* items that are separated by commas. For example, the following statements output the values 1, 3, and 5:

```
proc optmodel;
   number i;
   do i=1,3,5;
      put i;
   end;
```

In this case, the same effect can be achieved with a single range expression in place of the explicit list of values, as in the following statements:

```
proc optmodel;
   number i;
   do i=1 to 5 by 2;
      put 'value of i assigned by the DO loop = ' i;
      i=i**2;
      put 'value of i assigned in the body of the loop = ' i;
   end;
```

The output of these statements is shown in Figure 5.16.

Figure 5.16 DO Loop: Name Parameter Unaffected

```
value of i assigned by the DO loop = 1
value of i assigned in the body of the loop = 1
value of i assigned by the DO loop = 3
value of i assigned in the body of the loop = 9
value of i assigned by the DO loop = 5
value of i assigned in the body of the loop = 25
```

Unlike the DATA step, a range expression requires the limit to be specified. Additionally the BY part, if any, must follow the limit expression. Moreover, although the *name* parameter can be reassigned in the body of the loop, the sequence of values that is assigned by the DO loop is unaffected.

The argument *expression* can also be an expression that returns a set of numbers or strings. For example, the following statements produce the same sequence of values for i as the previous statements but use a set parameter value:

```
proc optmodel;
   set s = {1,3,5};
   number i;
   do i = s;
      put i;
   end;
```

Each *specification* can include a WHILE or UNTIL clause. A WHILE or UNTIL clause applies to the *expression* that immediately precedes the clause. The sequence that is specified by an *expression* can be terminated early by a WHILE or UNTIL clause. A WHILE *logic-expression* is evaluated for each sequence value before the nested *statements*. If the *logic-expression* returns a false (zero or missing) value, then the current sequence is terminated immediately. An UNTIL *logic-expression* is evaluated for each sequence value after the nested *statements*. The sequence from the current *specification* is terminated if the *logic-expression* returns a true value (nonzero and nonmissing). After early termination of a sequence due to a WHILE or UNTIL expression, the DO loop execution continues with the next *specification*, if any.

To demonstrate use of the WHILE clause, the following statements output the values 1, 2, and 3. In this case the sequence of values from the set s is stopped when the value of i reaches 4.

```
proc optmodel;
   set s = {1,2,3,4,5};
   number i;
   do i = s while(i NE 4);
      put i;
   end;
```

DO UNTIL Statement

DO UNTIL (*logic-expression* **)** *statements* **; END ;**

The DO UNTIL loop executes the specified sequence of statements repeatedly until the *logic-expression*, evaluated after the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do until (i=3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
   do until (i=3 and j=7);
```

For a list of expression operators, see Table 5.3.

DO WHILE Statement

DO WHILE (*logic-expression* **)** *statements* **; END ;**

The DO WHILE loop executes the specified sequence of statements repeatedly as long as the *logic-expression*, evaluated before the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do while (i<3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do while (i<3 and j<7);
```

For a list of expression operators, see Table 5.3.

DROP Statement

DROP *identifier-list* **;**

The DROP statement causes the solver to ignore a list of constraints, constraint arrays, or constraint array locations. The space-delimited *identifier-list* specifies the names of the dropped constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire constraint array is dropped if an *identifier-expression* omits the index for an array name.

The following example statements use the DROP statement:

```
proc optmodel;
   var x{1..10};
   con c1: x[1] + x[2] <= 3;
   con disp{i in 1..9}: x[i+1] >= x[i] + 0.1;

   drop c1;        /* drops the c1 constraint */
   drop disp[5];   /* drops just disp[5] */
   drop disp;      /* drops all disp constraints */
```

The constraint can be added back to the model with the RESTORE statement.

The following line drops both the c1 and disp[5] constraints:

```
drop c1 disp[5];
```

EXPAND Statement

EXPAND [*identifier-expression*] [/ *options*] **;**

The EXPAND statement prints the specified constraint, variable, implicit variable, or objective declaration expressions in the current problem after expanding aggregation operators, substituting the current value for

parameters and indices, and resolving constant subexpressions. *identifier-expression* is the name of a variable, objective, or constraint. If the name is omitted and no *options* are specified, then all variables, objectives, implicit variables, and undropped constraints in the current problem are printed. The following statements show an example EXPAND statement:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand;
```

These statements produce the output in Figure 5.17.

Figure 5.17 EXPAND Statement Output

```
Var x[1]
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 + (-x[2] + 2)**3
Constraint c[1]: x[1] >= 0
Constraint c[2]: x[2] >= 0
```

Specifying an *identifier-expression* restricts output to the specified declaration. A non-array name prints only the specified item. If an array name is used with a specific index, then information for the specified array location is output. Using an array name without an index restricts output to all locations in the array.

You can use the following *options* to further control the EXPAND statement output:

SOLVE

> causes the EXPAND statement to print the variables, objectives, and constraints in the same form that would be seen by the solver if a SOLVE statement were executed. This includes any transformations by the PROC OPTMODEL presolver (see the section "Presolver" on page 141). In this form any fixed variables are replaced by their values. Unless an *identifier-expression* specifies a particular non-array item or array location, the EXPAND output is restricted to only the variables, the constraints, and the current problem objective.

The following options restrict the types of declarations output when no specific non-array item or array location is requested. By default, all types of declarations are output. Only the requested declaration types are output when one or more of the following options are used.

CONSTRAINT | CON

> requests the output of undropped constraints.

FIX

> requests the output of fixed variables. These variables might have been fixed by the FIX statement (or by the presolver if the SOLVE option is specified). The FIX option can also be used in combination with the name of a variable array to display just the fixed elements of the array.

IIS

restricts the display to items found in the irreducible infeasible set (IIS) after the most recent SOLVE performed by the LP solver with the IIS=ON option. The IIS option for the EXPAND statement can also be used in combination with the name of a variable or constraint array to display only the elements of the array in the IIS. For more information about IIS, see the section "Irreducible Infeasible Set" on page 200.

IMPVAR

requests the output of implicit variables referenced in the current problem.

OBJECTIVE | OBJ

requests the output of objectives used in the current problem. This includes the current problem objective and any objectives referenced as implicit variables.

OMITTED   requests the output of variables that are referenced by problem equations but were not included in the current USE PROBLEM instance. The OPTMODEL procedure omits these variables from the generated problem.

VAR

requests the output of unfixed variables. The VAR option can also be used in combination with the name of a variable array to display just the unfixed elements of the array.

For example, you can see the effect of a FIX statement on the problem that is presented to the solver by using the SOLVE option. You can modify the previous example as follows:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand / solve;
```

These statements produce the output in Figure 5.18.

Figure 5.18 Expansion with Fixed Variable

```
Var x[1] >= 0
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 - 1
```

Compare the results in Figure 5.18 to those in Figure 5.17. The constraint c[1] has been converted to a variable bound. The subexpression that uses the fixed variable has been resolved to a constant.

FILE Statement

FILE *file-specification* [**LRECL=**value] ;

The FILE statement selects the current output file for the PUT statement. By default PUT output is sent to the SAS log. Use the FILE statement to manage a group of output files. The specified file is opened for output if it is not already open. The output file remains open until it is closed with the CLOSEFILE statement.

file-specification names the output file. It can use any of the following forms:

'external-file'

>specifies the physical name of an external file in quotation marks. The interpretation of the filename depends on the operating environment.

file-name

>specifies the logical name associated with a file by the FILENAME statement or by the operating environment. The names PRINT and LOG are reserved to refer to the SAS listing and log files, respectively.
>
>**NOTE:** Details about the FILENAME statement can be found in *SAS Statements: Reference*.

(*expression* **)**

>specifies an expression that evaluates to a string that contains the physical name of an external file.

The LRECL= option sets the line length of the output file. The LRECL= option is ignored if the file is already open or if the PRINT or LOG file is specified.

The LRECL= *value* can be specified in these forms:

integer

>specifies the desired line length.

identifier-expression

>specifies the name of a numeric parameter that contains the length.

(*expression* **)**

>specifies a numeric expression in parentheses that returns the line length.

The LRECL= *value* cannot exceed the largest four-byte signed integer, which is $2^{31} - 1$.

The following example shows how to use the FILE statement to handle multiple files:

```
proc optmodel;
   file 'file.txt' lrecl=80;   /* opens file.txt     */
   put 'This is line 1 of file.txt.';
   file print;                  /* selects the listing */
   put 'This goes to the listing.';
   file 'file.txt';            /* reselects file.txt */
   put 'This is line 2 of file.txt.';
   closefile 'file.txt';       /* closes file.txt    */
   file log;                    /* selects the SAS log */
   put 'This goes to the log.';

   /* using expression to open and write a collection of files */
   str ofile;
   num i;
   num l = 40;
   do i = 1 to 3;
      ofile = ('file' || i || '.txt');
      file (ofile) lrecl=(l*i);
      put ('This goes to ' || ofile);
      closefile (ofile);
   end;
```

The following statements illustrate the usefulness of using a logical name associated with a file by FILENAME statement:

```
proc optmodel;
   /* assigns a logical name to file.txt */
   /* see FILENAME statement in */
   /* SAS Statements: Reference */
   filename myfile 'file.txt' mod;

   file myfile;
   put 'This is line 3 of file.txt.';
   closefile myfile;
   file myfile;
   put 'This is line 4 of file.txt.';
   closefile myfile;
```

Notice that the FILENAME statement opens the file referenced for append. Therefore, new data are appended to the end every time the logical name, myfile, is used in the FILE statement.

FIX Statement

FIX *identifier-list* [**= (** *expression* **)**] **;**

The FIX statement causes the solver to treat a list of variables, variable arrays, or variable array locations as fixed in value. The *identifier-list* consists of one or more variable names separated by spaces. Each member of the *identifier-list* is fixed to the same *expression*. For example, the following statements fix the variables x and y to 3:

```
proc optmodel;
   var x, y;
   num a = 2;
   fix x y=(a+1);
```

A variable is specified with an *identifier-expression* (see the section "Identifier Expressions" on page 50). An entire variable array is fixed if the *identifier-expression* names an array without providing an index. A new value can be specified with the *expression*. If the *expression* is a constant, then the parentheses can be omitted. For example, the following statements fix all locations in array x to 0 except x[10], which is fixed to 1:

```
proc optmodel;
   var x{1..10};
   fix x = 0;
   fix x[10] = 1;
```

If *expression* is omitted, the variable is fixed at its current value. For example, you can fix some variables to be their optimal values after the SOLVE statement is invoked.

The effect of FIX can be reversed by using the UNFIX statement.

FOR Statement

> **FOR {** *index-set* **}** *statement* **;**

The FOR statement executes its substatement for each member of the specified *index-set*. The index set can declare local dummy parameters. You can reference the value of these parameters in the substatement. For example, consider the following statements:

```
proc optmodel;
    for {i in 1..2, j in {'a', 'b'}} put i= j=;
```

These statements produce the output in Figure 5.19.

Figure 5.19 FOR Statement Output

```
i=1 j=a
i=1 j=b
i=2 j=a
i=2 j=b
```

As another example, the following statements set the current values for variable x to random values between 0 and 1:

```
proc optmodel;
    var x{1..10};
    for {i in 1..10}
        x[i] = ranuni(-1);
```

Multiple statements can be controlled by specifying a DO statement group for the substatement.

CAUTION: Avoid modifying the parameters that are used by the FOR statement index set from within the substatement. The set value that is used for the left-most index set item is not affected by such changes. However, the effect of parameter changes on later index set items cannot be predicted.

IF Statement

> **IF** *logic-expression* **THEN** *statement* [**ELSE** *statement*] **;**

The IF statement evaluates the logical expression and then conditionally executes the THEN or ELSE substatements. The substatement that follows the THEN keyword is executed when the logical expression result is nonmissing and nonzero. The ELSE substatement, if any, is executed when the logical expression result is a missing value or zero. The ELSE part is optional and must immediately follow the THEN substatement. When IF statements are nested, an ELSE is always matched to the nearest incomplete unmatched IF-THEN. Multiple statements can be controlled by using DO statements with the THEN or ELSE substatements.

NOTE: When an IF-THEN statement is used **without** an ELSE substatement, substatements of the IF statement are executed when possible as they are entered. Under certain circumstances, such as when an IF statement is nested in a FOR loop, the statement is not executed during interactive input until the next statement is seen. By following the IF-THEN statement with an extra semicolon, you can cause it to be executed upon submission, since the extra semicolon is handled as a null statement.

LEAVE Statement

LEAVE ;

The LEAVE statement terminates the execution of the entire loop body (iterative DO, DO UNTIL, DO WHILE, or FOR) that immediately contains the LEAVE statement. Execution resumes at the statement that follows the loop. The following example demonstrates a simple use of the LEAVE statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i >= 3 and j = 2 then leave;
      end;
   print i j;
   end;
```

The results from these statements are displayed in Figure 5.20.

Figure 5.20 LEAVE Statement Output

i	j
1	5

i	j
2	5

i	j
3	2

i	j
4	2

i	j
5	2

For values of i equal to 1 or 2, the inner loop continues uninterrupted, leaving j with a value of 5. For values of i equal to 3, 4, or 5, the inner loop terminates early, leaving j with a value of 2.

Null Statement

;

The null statement is treated as a statement in the PROC OPTMODEL syntax, but its execution has no effect. It can be used as a placeholder statement.

PERFORMANCE Statement

> **PERFORMANCE** *options* **;**

The PERFORMANCE statement controls the multithreaded execution features of PROC OPTMODEL and the multithreaded and distributed execution features of PROC OPTMODEL solvers. The *options* that you specify in the PERFORMANCE statement are applied each time the statement is executed; they replace any previously specified options. For details about the options available for the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 28.

PRINT Statement

> **PRINT** *print-items* **;**

The PRINT statement outputs string and numeric data in tabular form. The statement specifies a list of arrays or other data items to print. Multiple items can be output together as data columns in the same table.

If no format is specified, the PRINT statement handles the details of formatting automatically (see the section "Formatted Output" on page 119 for details). The default format for a numerical column is the fixed-point format (*w.d* format), which is chosen based on the values of the PDIGITS= and PWIDTH= options (see the section "PROC OPTMODEL Statement" on page 56) and on the values in the column. The PRINT statement uses scientific notation (the E*w.* format) when a value is too large or too small to display in fixed format. The default format for a character column is the $*w.* format, where the width is set to be the length of the longest string (ignoring trailing blanks) in the column.

print-item can be specified in the following forms:

identifier-expression [*format*]
> specifies a data item to output. *identifier-expression* can name an array. In that case all defined array locations are output. *format* specifies a SAS format that overrides the default format.

(*expression* **)** [*format*]
> specifies a data value to output. *format* specifies a SAS format that overrides the default format.

{ *index-set* **}** *identifier-expression* [*format*]
> specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. This form can be used to print a subset of the locations in an array, such as a single column. If the *identifier-expression* names an array, then the indices of the array must match the indices of the *index-set*. The *format* argument specifies a SAS format that overrides the default format.

{ *index-set* **}** **(** *expression* **)** [*format*]
> specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. In this form the *expression* is evaluated for each member of the *index-set* to create the array values for output. *format* specifies a SAS format that overrides the default format.

string
> specifies a string value to print.

PAGE
> specifies a page break.

The following example demonstrates the use of several *print-item* forms:

```
proc optmodel;
   num x = 4.3;
   var y{j in 1..4} init j*3.68;
   print y; /* identifier-expression */
   print (x * .265) dollar6.2; /* (expression) [format] */
   print {i in 2..4} y; /* {index-set} identifier-expression */
   print {i in 1..3}(i + i*.2345692) best7.;
                             /* {index-set} (expression) [format] */
   print "Line 1"; /* string */
```

The output is displayed in Figure 5.21.

Figure 5.21 *Print-item* Forms

```
                       [1]         y

                        1        3.68
                        2        7.36
                        3       11.04
                        4       14.72


                          $1.14


                       [1]         y

                        2        7.36
                        3       11.04
                        4       14.72


                       [1]

                        1      1.23457
                        2      2.46914
                        3      3.70371


                          Line 1
```

Adjacent print items that have similar indexing are grouped together and output in the same table. Items have similar indexing if they specify arrays that have the same number of indices and have matching index types (numeric versus string). Nonarray items are considered to have the same indexing as other nonarray items. The resulting table has a column for each array index followed by a column for each print item value. This format is called *list form*. For example, the following statements produce a list form table:

```
proc optmodel;
   num a{i in 1..3} = i*i;
   num b{i in 3..5} = 4*i;
   print a b;
```

These statements produce the listing output in Figure 5.22.

Figure 5.22 List Form PRINT Table

```
           [1]    a    b

            1     1
            2     4
            3     9    12
            4          16
            5          20
```

The array index columns show the set of valid index values for the print items in the table. The array index column for the *i*th index is labeled [*i*]. There is a row for each combination of index values that was used. The index values are displayed in sorted ascending order.

The data columns show the array values that correspond to the index values in each row. If a particular array index is invalid or the array location is undefined, then the corresponding table entry is displayed as blank for numeric arrays and as an empty string for string arrays. If the print items are scalar, then the table has a single row and no array index columns.

If a table contains a single array print item, the array is two-dimensional (has two indices), and the array is dense enough, then the array is shown in *matrix form*. In this format there is a single index column that contains the row index values. The label of this column is blank. This column is followed by a column for every unique column index value for the array. The latter columns are labeled by the column value. These columns contain the array values for that particular array column. Table entries that correspond to array locations that have invalid or undefined combinations of row and column indices are blank or (for strings) printed as an empty string.

The following statements generate a simple example of matrix output:

```
proc optmodel;
   print {i in 1..6, j in i..6} (i*10+j);
```

The PRINT statement produces the output in Figure 5.23.

Figure 5.23 Matrix Form PRINT Table

```
                 1     2     3     4     5     6

          1     11    12    13    14    15    16
          2           22    23    24    25    26
          3                 33    34    35    36
          4                       44    45    46
          5                             55    56
          6                                   66
```

The PRINT statement prints single two-dimensional arrays in the form that uses fewer table cells (headings are ignored). Sparse arrays are normally printed in list form, and dense arrays are normally printed in matrix form. In a PROC OPTMODEL statement, the PMATRIX= option enables you to tune how the PRINT statement displays a two-dimensional array. The value of this option scales the total number of nonempty array elements, which is used to compute the tables cells needed for list form display. Specifying values for the PMATRIX= option less than 1 causes the list form to be used in more cases, while specifying values greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. The default value of the PMATRIX= option is 1. Changing the default can be done with the RESET OPTIONS statement.

The following statements illustrate how the PMATRIX= option affects the display of the PRINT statement:

```
proc optmodel;
   num a{i in 1..6, i..i} = i;
   num b{i in 1..3, j in 1..3} = i*j;
   print a;
   print b;
   reset options pmatrix=3;
   print a;
   reset options pmatrix=0.5;
   print b;
```

The output is shown in Figure 5.24.

Figure 5.24 PRINT Statement: Effects of PMATRIX= Option

```
                 [1]     [2]     a

          1       1       1
          2       2       2
          3       3       3
          4       4       4
          5       5       5
          6       6       6


                          b
                  1     2     3

          1       1     2     3
          2       2     4     6
          3       3     6     9
```

Figure 5.24 *continued*

```
                           a
             1      2      3      4      5      6

      1      1
      2             2
      3                    3
      4                           4
      5                                  5
      6                                         6


            [1]     [2]     b

             1       1      1
             1       2      2
             1       3      3
             2       1      2
             2       2      4
             2       3      6
             3       1      3
             3       2      6
             3       3      9
```

From Figure 5.24, you can see that, by default, the PRINT statement tries to make the display compact. However, you can change the default by using the PMATRIX= option.

PUT Statement

> **PUT** [*put-items*] [**@** | **@@**] **;**

The PUT statement writes text data to the current output file. The syntax of the PUT statement in PROC OPTMODEL is similar to the syntax of the PROC IML and DATA step PUT statements. The PUT statement contains a list of items that specify data for output and provide instructions for formatting the data.

The current output file is initially the SAS log. This can be overridden with the FILE statement. An output file can be closed with the CLOSEFILE statement.

Normally the PUT statement outputs the current line after processing all items. Final @ or @@ operators suppress this automatic line output and cause the current column position to be retained for use in the next PUT statement.

put-item can take any of the following forms.

identifier-expression [**=**] [*format*]

> outputs the value of the parameter or variable that is specified by the *identifier-expression*. The equal sign (=) causes a name for the location to be printed before each location value.

> Normally each item value is printed in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

*name***[*]** [*.suffix*] [**=**] [*format*]

> outputs each defined location value for an array parameter. The array name is specified as

in the *identifier-expression* form except that the index list is replaced by an asterisk (*). The equal sign (=) causes a name for the location to be printed before each location value along with the actual index values to be substituted for the asterisk.

Each item value normally prints in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

(*expression* **)** [**=**] [*format*]
> outputs the value of the expression enclosed in parentheses. This produces similar results to the *identifier-expression* form except that the equal sign (=) uses the expression to form the name.

'*quoted-string*'
> copies the string to the output file.

@*integer* | *identifier-expression* | **(** *expression* **)**   sets the absolute column position within the current line. The literal or expression value determines the new column position.

+*integer* | *identifier-expression* | **(** *expression* **)**   sets the relative column position within the current line. The literal or expression value determines the amount to update the column position.

/
> outputs the current line and moves to the first column of the next line.

PAGE
> outputs any pending line data and moves to the top of the next page.

QUIT Statement

QUIT ;

The QUIT statement terminates the OPTMODEL execution. The statement is executed immediately, so it cannot be a nested statement. A QUIT statement is implied when a DATA or PROC statement is read.

READ DATA Statement

READ DATA *SAS-data-set* [**NOMISS**] **INTO** [[*set-name* **=**] **[** *read-key-columns* **]**] [*read-columns*] **;**

The READ DATA statement reads data from a SAS data set into PROC OPTMODEL parameter and variable locations. The arguments to the READ DATA statement are as follows:

SAS-data-set
> specifies the input data set name and options.

set-name
> specifies a set parameter in which to save the set of observation key values read from the input data set.

read-key-columns
> provide the index values for array destinations.

read-columns
> specify the data values to read and the destination locations.

The following example uses the READ DATA statement to copy data set variables j and k from the SAS data set indata into parameters of the same name. The READ= data set option specifies a password.

```
proc optmodel;
   number j, k;
   read data indata(read=secret) into j k;
```

Key Columns

If any *read-key-columns* are specified, then the READ DATA statement reads all observations from the input data set. If no *read-key-columns* are specified, then only the first observation of the data set is read. The data set is closed after reading the requested information.

Each *read-key-column* declares a local dummy parameter and specifies a data set variable that supplies the column value. The values of the specified data set variables from each observation are combined into a key tuple. This combination is known as the *observation key*. The observation key is used to index array locations specified by the *read-columns* items. The observation key is expected to be unique for each observation read from the data set.

The syntax for a *read-key-column* is as follows:

> *name* [**=** *source-name*] [**/** *trim-option*] **;**

A *read-key-column* creates a local dummy parameter named *name* that holds an element of the observation key tuple. The dummy parameter can be used in subsequent *read-columns* items to reference the element value. If a *source-name* is given, then it specifies the data set variable that supplies the value. Otherwise the source data set variable has the same name as the dummy parameter, *name.* Use the special data set variable name _N_ to refer to the number identification of the observations.

You can specify a *set-name* to save the set of observation keys into a set parameter. If the observation key consists of a single scalar value, then the set member type must match the scalar type. Otherwise the set member type must be a tuple with element types that match the corresponding observation key element types.

The READ DATA statement initially assigns an empty set to the target *set-name* parameter. As observations are read, a tuple for each observation key is added to the set. A set used to index an array destination in the *read-columns* can be read at the same time as the array values. Consider a data set, invdata, created by the following statements:

```
data invdata;
   input item $ invcount;
   datalines;
table 100
sofa  250
chair 80
;
```

The following statements read the data set invdata, which has two variables, item and invcount. The READ DATA statement constructs a set of inventory items, Items. At the same time, the parameter location invcount[item] is assigned the value of the data set variable invcount in the corresponding observation.

```
proc optmodel;
   set<string> Items;
   number invcount{Items};
   read data invdata into Items=[item] invcount;
   print invcount;
```

The output of these statements is shown in Figure 5.25.

Figure 5.25 READ DATA Statement: Key Column

```
               [1]       invcount

               chair          80
               sofa          250
               table         100
```

When observations are read, the values of data set variables are copied to parameter locations. Numeric values are copied unchanged. For character values, *trim-option* controls how leading and trailing blanks are processed. *trim-option* is ignored when the value type is numeric. Specify any of the following keywords for *trim-option*:

TRIM | **TR**

 removes leading and trailing blanks from the data set value. This is the default behavior.

LTRIM | **LT**

 removes only leading blanks from the data set value.

RTRIM | **RT**

 removes only trailing blanks from the data set value.

NOTRIM | **NT**

 copies the data set value with no changes.

Columns

read-columns specify data set variables to read and PROC OPTMODEL parameter locations to which to assign the values. The types of the input data set variables must match the types of the parameters. Array parameters can be implicitly or explicitly indexed by the observation key values.

Normally, missing values from the data set are assigned to the parameters that are specified in the *read-columns*. The NOMISS keyword suppresses the assignment of missing values, leaving the corresponding parameter locations unchanged. Note that the parameter location does not need to have a valid index in this case. This permits a single statement to read data into multiple arrays that have different index sets.

read-columns have the following forms:

identifier-expression [**=** *name* | **COL(** *name-expression* **)**] [/ *trim-option*]

 transfers an input data set variable to a target parameter or variable. *identifier-expression* specifies the target. If the *identifier-expression* specifies an array without an explicit index, then the observation key provides an implicit index. The name of the input data set variable can be specified with a *name* or a COL expression. Otherwise the data set

variable name is given by the *name* part of the *identifier-expression*. For COL expressions, the string-valued *name-expression* is evaluated to determine the data set variable name. *trim-option* controls removal of leading and trailing blanks in the incoming data. For example, the following statements read the data set variables column1 and column2 from the data set exdata into the PROC OPTMODEL parameters p and q, respectively. The observation numbers in exdata are read into the set indx, which indexes p and q.

```
data exdata;
   input column1 column2;
   datalines;
1 2
3 4
;
```

```
proc optmodel;
   number n init 2;
   set<num> indx;
   number p{indx}, q{indx};
   read data exdata into
      indx=[_N_] p=column1 q=col("column"||n);
   print p q;
```

The output is shown in Figure 5.26.

Figure 5.26 READ DATA Statement: Identifier Expressions

```
            [1]    p    q

             1     1    2
             2     3    4
```

{ *index-set* } < *read-columns* >

performs the transfers by iterating each column specified by <*read-columns*> for each member of the *index-set*. If there are *n* columns and *m* index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct input data set variable names in the iterated columns, using COL expressions. The columns are expanded when the READ DATA statement is executed, before any observations are read. This form of *read-columns* cannot be nested. In other words, the following form of *read-columns* is NOT allowed:

{ *index-set* } < { *index-set* } < *read-columns* > >

An example that demonstrates the use of the iterated column *read-option* follows.

You can use an iterated column *read-option* to read multiple data set variables into the same array. For example, a data set might store an entire row of array data in a group of data set variables. The following statements demonstrate how to read a data set that contains demand data divided by day:

```
data dmnd;
   input loc $ day1 day2 day3 day4 day5;
   datalines;
East 1.1 2.3 1.3 3.6 4.7
West 7.0 2.1 6.1 5.8 3.2
;

proc optmodel;
   set DOW = 1..5;  /* days of week, 1=Monday, 5=Friday */
   set<string> LOCS; /* locations */
   number demand{LOCS, DOW};
   read data dmnd
        into LOCS=[loc]
        {d in DOW} < demand[loc, d]=col("day"||d) >;
   print demand;
```

These statements read a set of demand variables named DAY1–DAY5 from each observation, filling in the two-dimensional array demand. The output is shown in Figure 5.27.

Figure 5.27 Demand Data

	demand				
	1	2	3	4	5
East	1.1	2.3	1.3	3.6	4.7
West	7.0	2.1	6.1	5.8	3.2

RESET OPTIONS Statement

RESET OPTIONS *options* ;

RESET OPTION *options* ;

The RESET OPTIONS statement sets PROC OPTMODEL option values or restores them to their defaults. Options can be specified by using the same syntax as in the PROC OPTMODEL statement. The RESET OPTIONS statement provides two extensions to the option syntax. If an option normally requires a value (specified with an equal sign (=) operator), then specifying the option name alone resets it to its default value. You can also specify an expression enclosed in parentheses in place of a literal value. See the section "OPTMODEL Options" on page 149 for an example.

The RESET OPTIONS statement can be placed inside loops or conditional statements. The statement is applied each time it is executed.

RESTORE Statement

RESTORE *identifier-list* ;

The RESTORE statement adds a list of constraints, constraint arrays, or constraint array locations that were dropped by the DROP statement back into the solver model, or includes constraints in a problem where they were not previously present. The space-delimited *identifier-list* specifies the names of the constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire

constraint array is restored if an *identifier-expression* omits the index from an array name. For example, the following statements declare a constraint array and then drop it:

```
con c{i in 1..4}: x[i] + y[i] <=1;
drop c;
```

The following statement restores the first constraint:

```
restore c[1];
```

The following statement restores the second and third constraints:

```
restore c[2] c[3];
```

If you want to restore all of the constraints, you can submit the following statement:

```
restore c;
```

SAVE MPS Statement

> **SAVE MPS** *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] **;**

The SAVE MPS statement saves the structure and coefficients for a linear programming model into a SAS data set. This data set can be used as input data for the OPTLP or OPTMILP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 141) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. The output data set uses the MPS format described in Chapter 15. The generated data set contains observations that define different parts of the linear program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 131 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

The current problem objective is included in the data set. If the OBJECTIVE keyword is used, then the problem objective becomes the specified objective.

When an integer variable has been assigned a nondefault branching priority or direction, the MPS data set includes a BRANCH section. See Chapter 15, "The MPS-Format SAS Data Set" for more details.

The following statements show an example of the SAVE MPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the MPS data set MPSData, as shown in Figure 5.28. Next, PROC OPTLP is used to solve the resulting linear program.

```
proc optmodel;
   var x >= 0, y >= 0;
   con c: x >= y;
   con bx: x <= 2;
   con by: y <= 1;
   min obj=0.5*x-y;
   save mps MPSData;
quit;

proc optlp data=MPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.28 The MPS Data Set Generated by SAVE MPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		MPSData	.		.
2	ROWS			.		.
3	N	obj		.		.
4	G	c		.		.
5	L	bx		.		.
6	L	by		.		.
7	COLUMNS			.		.
8		x	obj	0.5	c	1
9		x	bx	1.0		.
10		y	obj	-1.0	c	-1
11		y	by	1.0		.
12	RHS			.		.
13		.RHS.	bx	2.0		.
14		.RHS.	by	1.0		.
15	ENDATA			.		.

SAVE QPS Statement

SAVE QPS *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] **;**

The SAVE QPS statement saves the structure and coefficients for a quadratic programming model into a SAS data set. This data set can be used as input data for the OPTQP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 141) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. The output data set uses the QPS format described in Chapter 15. The generated data set contains observations that define different parts of the quadratic program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 131 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

The current problem objective is included in the data set. If the OBJECTIVE keyword is used, then the problem objective becomes the specified objective. The coefficients of the objective function appear in the QSECTION section of the output data set.

The following statements show an example of the SAVE QPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the QPS data set QPSData, as shown in Figure 5.29. Next, PROC OPTQP is used to solve the resulting quadratic program.

```
proc optmodel;
   var x{1..2} >= 0;
      min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
                 + 2.5*x[1]*x[2];
      con c1: x[1] - x[2] <= 1;
      con c2: x[1] + 2*x[2] >= 100;
      save qps QPSData;
quit;

proc optqp data=QPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.29 QPS Data Set Generated by the SAVE QPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		QPSData	.		.
2	ROWS			.		.
3	N	z		.		.
4	L	c1		.		.
5	G	c2		.		.
6	COLUMNS			.		.
7		x[1]	z	2.0	c1	1
8		x[1]	c2	1.0		.
9		x[2]	z	3.0	c1	-1
10		x[2]	c2	2.0		.
11	RHS			.		.
12		.RHS.	c1	1.0		.
13		.RHS.	c2	100.0		.
14	QSECTION			.		.
15		x[1]	x[1]	2.0		.
16		x[1]	x[2]	2.5		.
17		x[2]	x[2]	20.0		.
18	ENDATA			.		.

SOLVE Statement

SOLVE [**WITH** *solver*] [**(OBJECTIVE | OBJ)** *name*] [**RELAXINT**] [/ *options*] **;**

The SOLVE statement invokes a PROC OPTMODEL solver. The current model is first resolved to the numeric form that is required by the solver. The resolved model and possibly the current values of any optimization variables are passed to the solver. After the solver finishes executing, the SOLVE statement prints a short table that shows a summary of results from the solver (see the section "ODS Table and Variable Names" on page 122) and updates the _OROPTMODEL_ macro variable.

Here are the arguments to the SOLVE statement:

solver

selects the named solver: LP, MILP, QP, or NLP (see corresponding chapters in this book for details). If no WITH clause is specified, then a solver is chosen automatically, depending on the problem type. Table 5.9 lists the default solver for each problem type.[1]

Table 5.9 Default Solvers and Algorithms in PROC OPTMODEL

Problem	Solver	Algorithm
Linear programming	**LP**	Dual simplex
Mixed integer linear programming	**MILP**	Branch-and-cut
Quadratic programming	**QP**	Interior point QP
General nonlinear programming	**NLP**	Interior point NLP

name

specifies the objective to use. This sets the current objective for the problem. You can abbreviate the OBJECTIVE keyword as OBJ. If this argument is not specified, then the problem objective is unchanged.

RELAXINT   requests that any integral variables be relaxed to be continuous. RELAXINT can be used with linear and nonlinear problems in addition to any solver.

options   specifies solver options. You can specify solver options directly only when you use the WITH clause. A list of the options available with the solver is provided in the individual chapters that describe each solver.

The SOLVE statement uses the value of the predeclared string parameters _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ to provide default solver options. Any options that are specified by these parameters are prepended to options specified in the SOLVE statement, with options from _SOLVER_OPTIONS_ appearing first. These options are included even when the SOLVE statement does not specify a solver with the WITH clause. In this case *solver* is the name of the default solver from Table 5.9.

Initially the predeclared string parameters _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ (for each solver) are empty strings, but they can be assigned by the user. Option values in these strings must be specified with keywords or literal values. Redundant white space is allowed. For example, the following statements set up some simple defaults:

```
_SOLVER_OPTIONS_ = "MAXTIME = 600"; /* options for all solvers */
_LP_OPTIONS_ = "PRESOLVER=AGGRESSIVE"; /* options for LP solver */
```

Optimization techniques that use initial values obtain them from the current values of the optimization variables unless the NOINITVAR option is specified. When the solver finishes executing, the current value of each optimization variable is replaced by the optimal value found by the solver. These values can then be

---

[1]If the QP solver detects nonconvexity (nonconcavity) for a minimization (maximization) problem, the NLP solver is called instead.

used as the initial values for subsequent solver invocations. The .init suffix location for each variable saves the initial value used for the most recent SOLVE statement.

NOTE: If a solver fails, any currently pending statement is stopped and processing continues with the next complete statement read from the input. For example, if a SOLVE statement that is enclosed in a DO group (see the section "DO Statement" on page 74) fails, then the subsequent statements in the group are not executed and processing resumes at the point immediately following the DO group. Neither an infeasible result, an unbounded result, nor reaching an iteration limit is considered to be a solver failure.

NOTE: The information that appears in the macro variable _OROPTMODEL_ (see the section "Macro Variable _OROPTMODEL_" on page 153) varies by solver.

NOTE: The RELAXINT keyword is applied immediately before the problem is passed to the solver, after any processing by the PROC OPTMODEL presolver. So the problem presented to the solver might not be equivalent to the one produced by setting the .RELAX suffix of all variables to a nonzero value. In particular, the bounds of integer variables are still adjusted to be integral, and PROC OPTMODEL's presolver might use integrality to tighten bounds further.

STOP Statement

STOP ;

The STOP statement halts the execution of all statements that contain it, including DO statements and other control or looping statements. Execution continues with the next top-level source statement. The following statements demonstrate a simple use of the STOP statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i = 3 and j = 2 then stop;
      end;
   end;
   print i j;
```

The output is shown in Figure 5.30.

Figure 5.30 STOP Statement: Output

```
          i    j

          3    2
```

When the counters i and j reach 3 and 2, respectively, the STOP statement terminates both loops. Execution continues with the PRINT statement.

SUBMIT Statement

SUBMIT *arguments* [/ *options*] **;**

  SAS statements **;**

ENDSUBMIT ;

The SUBMIT statement allows SAS code to be executed before PROC OPTMODEL processing continues. For example, you can use the SUBMIT statement to invoke other SAS procedures to perform analysis or to display results. The following statements use PROC SORT to order a list of nodes by decreasing priority; the nodes can be used for further processing:

```
proc optmodel;
   set<str> NODES;
   num priority{NODES};

   /* set up priority data... */

   /* sort nodes by descending priority */
   create data temppri from [id] priority;
   submit;
      proc sort;
         by descending priority;
      run;
   endsubmit;

   /* load nodes by priority */
   str nodesByPri{i in 1..card(NODES)};
   read data temppri into [_n_] nodesByPri=id;

   /* use the sorted list... */
```

The SUBMIT statement must appear as the last or only statement on a line. It is followed by lines of SAS statements, terminated by the ENDSUBMIT statement on a line of its own. The SAS statements between the SUBMIT and ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is sent to the SAS language processor each time the SUBMIT statement is executed.

The SUBMIT block can include SAS global statements and procedure and invocations. Macros are not expanded until the SUBMIT block is executed. So you can change macro variables to modify the behavior of the SUBMIT block each time it is processed.

The *arguments* list specifies macro variables to initialize in the SUBMIT block environment before the SUBMIT block is executed. List items are separated by spaces. Each of the *arguments* takes one of the following forms:

name

> copies the value of the PROC OPTMODEL parameter *name* to the macro variable that has the same name.

name = *identifier-expression*

> copies the value of the PROC OPTMODEL parameter specified by *identifier-expression* to the macro variable *name*.

name = *number* | "*string*" | '*string*'

> copies the value of the specified *number* or *string* constant to the macro variable *name*.

name = (*expression*)

> copies the result of evaluating *expression* to the macro variable *name*.

The following statements use a SUBMIT argument to modify the output each time the SUBMIT block is invoked:

```
for {i in 1..5}
   submit a=i;
   %put Value of a is &a..;
   endsubmit;
```

The *options* in the SUBMIT statement are used to retrieve status information after a SUBMIT block is executed. Each item in the space-delimited *options* list has one of the following forms:

OK = *identifier-expression*

>   specifies a PROC OPTMODEL numeric parameter location, *identifier-expression*, that is updated to indicate the success of the SUBMIT block execution. The location is set to 1 if execution is successful or 0 if errors are detected. PROC OPTMODEL continues execution when the SUBMIT block encounters errors only if the OK= option is specified.

OUT [**=**] *output-argument*

>   specifies a single *output-argument* for retrieving macro variable values from the SUBMIT block environment after each execution of the block.

OUT [**=**] **(** *output-argument* **)**

>   specifies a list of space-delimited *output-arguments* for retrieving macro variable values from the SUBMIT block environment after the block is executed.

Each *output-argument* item specifies a macro variable to copy out of the SUBMIT block environment after the block is executed. Each item takes one of the following two forms:

identifier-expression

>   copies the macro variable specified by the *name* portion of the *identifier-expression* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

identifier-expression **=** *name*

>   copies the macro variable specified by *name* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

The following statements show how to use the *options* in the SUBMIT statement to retrieve the result of a SUBMIT block execution:

```
proc optmodel;
   num success, syscc;
   submit / OK = success out syscc;
      data example;
         set notfound;
               j = i*i;
               run;
   endsubmit;
   print success syscc;
```

The DATA step fails, so the success parameter is set to 0 and syscc is set to the error code in the &SYSCC macro variable. The output is shown in .

Figure 5.31 SUBMIT Statement Error Handling

```
              success    syscc

                   0     1012
```

NOTE: The SUBMIT block runs in an environment that is nested in the environment that the OPTMODEL procedure is running in. Resources from the PROC OPTMODEL environment are initially visible in the nested environment. However, the nested environment can have its own local values for options, LIBNAME librefs, FILENAME filerefs, titles, footnotes, and macros. For example, the nested environment has its own global macro scope, which can hide macros visible in the outer environment. The *output-arguments* of the SUBMIT statement *options* can retrieve the values of macros defined in this scope.

NOTE: A SUBMIT statement can appear only in open code. An error message is displayed if the SUBMIT statement is read from a macro. You can avoid this limitation by placing the SUBMIT statement, SUBMIT block, and ENDSUBMIT in a separate file and by using the %INCLUDE statement to include the file in the macro.

UNFIX Statement

   UNFIX *identifier-list* [**= (** *expression* **)**] **;**

The UNFIX statement reverses the effect of FIX statements. The solver can vary the specified variables, variable arrays, or variable array locations specified by *identifier-list*. The *identifier-list* consists of one or more variable names separated by spaces.

Each variable name in the *identifier-list* is an *identifier expression* (see the section "Identifier Expressions" on page 50). The UNFIX statement affects an entire variable array if the identifier expression omits the index from an array name. The *expression* specifies a new initial value that is stored in each element of the *identifier-list*.

The following example demonstrates the UNFIX command:

```
proc optmodel;
   var x{1..3};
   fix x;       /* fixes entire array to 0 */
   unfix x[1]; /* x[1] can now be varied again */
   unfix x[2] = 2;  /* x[2] is given an initial value 2 */
                 /* and can be varied now */
   unfix x;  /* all x indices can now be varied */
```

After the following statements are executed, the variables x[1] and x[2] are not fixed. They each hold the value 4. The variable x[3] is fixed at a value of 2.

```
proc optmodel;
   var x{1..3} init 2;
   num a = 1;
   fix x;
   unfix x[1] x[2]=(a+3);
```

USE PROBLEM Statement

USE PROBLEM *identifier-expression* **;**

The USE PROBLEM programming statement makes the problem specified by the *identifier-expression* be the current problem. If the problem has not been previously used, the problem is created using the PROBLEM declaration corresponding to the name. The problem must have been previously declared.

OPTMODEL Expression Extensions

PROC OPTMODEL defines several new types of expressions for the manipulation of sets. Aggregation operators combine values of an expression that is evaluated over the members of an index set. Other operators create new sets by combining existing sets, or they test relationships between sets. PROC OPTMODEL also supports an IF expression operator that can conditionally evaluate expressions. These and other such expressions are described in this section.

AND Aggregation Expression

AND { *index-set* **}** *logic-expression*

The AND aggregation operator evaluates the logical expression *logic-expression* jointly for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a false value (zero or missing). The expression returns 0 if a false value is found or returns 1 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (and{i in 1..5} i < 10); /* returns 1 */
   put (and{i in 1..5} i NE 3); /* returns 0 */
```

CARD Function

CARD (*set-expression* **)**

The CARD function returns the number of members of its set operand. For example, the following statements produce the output 3 since the set has 3 members:

```
proc optmodel;
   put (card(1..3));
```

CROSS Expression

set-expression-1 **CROSS** *set-expression-2*

The CROSS expression returns the crossproduct of its set operands. The result is the set of tuples formed by concatenating the tuple value of each member of the left operand with the tuple value of each member of the

right operand. Scalar set members are treated as tuples of length 1. The following statements demonstrate the CROSS operator:

```
proc optmodel;
    set s1 = 1..2;
    set<string> s2 = {'a', 'b'};
    set<number, string> s3=s1 cross s2;
    put 's3 is ' s3;
    set<number, string, number> s4 = s3 cross 4..5;
    put 's4 is ' s4;
```

This code produces the output in Figure 5.32.

Figure 5.32 CROSS Expression Output

```
s3 is {<1,'a'>,<1,'b'>,<2,'a'>,<2,'b'>}
s4 is {<1,'a',4>,<1,'a',5>,<1,'b',4>,<1,'b',5>,<2,'a',4>,<2,'a',5>,<2,'b',4>,<2,
'b',5>}
```

DIFF Expression

> *set-expression-1* **DIFF** *set-expression-2*

The DIFF operator returns a set that contains the set difference of the left and right operands. The result set contains values that are members of the left operand but not members of the right operand. The operands must have compatible set types. The following statements evaluate and print a set difference:

```
proc optmodel;
    put ({1,3} diff {2,3}); /* outputs {1} */
```

IF-THEN/ELSE Expression

> **IF** *logic-expression* **THEN** *expression-2* [**ELSE** *expression-3*]

The IF-THEN/ELSE expression evaluates the logical expression *logic-expression* and returns the result of evaluating the second or third operand expression according to the logical test result. If the *logic-expression* is true (nonzero and nonmissing), then the result of evaluating *expression-2* is returned. If the *logic-expression* is false (zero or missing), then the result of evaluating *expression-3* is returned. The other subexpression that is not selected is not evaluated.

An ELSE clause is matched during parsing with the nearest IF-THEN clause that does not have a matching ELSE. The ELSE clause can be omitted for numeric expressions; the resulting IF-THEN is handled as if a default ELSE 0 clause were supplied.

Use the IF-THEN/ELSE expression to handle special cases in models. For example, an inventory model based on discrete time periods might require special handling for the first or last period. In the following example the initial inventory for the first period is assumed to be fixed:

```
proc optmodel;
   number T;
   var inv{1..T}, order{1..T};
   number sell{1..T};
   number inv0;
   . . .
   /* balance inventory flow */
   con iflow{i in 1..T}:
       inv[i] = order[i] - sell[i] +
       if i=1 then inv0 else inv[i-1];
   . . .
```

The IF-THEN/ELSE expression in the example models the initial inventory for a time period i. Usually the inventory value is the inventory at the end of the previous period, but for the first time period the inventory value is given by the inv0 parameter. The iflow constraints are linear because the IF-THEN/ELSE test subexpression does not depend on variables and the other subexpressions are linear.

IF-THEN/ELSE can be used as either a set expression or a scalar expression. The type of expression depends on the subexpression between the THEN and ELSE keywords. The type used affects the parsing of the subexpression that follows the ELSE keyword because the set form has a lower operator precedence. For example, the following two expressions are equivalent because the numeric IF-THEN/ELSE has a higher precedence than the range operator (..):

```
IF logic THEN 1 ELSE 2 .. 3

(IF logic THEN 1 ELSE 2) .. 3
```

But the set form of IF-THEN/ELSE has lower precedence than the range expression operator. So the following two expressions are equivalent:

```
IF logic THEN 1 .. 2 ELSE 3 .. 4

IF logic THEN (1 .. 2) ELSE (3 .. 4)
```

The IF-THEN and IF-THEN/ELSE operators always have higher precedence than the logic operators. So, for example, the following two expressions are equivalent:

```
IF logic THEN numeric1 < numeric2

(IF logic THEN numeric1) < numeric2
```

It is best to use parentheses when in doubt about precedence.

IN Expression

expression **IN** *set-expression*

expression **NOT IN** *set-expression*

The IN expression returns 1 if the value of the left operand is a member of the right operand set. Otherwise, the IN expression returns 0. The NOT IN operator logically negates the returned value. Unlike the DATA step, the right operand is an arbitrary set expression. The left operand can be a tuple expression. The following example demonstrates the IN and NOT IN operators:

```
proc optmodel;
   set s = 1..10;
   put (5 in s);          /* outputs 1 */
   put (-1 not in s);     /* outputs 1 */
   set<num, str> t = {<1,'a'>, <2,'b'>, <2,'c'>};
   put (<2, 'b'> in t);   /* outputs 1 */
   put (<1, 'b'> in t);   /* outputs 0 */
```

Index Set Expression

{ *index-set* **}**

The index set expression returns the set of members of an index set. This expression is distinguished from a set constructor (see the section "Set Constructor Expression" on page 108) because it contains a list of set expressions.

The following statements use an index set with a selection expression that excludes the value 3:

```
proc optmodel;
   put ({i in 1..5 : i NE 3}); /* outputs {1,2,4,5} */
```

INTER Expression

set-expression-1 **INTER** *set-expression-2*

The INTER operator returns a set that contains the intersection of the left and right operands. This is the set that contains values that are members of both operand sets. The operands must have compatible set types.

The following statements evaluate and print a set intersection:

```
proc optmodel;
   put ({1,3} inter {2,3}); /* outputs {3} */
```

INTER Aggregation Expression

INTER { *index-set* **}** *set-expression*

The INTER aggregation operator evaluates the *set-expression* for each member of the index set *index-set*. The result is the set that contains the intersection of the set of values that were returned by the *set-expression* for each member of the index set. An empty index set causes an expression evaluation error.

The following statements use the INTER aggregation operator to compute the value of $\{1,2,3,4\} \cap \{2,3,4,5\} \cap \{3,4,5,6\}$:

```
proc optmodel;
   put (inter{i in 1..3} i..i+3); /* outputs {3,4} */
```

MAX Aggregation Expression

MAX { *index-set* **}** *expression*

The MAX aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the maximum of the values that are returned by the *expression*. Missing values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing value. If the index set is empty, then the result is the negative number that has the largest absolute value representable on the machine.

The following example produces the output `0.5`:

```
proc optmodel;
   put (max{i in 2..5} 1/i);
```

MIN Aggregation Expression

MIN { *index-set* **}** *expression*

The MIN aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the minimum of the values that are returned by the *expression*. Missing values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing value. If the index set is empty, then the result is the largest positive number representable on the machine.

The following example produces the output `0.2`:

```
proc optmodel;
   put (min{i in 2..5} 1/i);
```

OR Aggregation Expression

OR { *index-set* **}** *logic-expression*

The OR aggregation operator evaluates the logical expression *logic-expression* for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a true value (nonzero and nonmissing). The result is 1 if a true value is found, or 0 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (or{i in 1..5} i = 2); /* returns 1 */
   put (or{i in 1..5} i = 7); /* returns 0 */
```

PROD Aggregation Expression

> **PROD {** *index-set* **}** *expression*

The PROD aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the product of the values that are returned by the *expression*. This operator is analogous to the $\prod$ operator used in mathematical notation. If the index set is empty, then the result is 1.

The following example uses the PROD operator to evaluate a factorial:

```
proc optmodel;
   number n = 5;
   put (prod{i in 1..n} i); /* outputs 120 */
```

Range Expression

> *expression-1 .. expression-n* [**BY** *expression*]

The range expression returns the set of numbers from the specified arithmetic progression. The sequence proceeds from the left operand value up to the right operand limit. The increment between numbers is 1 unless a different value is specified with a BY clause. If the increment is negative, then the progression is from the left operand down to the right operand limit. The result can be an empty set.

For compatibility with the DATA step iterative DO loop construct, the keyword TO can substitute for the range (..) operator.

The limit value is not included in the resulting set unless it belongs in the arithmetic progression. For example, the following range expression does not include 30:

```
proc optmodel;
   put (10..30 by 7); /* outputs {10,17,24} */
```

The actual numbers that the range expression "f..l by i" produces are in the arithmetic sequence

$$f, f + i, f + 2i, \ldots, f + ni$$

where

$$n = \left\lfloor \frac{l - f}{i} + \sqrt{\epsilon} \right\rfloor$$

and ϵ represents the relative machine precision. The limit is adjusted to avoid arithmetic roundoff errors.

PROC OPTMODEL represents the set specified by a range expression compactly when the value is stored in a parameter location, used as a set operand of an IN or NOT IN expression, used by an iterative DO loop, or used in an index set. For example, the following expression is evaluated efficiently:

```
999998.5 IN 1..1000000000
```

Set Constructor Expression

{ [*expression-1* [**,** ... *expression-n*]] **}**

The set constructor expression returns the set of the expressions in the member list. Duplicated values are added to the set only once. A warning message is produced when duplicates are detected. The constructor expression consists of zero or more subexpressions of the same scalar type or of tuple expressions that match in length and in element types.

The following statements output a three-member set and warn about the duplicated value 2:

```
proc optmodel;
   put ({1,2,3,2}); /* outputs {1,2,3} */
```

The following example produces a three-member set of tuples, using PROC OPTMODEL parameters and variables. The output is displayed in Figure 5.33.

```
proc optmodel;
   number m = 3, n = 4;
   var x{1..4} init 1;
   string y = 'c';
   put ({<'a', x[3]>, <'b', m>, <y, m/n>});
```

Figure 5.33 Set Constructor Expression Output

```
{<'a',1>,<'b',3>,<'c',0.75>}
```

Set Literal Expression

/ *members* **/**

The set literal expression provides compact specification of simple set values. It is equivalent in function to the set constructor expression but minimizes typing for sets that contain numeric and string constant values. The set members are specified by *members*, which are literal values. As with the set constructor expression, each member must have the same type.

The following statement specifies a simple numeric set:

```
/1 2.5 4/
```

The set contains the members `1`, `2.5`, and `4`. A string set could be specified as follows:

```
/Miami 'San Francisco' Seattle 'Washington, D.C.'/
```

This set contains the strings `'Miami'`, `'San Francisco'`, `'Seattle'`, and `'Washington, D.C.'`. You can specify string values in set literals without quotation marks when the text follows the rules for a SAS

name. Strings that begin with a digit or contain blanks or other special characters must be specified with quotation marks.

Specify tuple members of a set by enclosing the tuple elements within angle brackets (*<elements>*). The tuple elements can be specified with numeric and string literals. The following example includes the tuple elements <'New York', 4.5> and <'Chicago', -5.7>:

```
/<'New York' 4.5> <Chicago -5.7>/
```

SETOF Aggregation Expression

SETOF { *index-set* **}** *expression*

The SETOF aggregation operator evaluates the expression *expression* for each member of the index set *index-set*. The result is the set that is formed by collecting the values returned by the operand expression. The operand can be a tuple expression. For example, the following statements produce a set of tuples of numbers with their squared and cubed values:

```
proc optmodel;
    put (setof{i in 1..3}<i, i*i, i**3>);
```

Figure 5.34 shows the displayed output.

Figure 5.34 SETOF Aggregation Expression Output

```
{<1,1,1>,<2,4,8>,<3,9,27>}
```

SLICE Expression

SLICE (*< element-1,* ... *element-n >* **,** *set-expression* **)**

The SLICE expression produces a new set by selecting members in the operand set that match a pattern tuple. The pattern tuple is specified by the element list in angle brackets. Each *element* in the pattern tuple must specify a numeric or string expression. The expressions are used to match the values of the corresponding elements in the operand set member tuples. You can also specify an *element* by using an asterisk (*). The sequence of element values that correspond to asterisk positions in each matching tuple is combined into a tuple of the result set. At least one asterisk *element* must be specified.

The following statements demonstrate the SLICE expression:

```
proc optmodel;
    put (slice(<1,*>, {<1,3>, <1,0>, <3,1>}));
    put (slice(<*,2,*>, {<1,2,3>, <2,4,3>, <2,2,5>}));
```

These statements produce the output in Figure 5.35.

Figure 5.35 SLICE Expression Output

```
{3,0}
{<1,3>,<2,5>}
```

For the first PUT statement, <1,*> matches set members <1,3> and <1,0> but not <3,1>. The second element of each matching set tuple, corresponding to the asterisk element, becomes the value of the resulting set member. In the second PUT statement, the values of the first and third elements of the operand set member tuple are combined into a two-position tuple in the result set.

The following statements use the SLICE expression to help compute the transitive closure of a set of tuples representing a relation by using Warshall's algorithm. In these statements the set parameter dep represents a direct dependency relation.

```
proc optmodel;
   set<str,str> dep = {<'B','A'>, <'C','B'>, <'D','C'>};
   set<str,str> cl;
   set<str> cn;
   cl = dep;
   cn =  (setof{<i,j> in dep} i) inter (setof{<i,j> in dep} j);
   for {node in cn}
      cl = cl union (slice(<*,node>,cl) cross slice(<node,*>,cl));
   put cl;
```

The local dummy parameter node in the FOR statement iterates over the set cn of possible intermediate nodes that can connect relations transitively. At the end of each FOR iteration, the set parameter cl contains all tuples from the original set in addition to all transitive tuples found in the current or previous iterations.

The output in Figure 5.36 includes the indirect and direct transitive dependencies from the set dep.

Figure 5.36 Warshall's Algorithm Output

```
{<'B','A'>,<'C','B'>,<'D','C'>,<'C','A'>,<'D','B'>,<'D','A'>}
```

A special form of *index-set-item* uses the SLICE expression implicitly. See the section "More on Index Sets" on page 151 for details.

SUM Aggregation Expression

SUM { *index-set* **}** *expression*

The SUM aggregation operator evaluates the numeric expression *expression* for each member in the index set *index-set*. The result is the sum of the values that are returned by the *expression*. If the index set is empty, then the result is 0. This operator is analogous to the $\sum$ operator that is used in mathematical notation. The following statements demonstrate the use of the SUM aggregation operator:

```
proc optmodel;
   put (sum {i in 1..10} i);  /* outputs 55 */
```

SYMDIFF Expression

> *set-expression-1* **SYMDIFF** *set-expression-2*

The SYMDIFF expression returns the symmetric set difference of the left and right operands. The result set contains values that are members of either the left or right operand but are not members of both operands. The operands must have compatible set types.

The following example demonstrates a symmetric difference:

```
proc optmodel;
   put ({1,3} symdiff {2,3}); /* outputs {1,2} */
```

Tuple Expression

> *< expression-1, . . . expression-n >*

A tuple expression represents the value of a member in a set of tuples. Each scalar subexpression inside the angle brackets represents the value of a tuple element. This form is used only with IN, SETOF, and set constructor expressions.

The following statements demonstrate the tuple expression:

```
proc optmodel;
   put (<1,2,3> in setof{i in 1..2}<i,i+1,i+2>);
   put ({<1,'a'>, <2,'b'>} cross {<3,'c'>, <4,'d'>});
```

The first PUT statement checks whether the tuple <1, 2, 3> is a member of a set of tuples. The second PUT statement outputs the cross product of two sets of tuples that are constructed by the set constructor.

These statements produce the output in Figure 5.37.

Figure 5.37 Tuple Expression Output

```
1
{<1,'a',3,'c'>,<1,'a',4,'d'>,<2,'b',3,'c'>,<2,'b',4,'d'>}
```

UNION Expression

> *set-expression-1* **UNION** *set-expression-2*

The UNION expression returns the set union of the left and right operands. The result set contains values that are members of either the left or right operand. The operands must have compatible set types. The following example performs a set union:

```
proc optmodel;
   put ({1,3} union {2,3}); /* outputs {1,3,2} */
```

UNION Aggregation Expression

UNION { *index-set* **}** *set-expression*

The UNION aggregation expression evaluates the *set-expression* for each member of the index set *index-set*. The result is the set union of the values that are returned by the *set-expression*. If the index set is empty, then the result is an empty set.

The following statements demonstrate a UNION aggregation. The output is the value of $\{1,2,3,4\} \cup \{2,3,4,5\} \cup \{3,4,5,6\}$.

```
proc optmodel;
   put (union{i in 1..3} i..i+3); /* outputs {1,2,3,4,5,6} */
```

WITHIN Expression

set-expression-1 **WITHIN** *set-expression-2*

set-expression **NOT WITHIN** *set-expression*

The WITHIN expression returns 1 if the left operand set is a subset of the right operand set and returns 0 otherwise. (That is, the operator returns true if every member of the left operand set is a member of the right operand set.) The NOT WITHIN form logically negates the result value. The following statements demonstrate the WITHIN and NOT WITHIN operators:

```
proc optmodel;
   put ({1,3} within {2,3});     /* outputs 0 */
   put ({1,3} not within {2,3}); /* outputs 1 */
   put ({1,3} within {1,2,3});   /* outputs 1 */
```

Details: OPTMODEL Procedure

Conditions of Optimality

Linear Programming

A standard linear program has the following formulation:

$$
\begin{aligned}
\text{minimize} \quad & \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\
& \mathbf{x} \geq 0
\end{aligned}
$$

where

$$
\begin{array}{lll}
\mathbf{x} & \in & \mathbb{R}^n \qquad \text{is the vector of decision variables} \\
\mathbf{A} & \in & \mathbb{R}^{m \times n} \quad \text{is the matrix of constraints} \\
\mathbf{c} & \in & \mathbb{R}^n \qquad \text{is the vector of objective function coefficients} \\
\mathbf{b} & \in & \mathbb{R}^m \qquad \text{is the vector of constraints right-hand sides (RHS)}
\end{array}
$$

This formulation is called the primal problem. The corresponding dual problem (see the section "Dual Values" on page 134) is

$$
\begin{array}{ll}
\text{maximize} & \mathbf{b}^T \mathbf{y} \\
\text{subject to} & \mathbf{A}^T \mathbf{y} \le \mathbf{c} \\
& \mathbf{y} \ge 0
\end{array}
$$

where $\mathbf{y} \in \mathbb{R}^m$ is the vector of dual variables.

The vectors $\mathbf{x}$ and $\mathbf{y}$ are optimal to the primal and dual problems, respectively, only if there exist primal slack variables $\mathbf{s} = \mathbf{A}\mathbf{x} - \mathbf{b}$ and dual slack variables $\mathbf{w} = \mathbf{A}^T \mathbf{y} - \mathbf{c}$ such that the following *Karush-Kuhn-Tucker (KKT) conditions* are satisfied:

$$
\begin{array}{rclll}
\mathbf{A}\mathbf{x} + \mathbf{s} & = & \mathbf{b}, & \mathbf{x} \ge 0, & \mathbf{s} \ge 0 \\
\mathbf{A}^T \mathbf{y} + \mathbf{w} & = & \mathbf{c}, & \mathbf{y} \ge 0, & \mathbf{w} \ge 0 \\
\mathbf{s}^T \mathbf{y} & = & 0 \\
\mathbf{w}^T \mathbf{x} & = & 0
\end{array}
$$

The first line of equations defines primal feasibility, the second line of equations defines dual feasibility, and the last two equations are called the complementary slackness conditions.

Nonlinear Programming

To facilitate discussion of optimality conditions in nonlinear programming, you write the general form of nonlinear optimization problems by grouping the equality constraints and inequality constraints. You also write all the general nonlinear inequality constraints and bound constraints in one form as "$\ge$" inequality constraints. Thus, you have the following formulation:

$$
\begin{array}{ll}
\underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) \\
\text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\
& c_i(x) \ge 0, \quad i \in \mathcal{I}
\end{array}
$$

where $\mathcal{E}$ is the set of indices of the equality constraints, $\mathcal{I}$ is the set of indices of the inequality constraints, and $m = |\mathcal{E}| + |\mathcal{I}|$.

A point x is *feasible* if it satisfies all the constraints $c_i(x) = 0, i \in \mathcal{E}$ and $c_i(x) \ge 0, i \in \mathcal{I}$. The feasible region $\mathcal{F}$ consists of all the feasible points. In unconstrained cases, the feasible region $\mathcal{F}$ is the entire $\mathbb{R}^n$ space.

A feasible point x^* is a *local solution* of the problem if there exists a neighborhood $\mathcal{N}$ of x^* such that

$$
f(x) \ge f(x^*) \ \textit{for all} \ x \in \mathcal{N} \cap \mathcal{F}
$$

Further, a feasible point x^* is a *strict local solution* if strict inequality holds in the preceding case; that is,

$$f(x) > f(x^*) \ \ for \ all \ x \in \mathcal{N} \cap \mathcal{F}$$

A feasible point x^* is a *global solution* of the problem if no point in $\mathcal{F}$ has a smaller function value than $f(x^*)$; that is,

$$f(x) \geq f(x^*) \ \ for \ all \ x \in \mathcal{F}$$

Unconstrained Optimization

The following conditions hold true for unconstrained optimization problems:

- **First-order necessary conditions**: If x^* is a local solution and $f(x)$ is continuously differentiable in some neighborhood of x^*, then

$$\nabla f(x^*) = 0$$

- **Second-order necessary conditions**: If x^* is a local solution and $f(x)$ is twice continuously differentiable in some neighborhood of x^*, then $\nabla^2 f(x^*)$ is positive semidefinite.

- **Second-order sufficient conditions**: If $f(x)$ is twice continuously differentiable in some neighborhood of x^*, $\nabla f(x^*) = 0$, and $\nabla^2 f(x^*)$ is positive definite, then x^* is a strict local solution.

Constrained Optimization

For constrained optimization problems, the *Lagrangian function* is defined as follows:

$$L(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x)$$

where $\lambda_i, i \in \mathcal{E} \cup \mathcal{I}$, are called *Lagrange multipliers*. $\nabla_x L(x, \lambda)$ is used to denote the gradient of the Lagrangian function with respect to x, and $\nabla_x^2 L(x, \lambda)$ is used to denote the Hessian of the Lagrangian function with respect to x. The active set at a feasible point x is defined as

$$\mathcal{A}(x) = \mathcal{E} \cup \{i \in \mathcal{I} : c_i(x) = 0\}$$

You also need the following definition before you can state the first-order and second-order necessary conditions:

- **Linear independence constraint qualification and regular point**: A point x is said to satisfy the *linear independence constraint qualification* if the gradients of active constraints

$$\nabla c_i(x), \quad i \in \mathcal{A}(x)$$

are linearly independent. Such a point x is called a *regular point*.

You now state the theorems that are essential in the analysis and design of algorithms for constrained optimization:

- **First-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are continuously differentiable, there exist Lagrange multipliers $\lambda^* \in \mathbb{R}^m$ such that the following conditions hold:

$$\nabla_x L(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i^* \nabla c_i(x^*) = 0$$

$$\begin{aligned}
c_i(x^*) &= 0, & i \in \mathcal{E} \\
c_i(x^*) &\geq 0, & i \in \mathcal{I} \\
\lambda_i^* &\geq 0, & i \in \mathcal{I} \\
\lambda_i^* c_i(x^*) &= 0, & i \in \mathcal{I}
\end{aligned}$$

The preceding conditions are often known as the *Karush-Kuhn-Tucker conditions*, or *KKT conditions* for short.

- **Second-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. Let λ^* be the Lagrange multipliers that satisfy the KKT conditions. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are twice continuously differentiable, the following conditions hold:

$$z^T \nabla_x^2 L(x^*, \lambda^*) z \geq 0$$

for all $z \in \mathbb{R}^n$ that satisfy

$$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

- **Second-order sufficient conditions**: Suppose there exist a point x^* and some Lagrange multipliers λ^* such that the KKT conditions are satisfied. If

$$z^T \nabla_x^2 L(x^*, \lambda^*) z > 0$$

for all $z \in \mathbb{R}^n$ that satisfy

$$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

then x^* is a strict local solution.

Note that the set of all such z's forms the null space of the matrix $\left[\nabla c_i(x^*)^T\right]_{i \in \mathcal{A}(x^*)}$. Thus, you can search for strict local solutions by numerically checking the Hessian of the Lagrangian function projected onto the null space. For a rigorous treatment of the optimality conditions, see Fletcher (1987) and Nocedal and Wright (1999).

Data Set Input/Output

You can use the CREATE DATA and READ DATA statements to exchange PROC OPTMODEL data with SAS data sets. The statements can move data into and out of PROC OPTMODEL parameters and variables. For example, the following statements use a CREATE DATA statement to save the results from an optimization into a data set:

```
proc optmodel;
   var x;
   min z = (x-5)**2;
   solve;
   create data optdata from xopt=x z;
```

These statements write a single observation into the data set OPTDATA. The data set contains two variables, xopt and z, and the values contain the optimized values of the PROC OPTMODEL variable x and objective z, respectively. The statement "xopt=x" renames the variable x to xopt.

The group of values held by a data set variable in different observations of a data set is referred to as a *column*. The READ DATA and CREATE DATA statements specify a set of columns for a data set and define how data are to be transferred between the columns and PROC OPTMODEL parameters.

Columns in square brackets ([ ]) are handled specially. Such columns are called *key columns*. Key columns specify element values that provide an implicit index for subsequent array columns. The following example uses key columns with the CREATE DATA statement to write out variable values from an array:

```
proc optmodel;
   set LOCS = {'New York', 'Washington', 'Boston'};  /* locations */
   set DOW = 1..7;  /* day of week */
   var s{LOCS, DOW} init 1;
   create data soldata from [location day_of_week]={LOCS, DOW} sale=s;
```

In this case the optimization variable s is initialized to a value of 1 and is indexed by values from the set parameters LOCS and DOW. The output data set contains an observation for each combination of values in these sets. The output data set contains three variables, location, day_of_week, and sale. The data set variables location and day_of_week save the index element values for the optimization variable s that is written in each observation. The data set created is shown in Figure 5.38.

Figure 5.38 Data Sets Created

```
                  Data Set: SOLDATA


                              day_of_
        Obs      location       week      sale

          1    New York          1          1
          2    New York          2          1
          3    New York          3          1
          4    New York          4          1
          5    New York          5          1
          6    New York          6          1
          7    New York          7          1
          8    Washington        1          1
          9    Washington        2          1
         10    Washington        3          1
         11    Washington        4          1
         12    Washington        5          1
         13    Washington        6          1
         14    Washington        7          1
         15    Boston            1          1
         16    Boston            2          1
         17    Boston            3          1
         18    Boston            4          1
         19    Boston            5          1
         20    Boston            6          1
         21    Boston            7          1
```

Note that the key columns in the preceding example do not name existing PROC OPTMODEL variables. They create new local dummy parameters, location and day_of_week, in the same manner as dummy parameters in index sets. These local parameters can be used in subsequent columns. For example, the following statements demonstrate how to use a key column value in an expression for a later column value:

```
proc optmodel;
   create data tab
           from [i]=(1..10)
           Square=(i*i) Cube=(i*i*i);
```

These statements create a data set that has 10 observations that hold squares and cubes of the numbers from 1 to 10. The key column variable here is named i and is explicitly assigned the values from 1 to 10, while the data set variables Square and Cube hold the square and cube, respectively, of the corresponding value of i.

In the preceding example the key column values are simply the numbers from 1 to 10. The value is the same as the observation number, so the variable i is redundant. You can remove the data set variable for a key column via the DROP data set option, as follows:

```
proc optmodel;
   create data tab2 (drop=i)
           from [i] =(1..10)
           Square=(i*i) Cube=(i*i*i);
```

The local parameters declared by key columns receive their values in various ways. For a READ DATA statement, the key column values come from the data set variables for the column. In a CREATE DATA statement, the values can be defined explicitly, as shown in the previous example. Otherwise, the CREATE DATA statement generates a set of values that combines the index sets of array columns that need implicit indexing. The statements that produce the output in Figure 5.38 demonstrate implicit indexing.

Use a suffix ("Suffixes" on page 131) to read or write auxiliary values, such as variable bounds or constraint duals. For example, consider the following statements:

```
data pdat;
   input p $ maxprod cost;
   datalines;
ABQ    12  0.7
MIA     9  0.6
CHI    14  0.5
run;

proc optmodel;
   set<string> plants;
   var prod{plants} >= 0;
   number cost{plants};
   read data pdat into plants=[p] prod.ub=maxprod cost;
```

The statement "plants=[p]" in the READ DATA statement declares p as a key column and instructs PROC OPTMODEL to store the set of plant names from the data set variable p into the set parameter plants. The statement assigns the upper bound for the variable prod indexed by p to be the value of the data set variable maxprod. The cost parameter location indexed by p is also assigned to be the value of the data set variable cost.

The target variables prod and cost in the preceding example use implicit indexing. Indexing can also be performed explicitly. The following version of the READ DATA statement makes the indices explicit:

```
read data pdat into plants=[p] prod[p].ub=maxprod cost[p];
```

Explicit indexing is useful when array indices need to be transformed from the key column values in the data set. For example, the following statements reverse the order in which elements from the data set are stored in an array:

```
data abcd;
   input letter $ @@;
   datalines;
a b c d
;

proc optmodel;
   set<num> subscripts=1..4;
   string  letter{subscripts};
   read data abcd into [_N_] letter[5-_N_];
   print letter;
```

The output from this example appears in Figure 5.39.

Figure 5.39 READ DATA Statement: Explicit Indexing

```
                           [1]     letter

                           1        d
                           2        c
                           3        b
                           4        a
```

The following example demonstrates the use of explicit indexing to save sequential subsets of an array in individual data sets:

```
data revdata;
   input month rev @@;
   datalines;
1 200 2 345 3 362 4 958
5 659 6 804 7 487 8 146
9 683 10 732 11 652 12 469
;

proc optmodel;
   set m = 1..3;
   var revenue{1..12};
   read data revdata into [_N_] revenue=rev;
   create data qtr1 from [month]=m revenue[month];
   create data qtr2 from [month]=m revenue[month+3];
   create data qtr3 from [month]=m revenue[month+6];
   create data qtr4 from [month]=m revenue[month+9];
```

Each CREATE DATA statement generates a data set that represents one quarter of the year. Each data set contains the variables month and revenue. The data set qtr2 is shown in Figure 5.40.

Figure 5.40 CREATE DATA Statement: Explicit Indexing

```
         Obs    month    revenue

          1       1        958
          2       2        659
          3       3        804
```

Control Flow

Most of the control flow statements in PROC OPTMODEL are familiar to users of the DATA step or the IML procedure. PROC OPTMODEL supports the IF statement, DO blocks, the iterative DO statement, the DO WHILE statement, and the DO UNTIL statement. You can also use the CONTINUE, LEAVE, and STOP statements to modify control flow.

PROC OPTMODEL adds the FOR statement. This statement is similar in operation to an iterative DO loop. However, the iteration is performed over the members of an index set. This form is convenient for iteration over all the locations in an array, since the valid array indices are also defined using an index set. For example, the following statements initialize the array parameter A, indexed by i and j, to random values sampled from a normal distribution with mean 0 and variance 1:

```
proc optmodel;
   set R=1..10;
   set C=1..5;
   number A{R, C};
   for {i in R, j in C}
       A[i, j]=rannor(-1);
```

The FOR statement provides a convenient way to perform a statement such as the preceding assignment statement for each member of a set.

Formatted Output

PROC OPTMODEL provides two primary means of producing formatted output. The PUT statement provides output of data values with detailed format control. The PRINT statement handles arrays and produces formatted output in tabular form.

The PUT statement is similar in syntax to the PUT statement in the DATA step and in PROC IML. The PUT statement can output data to the SAS log, the SAS listing, or an external file. Arguments to the PUT statement specify the data to output and provide instructions for formatting. The PUT statement provides enough control to create reports within PROC OPTMODEL. However, typically the PUT statement is used to produce output for debugging or to quickly check data values.

The following example demonstrates some features of the PUT statement:

```
proc optmodel;
   number a=1.7, b=2.8;
   set s={a,b};
   put a b;          /* list output */
   put a= b=;        /* named output */
   put 'Value A: ' a 8.1 @30 'Value B: ' b 8.; /* formatted */
   string str='Ratio (A/B) is:';
   put str (a/b);   /* strings and expressions */
   put s=;          /* named set output */
```

These statements produce the output in Figure 5.41.

Figure 5.41 PUT Statement Output

```
1.7 2.8
a=1.7 b=2.8
Value A:        1.7             Value B:         3
Ratio (A/B) is: 0.6071428571
s={1.7,2.8}
```

The first PUT statement demonstrates list output. The numeric data values are output in a default format, BEST12., with leading and trailing blanks removed. A blank space is inserted after each data value is output. The second PUT statement uses the equal sign (=) to request that the variable name be output along with the regular list output.

The third PUT statement demonstrates formatted output. It uses the @ operator to position the output in a specific column. This style of output can be used in report generation. The format specification "8." causes the displayed value of parameter b to be rounded.

The fourth PUT statement shows the output of a string value, str. It also outputs the value of an expression enclosed in parentheses. The final PUT statement outputs a set along with its name.

The default destination for PUT statement output is the SAS log. The FILE and CLOSEFILE statements can be used to send output to the SAS listing or to an external data file. Multiple files can be open at the same time. The FILE statement selects the current destination for PUT statement output, and the CLOSEFILE statement closes the corresponding file. See the section "FILE Statement" on page 79 for more details.

The PRINT statement is designed to output numeric and string data in the form of tables. The PRINT statement handles the details of formatting automatically. However, the output format can be overridden by PROC OPTMODEL options and through Output Delivery System (ODS) facilities.

The PRINT statement can output array data in a table form that contains a row for each combination of array index values. This form uses columns to display the array index values for each row and uses other columns to display the value of each requested data item. The following statements demonstrate the table form:

```
proc optmodel;
   number square{i in 0..5} = i*i;
   number recip{i in 1..5} = 1/i;
   print square recip;
```

The PRINT statement produces the output in Figure 5.42.

Figure 5.42 PRINT Statement Output (List Form)

```
                    [1]     square      recip

                    0          0
                    1          1      1.00000
                    2          4      0.50000
                    3          9      0.33333
                    4         16      0.25000
                    5         25      0.20000
```

The first table column, labeled "[1]," contains the index values for the parameters square and recip. The columns that are labeled "square" and "recip" contain the parameter values for each array index. For example, the last row corresponds to the index 5 and the value in the last column is 0.2, which is the value of recip[5].

Note that the first row of the table contains no value in the recip column. Parameter location recip[0] does not have a valid index, so no value is printed. The PRINT statement does not display variables that are undefined or have invalid indices. This permits arrays that have similar indexing to be printed together. The sets of defined indices in the arrays are combined to generate the set of indices shown in the table.

Also note that the PRINT statement has assigned formats and widths that differ between the square and recip columns. The PRINT statement assigns a default fixed-point format to produce the best overall output for each data column. The format that is selected depends on the PDIGITS= and PWIDTH= options.

The PDIGITS= and PWIDTH= options specify the desired significant digits and formatted width, respectively. If the range of magnitudes is large enough that no suitable format can be found, then the data item is displayed in scientific format. The table in the preceding example displays the last column with five decimal places in order to display the five significant digits that were requested by the default PDIGITS= value. The square column, on the other hand, does not need any decimal places.

The PRINT statement can also display two-dimensional arrays in matrix form. If the list following the PRINT statement contains only a single array that has two index elements, then the array is displayed in matrix form when it is sufficiently dense (otherwise the display is in table form). In this form the left-most column contains the values of the first index element. The remaining columns correspond to and are labeled by the values of the second index element. The following statements print an example of matrix form:

```
proc optmodel;
   set R=1..6;
   set C=1..4;
   number a{i in R, j in C} = 10*i+j;
   print a;
```

The PRINT statement produces the output in Figure 5.43.

Figure 5.43 PRINT Statement Output (Matrix Form)

```
                        a
                1     2     3     4

        1      11    12    13    14
        2      21    22    23    24
        3      31    32    33    34
        4      41    42    43    44
        5      51    52    53    54
        6      61    62    63    64
```

In the example the first index element ranges from 1 to 6 and corresponds to the table rows. The second index element ranges from 1 to 4 and corresponds to the table columns. Array values can be found based on the row and column values. For example, the value of parameter a[3,2] is 32. This location is found in the table in the row labeled "3" and the column labeled "2."

ODS Table and Variable Names

PROC OPTMODEL assigns a name to each table it creates. You can use these names to reference the table when you use the Output Delivery System (ODS) to select tables and create output data sets. The names of tables common to all solvers are listed in Table 5.10. Some solvers can generate additional tables; see the individual solver chapters for more information. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Table 5.10 ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Description	Statement/Option
DerivMethods	List of derivatives used by the solver, including the method of computation	SOLVE
OptStatistics	Solver-dependent description of the resources required for solution, including function evaluations and solver time	SOLVE
PrintTable	Specified parameter or variable values	PRINT
ProblemSummary	Description of objective, variables, and constraints	SOLVE
SolutionSummary	Overview of solution, including solver-dependent solution quality values	SOLVE
SolverOptions	List of solver options and their values	SOLVE
PerformanceInfo	List of performance options and their values	SOLVE
Timing	Detailed solution timing	PERFORMANCE / DETAILS

To guarantee that ODS output data sets contain information from all executed statements, use the PERSIST= option in the ODS OUTPUT statement. For details, see *SAS Output Delivery System: User's Guide*. NOTE: The SUBMIT statement resets ODS SELECT and EXCLUDE lists.

Table 5.11 lists the variable names of the preceding tables used in the ODS template of the OPTMODEL procedure.

Table 5.11 Variable Names for the ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Variables
DerivMethods	Label1, cValue1, and nValue1
OptStatistics	Label1, cValue1, and nValue1
PrintTable (matrix form)	ROW, COL1 – COLn
PrintTable (table form)	COL1 – COLn, *identifier-expression(_suffix)*
ProblemSummary	Label1, cValue1, and nValue1
SolutionSummary	Label1, cValue1, and nValue1
SolverOptions	Label1, cValue1, nValue1, cValue2, and nValue2
PerformanceInfo	Label1, cValue1, and nValue1
Timing	Label1, cValue1, nValue1, cValue2, and nValue2

The PRINT statement produces an ODS table named PrintTable. The variable names that are used depend on the display format used. See the section "Formatted Output" on page 119 for details about choosing the display format.

For the PRINT statement with table format, the columns that display array indices are named COL1–COLn, where n is the number of index elements. Columns that display values from identifier expressions are named using the expression's name and suffix. The identifier name becomes the output variable name if no suffix is used. Otherwise the variable name is formed by appending an underscore (_) and the suffix to the identifier name. Columns that display the value of expressions are named COLn, where n is the column number in the table.

For the PRINT statement with matrix format, the first column has the variable name ROW. The remaining columns are named COL1–COLn, where n is the number of distinct column indices. When an ODS table displays values from identifier expressions, a label is generated based on the expression's name and suffix, as described for column names in the case of table format.

The PRINTLEVEL= option controls the ODS tables produced by the SOLVE statement. When PRINTLEVEL=0, the SOLVE statement produces no ODS tables. When PRINTLEVEL=1, the SOLVE statement produces the ODS tables ProblemSummary, SolutionSummary, and PerformanceInfo. When PRINTLEVEL=2, the SOLVE statement produces the ODS tables ProblemSummary, SolverOptions, DerivMethods, SolutionSummary, OptStatistics, and PerformanceInfo.

The PERFORMANCE statement controls additional ODS tables that can be produced by the SOLVE statement. The PerformanceInfo table displays options that are controlled by the PERFORMANCE statement. If you specify the DETAILS option in the PERFORMANCE statement, then the SOLVE statement also produces the ODS table Timing.

The following statements generate several ODS tables and write each table to a SAS data set:

```
proc optmodel printlevel=2;
   ods output PrintTable=expt ProblemSummary=exps DerivMethods=exdm
           SolverOptions=exso SolutionSummary=exss OptStatistics=exos;
   var x{1..2} >= 0;
   min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
           + 2.5*x[1]*x[2] + x[1]**3;
   con c1: x[1] - x[2] <= 1;
   con c2: x[1] + 2*x[2] >= 100;
   solve;
   print x;
```

The data set expt contains the PrintTable table and is shown in Figure 5.44. The variable names are COL1 and x.

Figure 5.44 ODS Table PrintTable

	PrintTable	
Obs	COL1	x
1	1	10.448
2	2	44.776

The data set exps contains the ProblemSummary table and is shown in Figure 5.45. The variable names are Label1, cValue11, and nValue1. The rows describe the objective function, variables, and constraints. The rows depend on the form of the problem.

Figure 5.45 ODS Table ProblemSummary

ProblemSummary

Obs	Label1	cValue1	nValue1
1	Objective Sense	Minimization	.
2	Objective Function	z	.
3	Objective Type	Nonlinear	.
4			.
5	Number of Variables	2	2.000000
6	Bounded Above	0	0
7	Bounded Below	2	2.000000
8	Bounded Below and Above	0	0
9	Free	0	0
10	Fixed	0	0
11			.
12	Number of Constraints	2	2.000000
13	Linear LE (<=)	1	1.000000
14	Linear EQ (=)	0	0
15	Linear GE (>=)	1	1.000000
16	Linear Range	0	0

The data set **exso** contains the SolverOptions table and is shown in Figure 5.46. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The rows, which depend on the solver called by PROC OPTMODEL, list the values taken by each of the solver options. The presence of an asterisk (*) next to an option indicates that a value has been specified for that option.

Figure 5.46 ODS Table SolverOptions

				c	
Obs	Label1	cValue1	nValue1	Value2	nValue2
1	ALGORITHM	INTERIORPOINT	.		.
2	FEASTOL	1E-6	0.000001000		.
3	HESSTYPE	FULL	.		.
4	IIS	OFF	.		.
5	LOGFREQ	1	1.000000		.
6	MAXITER	5000	5000.000000		.
7	MAXTIME	I	I		.
8	NOMULTISTART		.		.
9	OBJLIMIT	1E20	1E20		.
10	OPTTOL	1E-6	0.000001000		.
11	SOLTYPE	1	1.000000		.
12	TIMETYPE	REAL	.		.

The data set **exdm** contains the DerivMethods table, which displays the methods of derivative computation, and is shown in Figure 5.47. The variable names are Label1, cValue11, and nValue1. The rows, which depend on the derivatives used by the solver, specify the method used to calculate each derivative.

Figure 5.47 ODS Table DerivMethods

Obs	Label1	cValue1	nValue1
1	Objective Gradient	Analytic Formulas	.
2	Objective Hessian	Analytic Formulas	.

The data set **exss** contains the SolutionSummary table and is shown in Figure 5.48. The variable names are Label1, cValue11, and nValue1. The rows give an overview of the solution, including the solver chosen, the objective value, and the solution status. Depending on the values returned by the solver, the SolutionSummary table might also include some solution quality values such as optimality error and infeasibility. The values in the SolutionSummary table appear in the _OROPTMODEL_ macro variable; each solver chapter has a section that describes the solver's contribution to this macro variable.

Figure 5.48 ODS Table SolutionSummary

```
                         SolutionSummary

        Obs     Label1                cValue1              nValue1

         1      Solver                NLP                        .
         2      Algorithm             Interior Point             .
         3      Objective Function    z                          .
         4      Solution Status       Optimal                    .
         5      Objective Value       22623.347101           22623
         6                                                       .
         7      Optimality Error            5E-7         0.000000500
         8      Infeasibility                  0                  0
         9                                                       .
        10      Iterations            5                   5.000000
        11      Presolve Time               0.00                 0
        12      Solution Time               0.01          0.014000
```

The data set exos contains the OptStatistics table, which displays the optimization statistics, and is shown in
Figure 5.49. The variable names are Label1, cValue11, and nValue1. The rows, which depend on the solver
called by PROC OPTMODEL, describe the amount of time and function evaluations used by the solver.

Figure 5.49 ODS Table OptStatistics

```
                          OptStatistics

        Obs     Label1                  cValue1            nValue1

         1      Function Evaluations    28               28.000000
         2      Gradient Evaluations    28               28.000000
         3      Hessian Evaluations     6                 6.000000
         4      Problem Generation Time       0.00               0
         5      Code Generation Time          0.10        0.095000
         6      Presolve Time                 0.00               0
         7      Solution Time                 0.01        0.014000
         8      Total Time                    0.18        0.182000
```

Constraints

You can add constraints to a PROC OPTMODEL model. The solver tries to satisfy the specified constraints
while minimizing or maximizing the objective.

Constraints in PROC OPTMODEL have names. By using the name, you can examine various attributes of
the constraint, such as the dual value that is returned by the solver (see the section "Suffixes" on page 131 for
details). A constraint is not allowed to have the same name as any other model component.

PROC OPTMODEL provides a default name if none is supplied by the constraint declaration. The predefined array _ACON_ provides names for otherwise anonymous constraints. The predefined numeric parameter _NACON_ contains the number of such constraints. The constraints are assigned integer indices in sequence, so _ACON_[1] refers to the first unnamed constraint declared, while _ACON_[_NACON_] refers to the newest.

Consider the following example of a simple model that has a constraint:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

Without the constraint named c, the solver would find the point $x = y = 0$ that has an objective value of 0. However, the constraint makes this point infeasible. The resulting output is shown in Figure 5.50.

Figure 5.50 Constrained Model Solution

```
                    Problem Summary

           Objective Sense              Minimization
           Objective Function                      r
           Objective Type                  Quadratic

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

           Number of Constraints                   1
           Linear LE  (<=)                         0
           Linear EQ  (=)                          0
           Linear GE  (>=)                         1
           Linear Range                            0

           Constraint Coefficients                 2


                 Performance Information

            Execution Mode        Single-Machine
            Number of Threads                   4
```

Figure 5.50 *continued*

```
                       Solution Summary

             Solver                             QP
             Algorithm               Interior Point
             Objective Function                   r
             Solution Status               Optimal
             Objective Value           0.4999995397

             Primal Infeasibility         2.5005E-7
             Dual Infeasibility        2.3570226E-7
             Bound Infeasibility                  0
             Duality Gap               1.9575231E-7
             Complementarity           3.9804725E-8

             Iterations                           3
             Presolve Time                     0.00
             Solution Time                     0.01


                      x        y

                     0.5     0.5
```

The solver has found the point where the objective function is minimized in the region $x + y \geq 1$. This is actually on the border of the region: the constraint c is active (see the section "Dual Values" on page 134 for details).

In the preceding example the constraint c had only a lower bound. You can specify constraints that have both upper and lower bounds. For example, replacing the constraint c in the previous example would further restrict the feasible region:

```
   con c: 3 >= x+y >= 1;
```

PROC OPTMODEL standardizes constraints to collect the expression terms that depend on variables and to separate the expression terms that are constant. When there is a single equality or inequality operator, the separable constant terms are moved to the right operand while the variable terms are moved to the left operand. For range constraints, the separable constant terms from the middle expression are subtracted from the lower and upper bounds. You can see the standardized constraints with the use of the EXPAND statement in the following example. Consider the following PROC OPTMODEL statements:

```
proc optmodel;
   var x{1..3};
   con b: sum{i in 1..3}(x[i] - i) = 0;
   expand b;
```

These statements produce an optimization problem with the following constraint:

```
   (x[1] - 1) + (x[2] - 2) + (x[3] - 3) = 0
```

The EXPAND statement produces the output in Figure 5.51.

Figure 5.51 Expansion of a Standardized Constraint

```
Constraint b: x[1] + x[2] + x[3] = 6
```

Here the i separable constant terms in the operand of the SUM operation were moved to the right-hand side of the constraint. The sum of these i values is 6.

After standardization the constraint expression that contains all the variables is called the *body* of the constraint. You can reference the current value of the body expression by attaching the .body suffix to the constraint name. Similarly, the upper and lower bound expressions can be referenced by using the .ub and .lb suffixes, respectively. (See the section "Suffixes" on page 131 for more information.)

As a result of standardization, the value of a body expression depends on how the corresponding constraint is entered. The following example demonstrates how using equivalent relational syntax can result in different .body values:

```
proc optmodel;
    var x init 1;
    con c1: x**2 <= 5;
    con c2: 5 >= x**2;
    con c3: -x**2 >= -5;
    con c4: -5 <= -x**2;
    expand;
    print c1.body c2.body c3.body c4.body;
```

The EXPAND and PRINT statements produce the output in Figure 5.52.

Figure 5.52 Expansion and Body Values of Standardized Constraints

```
Var x
Constraint c1: x**2 <= 5
Constraint c2: -x**2 >= -5
Constraint c3: -x**2 >= -5
Constraint c4: --x**2 <= 5
```

c1.BODY	c2.BODY	c3.BODY	c4.BODY
1	-1	-1	1

CAUTION: Each constraint has an associated *dual value* (see "Dual Values" on page 134). As a result of standardization, the sign of a dual value depends in some instances on the way in which the corresponding constraint is entered into PROC OPTMODEL. In the case of a minimization objective with one-sided constraint $g(x) \geq L$, avoid entering the constraint as $L \leq g(x)$. For example, the following statements produce a value of 2:

```
proc optmodel;
    var x;
    min o1 = x**2;
    con c1: x >= 1;
```

```
    solve;
    print (c1.dual);
```

Replacing the constraint as follows results in a value of –2:

```
    con c1: 1 <= x;
```

In the case of a maximization objective with the one-sided constraint $g(x) \leq U$, avoid entering the constraint as $U \geq g(x)$.

When a constraint has variables on both sides, the sign of the dual value depends on the direction of the inequality. For example, you can enter the following constraint:

```
    con c1: x**5 - y + 8 <= 5*x + y**2;
```

This is a $\leq$ constraint, so c1.dual is nonpositive. If you enter the same constraint as follows, then c1.dual is nonnegative:

```
    con c1: 5*x + y**2 >= x**5 - y + 8;
```

It is also important to note that the signs of the dual values are negated in the case of maximization. The following statements output a value of 2:

```
  proc optmodel;
     var x;
     min o1 = x**2;
     con c1: 1 <= x <= 2;
     solve;
     print (c1.dual);
```

Changing the objective function as follows yields the same value of x, but c1.dual now holds the value –2:

```
     max o1 = -x**2;
```

NOTE: A simple bound constraint on a decision variable x can be entered either by using a CONSTRAINT declaration or by assigning values to x.lb and x.ub. If you require dual values for simple bound constraints, use the CONSTRAINT declaration.

Constraints can be linear or nonlinear. PROC OPTMODEL determines the type of constraint automatically by examining the form of the body expression. Subexpressions that do not involve variables are treated as constants. Constant subexpressions that are multiplied by or added to linear subexpressions produce new linear subexpressions. For example, constraint A in the following statements is linear:

```
  proc optmodel;
     var x{1..3};
     con A: 0.5*(x[1]-x[2]) + x[3] >= 0;
```

Suffixes

Use suffixes with *identifier-expressions* to retrieve and modify various auxiliary values maintained by the solver. The values of the suffixes can come from expressions in the declaration of the name that is suffixed. For example, the following declaration of variable v provides the values of several suffixes of v at the same time:

```
var v >= 0 <= 2 init 1;
```

The values of the suffixes also come from the solver or from values assigned by assignment or READ DATA statements (see an example in the section "Data Set Input/Output" on page 115).

You must use suffixes with names of the appropriate type. For example, the .init suffix cannot be used with the name of an objective. In particular, local dummy parameter names cannot have suffixes.

Table 5.12 shows the names of the available suffixes.

Table 5.12   Suffix Names

Name Kind	Suffix	Modifiable	Description
Variable	.init	No	Initial value for the solver
Variable	.lb	Yes	Lower bound
Variable	.ub	Yes	Upper bound
Variable	.sol	No	Current solution value
Variable	.rc	No	Reduced cost (LP) or gradient of Lagrangian function
Variable	.dual	No	Reduced cost (LP) or gradient of Lagrangian function
Variable	.relax	Yes	Relaxation of integrality restriction
Variable	.priority	Yes	Branching priority
Variable	.direction	Yes	Branching direction
Variable	.msinit	No	Numeric value at the best starting point reported by solver
Variable	.status	Yes	Status information from solver
Variable	.label	Yes	Label text for the solver
Objective	.sol	No	Current objective value
Objective	.label	Yes	Label text for the solver
Constraint	.body	No	Current constraint body value
Constraint	.dual	No	Dual value from the solver
Constraint	.lb	Yes	Current lower bound
Constraint	.ub	Yes	Current upper bound
Constraint	.status	Yes	Status information from solver
Constraint	.label	Yes	Label text for the solver
Constraint	.block	Yes	Block ID for decomposition
Implicit Variable	.sol	No	Current solution value
Problem	.label	Yes	Label text for the solver
any	.name	No	Name text for any non-dummy symbol

NOTE: The .init value of a variable represents the value it had before the most recent SOLVE statement that used the variable. The value is zero before a successful completion of a SOLVE statement that uses the variable.

The .sol suffix for a variable, implicit variable, or objective can be used within a declaration to reference the current value of the symbol. It is treated as a constant in such cases. When processing a SOLVE statement, the value is fixed at the start of the SOLVE. Outside of declarations, a variable, implicit variable, or objective name with the .sol suffix is equivalent to the unsuffixed name.

The .status suffix reports status information from the solver. Currently, only the LP solver provides status information. The .status suffix takes on the same character values that are found in the _STATUS_ variable of the PRIMALOUT and DUALOUT data sets for the OPTLP procedure, including values set by the IIS= option. See the section "Variable and Constraint Status" on page 199 and the section "Irreducible Infeasible Set" on page 200, both in Chapter 6, "The Linear Programming Solver," for more information. For other solvers, the .status values default to a single blank character.

If you choose to modify the .status suffix for a variable or constraint, the assigned suffix value can be a single character or an empty string. The LP solver rejects invalid status characters. Blank or empty strings are treated as new row or column entries for the purpose of "warm starting" the solver.

The .msinit suffix reports the numeric value of a variable at the best starting point, as reported by the NLP solver when the MULTISTART option is specified. If the solver does not report a best starting point, then the value is missing. The value is tracked independently for each problem to support multiple subproblems. See the section "Multistart" on page 317 in Chapter 8, "The Nonlinear Programming Solver," for more information.

The .block suffix identifies the subproblem for constraints when used with the METHOD=USER option of the decomposition algorithm. The value must be numeric and is initially assigned a missing value. A constraint with a missing value for the .block suffix is part of the master problem. Otherwise constraints belong to the same subproblem if and only if they have the same .block suffix values. See Chapter 13, "The Decomposition Algorithm," for more information.

The .label suffix represents the text passed to the solver to identify a variable, constraint, or objective. Some solvers can display this label in their output. The maximum text length passed to the solver is controlled by the MAXLABLEN= option. The default text is based on the name in the model, abbreviated to fit within MAXLABLEN. For example, a model variable x[1] would be labeled "x[1]". This label text can be reassigned. The .label suffix value is also used to create MPS labels stored in the output data set for the SAVE MPS and SAVE QPS statements.

The .name suffix represents the name of a symbol as a text string. The .name suffix can be used with any declared name except for local dummy parameters. This suffix is primarily useful when applied to problem symbols (see the section "Problem Symbols" on page 147), since the .name suffix returns the name of the referenced symbol, not the problem symbol name. The name text is based on the name in the model, abbreviated to fit in 256 characters.

Suffixed names can be used wherever a parameter name is accepted, provided only the value is required. However, you are not allowed to change the value of certain suffixes. Table 5.12 marks these suffixes as not modifiable. Suffixed names that are used as a target in an assignment or READ DATA statement must be modifiable.

The following statements formulate a trivial linear programming problem. The objective value is unbounded, which is reported after the execution of the SOLVE statement. The PRINT statements illustrate the corresponding default auxiliary values. This is shown in Figure 5.53.

```
proc optmodel;
    var x, y;
    min z = x + y;
    con c: x + 2*y <= 3;
    solve;
    print x.lb x.ub x.init x.sol;
    print y.lb y.ub y.init y.sol;
    print c.lb c.ub c.body c.dual;
```

Figure 5.53 Using a Suffix: Retrieving Auxiliary Values

```
           x.LB            x.UB    x.INIT    x.SOL

      -1.7977E+308    1.7977E+308        0        0



           y.LB            y.UB    y.INIT    y.SOL

      -1.7977E+308    1.7977E+308        0        0



              c.LB     c.UB    c.BODY    c.DUAL

      -1.7977E+308        3        0        0
```

Next, continue to submit the following statements to change the default bounds and solve again. The output is shown in Figure 5.54.

```
    x.lb=0;
    y.lb=0;
    c.lb=1;
    solve;
    print x.lb x.ub x.init x.sol;
    print y.lb y.ub y.init y.sol;
    print c.lb c.ub c.body c.dual;
```

Figure 5.54 Using a Suffix: Modifying Auxiliary Values

```
        x.LB            x.UB    x.INIT    x.SOL

           0    1.7977E+308        0        0



        y.LB            y.UB    y.INIT    y.SOL

           0    1.7977E+308        0      0.5



           c.LB     c.UB    c.BODY    c.DUAL

              1        3        1      0.5
```

NOTE: Spaces are significant. The form NAME.␣TAG is treated as a SAS format name followed by the tag name, not as a suffixed identifier. The forms NAME.TAG, NAME␣.␣TAG, and NAME␣.TAG (note the location of spaces) are interpreted as suffixed references.

Integer Variable Suffixes

The suffixes .relax, .priority, and .direction are applicable to integer variables.

For an integer variable x, setting x.relax to a nonzero, nonmissing value relaxes the integrality restriction. The value of x.relax is read as either 1 or 0, depending on whether or not integrality is relaxed. This suffix is ignored for noninteger variables.

The value contained in x.priority sets the branching priority of an integer variable x for use with the MILP solver. This value can be any nonnegative, nonmissing number. The default value is 0, which indicates default branching priority. Variables with positive .priority values are assigned greater priority than the default. Variables with the highest .priority values are assigned the highest priority. Variables with the same .priority value are assigned the same branching priority.

The value of x.direction assigns a branching direction to an integer variable x. This value should be an integer in the range –1 to 3. A noninteger value in this range is rounded on assignment. The default value is 0. The significance of each integer is found in Table 5.13.

Table 5.13 Branching Directions

Value	Direction
–1	Round down to nearest integer
0	Default
1	Round up to nearest integer
2	Round to nearest integer
3	Round to closest presolved bound

Suppose the solver branches next on an integer variable x whose last LP relaxation solution is 3.3. Suppose also that after passing through the presolver, the lower bound of x is 0 and the upper bound of x is 10. If the value in x.direction is –1 or 2, then the solver sets x to 3 for the next iteration. If the value in x.direction is 1, then the solver sets x to 4. If the value in x.direction is 3, then the solver sets x to 0.

The MPS data set created by the SAVE MPS statement ("SAVE MPS Statement" on page 94) includes a BRANCH section if any nondefault .priority or .direction values have been specified for integer variables.

Dual Values

A dual value is associated with each constraint. To access the dual value of a constraint, use the constraint name followed by the suffix .dual.

For linear programming problems, the dual value associated with a constraint is also known as the dual price (also called the shadow price). The shadow price is usually interpreted economically as the rate at which the

optimal value changes with respect to a change in some right-hand side that represents a resource supply or demand requirement.

For nonlinear programming problems, the dual values correspond to the values of the optimal Lagrange multipliers. For more details about duality in nonlinear programming, see Bazaraa, Sherali, and Shetty (1993).

From the dual value associated with the constraint, you can also tell whether the constraint is active or not. A constraint is said to be active (tight at a point) if it holds with equality at that point. It can be informative to identify active constraints at the optimal point and check their corresponding dual values. Relaxing the active constraints might improve the objective value.

Background on Duality in Mathematical Programming

For a minimization problem, there exists an associated problem with the following property: any feasible solution to the associated problem provides a lower bound for the original problem, and conversely any feasible solution to the original problem provides an upper bound for the associated problem. The original and the associated problems are referred to as the primal and the dual problem, respectively. More specifically, consider the primal problem,

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) \\
\text{subject to} \quad & c_i(x) = 0, \quad i \in \mathcal{E} \\
& c_i(x) \leq 0, \quad i \in \mathcal{L} \\
& c_i(x) \geq 0, \quad i \in \mathcal{G}
\end{aligned}
$$

where $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{G}$ denote the sets of equality, $\leq$ inequality, and $\geq$ inequality constraints, respectively. Variables $x \in \mathbb{R}^n$ are called the primal variables. The Lagrangian function of the primal problem is defined as

$$
L(x, \lambda, \mu, \nu) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{L}} \mu_i c_i(x) - \sum_{i \in \mathcal{G}} \nu_i c_i(x)
$$

where $\lambda_i \in \mathbb{R}$, $\mu_i \leq 0$, and $\nu_i \geq 0$. By convention, the Lagrange multipliers for inequality constraints have to be nonnegative. Hence λ, $-\mu$, and ν correspond to the Lagrange multipliers in the preceding Lagrangian function. It can be seen that the Lagrangian function is a linear combination of the objective function and constraints of the primal problem.

The Lagrangian function plays a fundamental role in nonlinear programming. It is used to define the optimality conditions that characterize a local minimum of the primal problem. It is also used to formulate the dual problem of the preceding primal problem. To this end, consider the following *dual* function:

$$
d(\lambda, \mu, \nu) = \inf_{x} \, L(x, \lambda, \mu, \nu)
$$

The dual problem is defined as

$$
\begin{aligned}
\underset{\lambda, \mu, \nu}{\text{maximize}} \quad & d(\lambda, \mu, \nu) \\
\text{subject to} \quad & \mu \leq 0 \\
& \nu \geq 0.
\end{aligned}
$$

The variables λ, μ, and ν are called the dual variables. Note that the dual variables associated with the equality constraints (λ) are free, whereas those associated with $\leq$ inequality constraints (μ) have to be nonpositive and those associated with $\geq$ inequality constraints (ν) have to be nonnegative.

The relation between the primal and the dual problems provides a nice connection between the optimal solutions of the problems. Suppose x^* is an optimal solution of the primal problem and $(\lambda^*, \mu^*, \nu^*)$ is an

optimal solution of the dual problem. The difference between the objective values of the primal and dual problems, $\delta = f(x^*) - d(\lambda^*, \mu^*, \nu^*) \geq 0$, is called the duality gap. For some restricted class of convex nonlinear programming problems, both the primal and the dual problems have an optimal solution and the optimal objective values are equal—that is, the duality gap $\delta = 0$. In such cases, the optimal values of the dual variables correspond to the optimal Lagrange multipliers of the primal problem with the correct signs.

A maximization problem is treated analogously to a minimization problem. For the maximization problem

$$
\begin{array}{ll}
\underset{x}{\text{maximize}} & f(x) \\
\text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\
& c_i(x) \leq 0, \quad i \in \mathcal{L} \\
& c_i(x) \geq 0, \quad i \in \mathcal{G},
\end{array}
$$

the dual problem is

$$
\begin{array}{ll}
\underset{\lambda, \mu, \nu}{\text{minimize}} & d(\lambda, \mu, \nu) \\
\text{subject to} & \mu \geq 0 \\
& \nu \leq 0.
\end{array}
$$

where the dual function is defined as $d(\lambda, \mu, \nu) = \sup_x L(x, \lambda, \mu, \nu)$ and the Lagrangian function $L(x, \lambda, \mu, \nu)$ is defined the same as earlier. In this case, λ, μ, and $-\nu$ correspond to the Lagrange multipliers in $L(x, \lambda, \mu, \nu)$.

Minimization Problems

For inequality constraints in minimization problems, a positive optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution. In PROC OPTMODEL, the optimal dual value for a *range constraint* (a constraint with both upper and lower bounds) is the sum of the dual values associated with the upper and lower inequalities. Since only one of the two inequalities can be active, the sign of the optimal dual value, if nonzero, identifies which one is active.

For equality constraints in minimization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint.

The following is an example where simple linear programming is considered:

```
proc optmodel;
   var x, y;
   min z = 6*x + 7*y;
   con
      4*x +   y >=   5,
       -x - 3*y <=  -4,
        x +   y <=   4;
   solve;
   print x y;
   expand _ACON_ ;
   print _ACON_.dual _ACON_.body;
```

The PRINT statements generate the output shown in Figure 5.55.

Figure 5.55 Dual Values in Minimization Problem: Display

```
                        Problem Summary

            Objective Sense          Minimization
            Objective Function                  z
            Objective Type                 Linear

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

            Number of Constraints               3
            Linear LE (<=)                      2
            Linear EQ (=)                       0
            Linear GE (>=)                      1
            Linear Range                        0

            Constraint Coefficients             6


                   Performance Information

             Execution Mode       Single-Machine
             Number of Threads                 1


                     Solution Summary

            Solver                             LP
            Algorithm                Dual Simplex
            Objective Function                  z
            Solution Status               Optimal
            Objective Value                    13

            Primal Infeasibility                0
            Dual Infeasibility                  0
            Bound Infeasibility                 0

            Iterations                          5
            Presolve Time                    0.00
            Solution Time                    0.00


                          x     y

                          1     1


Constraint _ACON_[1]: y + 4*x >= 5
Constraint _ACON_[2]: - 3*y - x <= -4
Constraint _ACON_[3]: y + x <= 4
```

Figure 5.55 *continued*

	ACON.	_ACON_.
[1]	DUAL	BODY
1	1	5
2	−2	−4
3	0	2

It can be seen that the first and second constraints are active, with dual values 1 and −2. Continue to submit the following statements. Notice how the objective value is changed in Figure 5.56.

```
_ACON_[1].lb = _ACON_[1].lb - 1;
solve;
_ACON_[2].ub = _ACON_[2].ub + 1;
solve;
```

Figure 5.56 Dual Values in Minimization Problem: Interpretation

```
                    Problem Summary

        Objective Sense           Minimization
        Objective Function                   z
        Objective Type              Linear

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

        Number of Constraints                3
        Linear LE  (<=)                      2
        Linear EQ  (=)                       0
        Linear GE  (>=)                      1
        Linear Range                         0

        Constraint Coefficients              6


                Performance Information

         Execution Mode        Single-Machine
         Number of Threads                  1
```

Figure 5.56 *continued*

```
                 Solution Summary

      Solver                           LP
      Algorithm             Dual Simplex
      Objective Function               z
      Solution Status            Optimal
      Objective Value                 12

      Primal Infeasibility             0
      Dual Infeasibility               0
      Bound Infeasibility              0

      Iterations                       5
      Presolve Time                 0.00
      Solution Time                 0.00


                 Problem Summary

   Objective Sense          Minimization
   Objective Function                  z
   Objective Type                 Linear

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

   Number of Constraints               3
   Linear LE (<=)                      2
   Linear EQ (=)                       0
   Linear GE (>=)                      1
   Linear Range                        0

   Constraint Coefficients             6


              Performance Information

    Execution Mode        Single-Machine
    Number of Threads                  1
```

Figure 5.56 *continued*

```
                    Solution Summary

        Solver                            LP
        Algorithm              Dual Simplex
        Objective Function                 z
        Solution Status            Optimal
        Objective Value                   10

        Primal Infeasibility               0
        Dual Infeasibility                 0
        Bound Infeasibility                0

        Iterations                         5
        Presolve Time                   0.00
        Solution Time                   0.00
```

The change is just as the dual values imply. After the first constraint is relaxed by one unit, the objective value is improved by one unit. For the second constraint, the relaxation and improvement are one unit and two units, respectively.

NOTE: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 126 for details.

Maximization Problems

For inequality constraints in maximization problems, a positive optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution. The optimal dual value for a range constraint is the sum of the dual values associated with the upper and lower inequalities. The sign of the optimal dual value identifies which inequality is active.

For equality constraints in maximization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint.

CAUTION: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 126 for details.

Reduced Costs

In linear programming problems, each variable has a corresponding reduced cost. To access the reduced cost of a variable, add the suffix .rc or .dual to the variable name. These two suffixes are interchangeable.

The reduced cost of a variable is the rate at which the objective value changes when the value of that variable changes. At optimality, basic variables have a reduced cost of zero; a nonbasic variable with zero reduced cost indicates the existence of multiple optimal solutions.

In nonlinear programming problems, the reduced cost interpretation does not apply. The .dual and .rc variable suffixes represent the gradient of the Lagrangian function, computed using the values returned by the solver.

The following example illustrates the use of the .rc suffix:

```
proc optmodel;
   var x >= 0, y >= 0, z >= 0;
   max cost = 4*x + 3*y - 5*z;
   con
      -x + y + 5*z <= 15,
      3*x - 2*y - z <= 12,
      2*x + 4*y + 2*z <= 16;
   solve;
   print x y z;
   print x.rc y.rc z.rc;
```

The PRINT statements generate the output shown in Figure 5.57.

Figure 5.57 Reduced Cost in Maximization Problem: Display

x	y	z
5	1.5	0

x.RC	y.RC	z.RC
0	0	-6.5

In this example, x and y are basic variables, while z is nonbasic. The reduced cost of z is –6.5, which implies that increasing z from 0 to 1 decreases the optimal value from 24.5 to 18.

Presolver

PROC OPTMODEL includes a simple presolver that processes linear constraints to produce tighter bounds on variables. The presolver can reduce the number of variables and constraints that are presented to the solver. These changes can result in reduced solution times.

Linear constraints that involve only a single variable are converted into variable bounds. The presolver then eliminates redundant linear constraints for which variable bounds force the constraint to always be satisfied. Tightly bounded variables where upper and lower bounds are within the range specified by the VARFUZZ= option (see the section "PROC OPTMODEL Statement" on page 56) are automatically fixed to the average of the bounds. The presolver also eliminates variables that are fixed by the user or by the presolver.

The presolver can infer tighter variable bounds from linear constraints when all variables in the constraint or all but one variable have known bounds. For example, when given the following PROC OPTMODEL declarations, the presolver can determine the bound $y \le 4$:

```
proc optmodel;
   var x >= 3;
   var y;
```

```
con c: x + y <= 7;
```

The presolver makes multiple passes and rechecks linear constraints after bounds are tightened for the referenced variables. The number of passes is controlled by the PRESOLVER= option. After the passes are finished, the presolver attempts to fix the value of all variables that are not used in the updated objective and constraints. The current value of such a variable is used if the value lies between the variable's upper and lower bounds. Otherwise, the value is adjusted to the nearer bound. The value of an integer variable is rounded before being checked against its bounds.

In some cases the solver might perform better without the presolve transformations, so almost all such transformations are unavailable when the option PRESOLVER=BASIC is specified. However, the presolver still eliminates variables that have values that have been fixed by the FIX statement. To disable the OPTMODEL presolver entirely, use PRESOLVER=NONE. The solver assigns values to any unused, unfixed variables when the option PRESOLVER=NONE is specified.

Model Update

The PROC OPTMODEL modeling language provides several means of modifying a model after it is first specified. You can change the parameter values of the model. You can add new model components. The FIX and UNFIX statements can fix variables to specified values or rescind previously fixed values. The DROP and RESTORE statements can deactivate and reactivate constraints. See also the section "Multiple Subproblems" on page 146 for information on how to maintain multiple models.

To illustrate how these statements work, reconsider the following example from the section "Constraints" on page 126:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

As described previously, the solver finds the optimal point $x = y = 0.5$ with $r = 0.5$. You can see the effect of the constraint c on the solution by temporarily removing it. You can add the following statements:

```
   drop c;
   solve;
   print x y;
```

This change produces the output in Figure 5.58.

Figure 5.58 Solution with Dropped Constraint

```
                    Problem Summary

        Objective Sense            Minimization
        Objective Function                    r
        Objective Type               Quadratic

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

        Number of Constraints                 0

        Constraint Coefficients               0


                 Performance Information

         Execution Mode        Single-Machine
         Number of Threads                  4


                   Solution Summary

         Solver                               QP
         Algorithm                 Interior Point
         Objective Function                    r
         Solution Status                 Optimal
         Objective Value                       0

         Primal Infeasibility                  0
         Dual Infeasibility                    0
         Bound Infeasibility                   0
         Duality Gap                           0
         Complementarity                       0

         Iterations                            0
         Presolve Time                      0.00
         Solution Time                      0.00


                    x     y

                    0     0
```

The optimal point is $x = y = 0$, as expected.

You can restore the constraint c with the RESTORE statement, and you can also investigate the effect of forcing the value of variable x to 0.3. This requires the following statements:

```
restore c;
fix x=0.3;
solve;
print x y c.dual;
```

This produces the output in Figure 5.59.

Figure 5.59 Solution with Fixed Variable

```
                        Problem Summary

          Objective Sense              Minimization
          Objective Function                      r
          Objective Type                  Quadratic

          Number of Variables                     2
          Bounded Above                           0
          Bounded Below                           0
          Bounded Below and Above                 0
          Free                                    1
          Fixed                                   1

          Number of Constraints                   1
          Linear LE (<=)                          0
          Linear EQ (=)                           0
          Linear GE (>=)                          1
          Linear Range                            0

          Constraint Coefficients                 0


                   Performance Information

           Execution Mode        Single-Machine
           Number of Threads                   4


                     Solution Summary

          Solver                                 QP
          Algorithm                   Interior Point
          Objective Function                      r
          Solution Status                   Optimal
          Objective Value                      0.58

          Primal Infeasibility                    0
          Dual Infeasibility                      0
          Bound Infeasibility                     0
          Duality Gap                             0
          Complementarity                         0

          Iterations                              0
          Presolve Time                        0.00
          Solution Time                        0.01
```

Figure 5.59 *continued*

```
              x        y       c.DUAL

             0.3      0.7        1.4
```

The variable x still has the value that was defined in the FIX statement. The objective value has increased by 0.08 from its constrained optimum 0.5 (see Figure 5.50). The constraint c is active, as confirmed by the positive dual value.

You can return to the original optimization problem by allowing the solver to vary variable x with the UNFIX statement, as follows:

```
unfix x;
solve;
print x y c.dual;
```

This produces the output in Figure 5.60. The model was returned to its original conditions.

Figure 5.60 Solution with Original Model

```
                    Problem Summary

          Objective Sense           Minimization
          Objective Function                   r
          Objective Type              Quadratic

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

          Number of Constraints                1
          Linear LE (<=)                       0
          Linear EQ (=)                        0
          Linear GE (>=)                       1
          Linear Range                         0

          Constraint Coefficients              2


                Performance Information

           Execution Mode       Single-Machine
           Number of Threads                  4
```

Figure 5.60 *continued*

```
                      Solution Summary

        Solver                              QP
        Algorithm                Interior Point
        Objective Function                    r
        Solution Status                 Optimal
        Objective Value           0.4999995397

        Primal Infeasibility          2.5005E-7
        Dual Infeasibility        2.3570226E-7
        Bound Infeasibility                   0
        Duality Gap               1.9575231E-7
        Complementarity           3.9804725E-8

        Iterations                            3
        Presolve Time                      0.00
        Solution Time                      0.01


              x       y     c.DUAL

            0.5     0.5         1
```

Multiple Subproblems

The OPTMODEL procedure enables multiple models to be manipulated easily by using named problems to switch the active model components. Problems keep track of an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. Other data, such as parameter values, bounds, and the current value of variables, are shared by all problems.

Problems are declared with the PROBLEM declaration. You can easily switch between problems by using the USE PROBLEM statement. The USE PROBLEM statement makes the specified problem become the current problem. The various statements that generate problem data, such as SOLVE, EXPAND, and SAVE MPS, always operate using the model components included in the current problem.

A problem declaration can specify the problem's initial objective by copying it from the problem named in a FROM clause or by including the objective symbol. This objective can be overridden while the problem is current by declaring a new non-array objective or by executing programming statements that specify a new objective.

Variables can also be included when the problem is current by declaring them or by using the FIX or UNFIX statement. Similarly, constraints can be included when the problem is current by declaring them or by using the RESTORE or DROP statement. There is no way to exclude a variable or constraint item after it has been included in a problem, although the variable or constraint can be fixed or dropped.

Variables that are declared but not included in a problem are treated as fixed when a problem is generated, while constraints that are declared but not included are ignored. The solver does not update the values and status for these model components.

A problem also tracks certain other status information that is associated with its included symbols, and this information can be changed without affecting other problems. This information includes the fixed status for variables, and the dropped status for constraints. The following additional data that are tracked by the problem are available through variable and constraint suffixes:

- *var*.STATUS (including IIS status)

- *var*.INIT

- *var*.MSINIT

- *var*.RC

- *var*.DUAL (alias of *var*.RC)

- *con*.STATUS (including IIS status)

- *con*.DUAL

The initial problem when OPTMODEL starts is predeclared with the name _START_. This problem can be reinstated again (after other USE PROBLEM statements) with the statement

```
use problem _start_;
```

See "Example 5.5: Multiple Subproblems" on page 164 for example statements that use multiple subproblems.

Problem Symbols

The OPTMODEL procedure declares a number of symbols that are aliases for model components in the current problem. These symbols allow the model components to be accessed uniformly. These symbols are described in Table 5.14.

Table 5.14 Problem Symbols

Symbol	Indexing	Description
NVAR		Number of variables
VAR	{1.._NVAR_}	Variable array
NCON		Number of constraints
CON	{1.._NCON_}	Constraint array
_S_NVAR_		Number of presolved variables
_S_VAR_	{1.._S_VAR_}	Presolved variable array
_S_NCON_		Number of presolved constraints
_S_CON_	{1.._S_CON_}	Presolved constraint array
OBJ		Current objective
PROBLEM		Current problem

If the table specifies indexing, then the corresponding symbol is accessed as an array. For example, if the problem includes two variables, x and y, then the value of _NVAR_ is 2 and the current variable values can be accessed as _var_[1] and _var_[2]. The problem variables prefixed with _S are restricted to model components in the problem after processing by the OPTMODEL presolver.

The following statements define a simple linear programming model and then use the problem symbols to print out some of the problem results. The .name suffix is used in the PRINT statements to display the actual variable and constraint names. Any of the suffixes that apply to a model component can be applied to the corresponding generic symbol.

```
proc optmodel printlevel=0;
   var x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0, x5 >= 0;

   minimize z = x1 + x2 + x3 + x4;

   con a1: x1 + x2 + x3            <= 4;
   con a2:               x4 + x5 <= 6;
   con a3: x1 +          x4        >= 5;
   con a4:      x2 +          x5 >= 2;
   con a5:           x3           >= 3;

   solve with lp;

   print _var_.name _var_ _var_.rc _var_.status;
   print _con_.name _con_.lb _con_.body _con_.ub _con_.dual _con_.status;
```

The PRINT statement output is shown in Figure 5.61.

Figure 5.61 Problem Symbol Output

[1]	_VAR_. NAME	_VAR_	_VAR_.RC	_VAR_. STATUS
1	x1	1	0	B
2	x2	0	1	L
3	x3	3	0	B
4	x4	4	0	B
5	x5	2	0	B

[1]	_CON_. NAME	_CON_.LB	_CON_. BODY	_CON_.UB	_CON_. DUAL	_CON_. STATUS
1	a1	-1.7977E308	4	4.0000E+00	0	L
2	a2	-1.7977E308	6	6.0000E+00	0	B
3	a3	5	5	1.7977E+308	1	U
4	a4	2	2	1.7977E+308	0	U
5	a5	3	3	1.7977E+308	1	U

OPTMODEL Options

All PROC OPTMODEL options can be specified in the PROC statement (see the section "PROC OPTMODEL Statement" on page 56 for more information). However, it is sometimes necessary to change options after other PROC OPTMODEL statements have been executed. For example, if an optimization technique had trouble with convergence, then it might be useful to vary the PRESOLVER= option value. This can be done with the RESET OPTIONS statement.

The RESET OPTIONS statement accepts options in the same form used by the PROC OPTMODEL statement. The RESET OPTIONS statement is also able to reset option values and to change options programmatically. For example, the following statements print the value of parameter n at various precisions:

```
proc optmodel;
   number n = 1/7;
   for {i in 1..9 by 4}
   do;
      reset options pdigits=(i);
      print n;
   end;
   reset options pdigits; /* reset to default */
```

The output generated is shown in Figure 5.62. The RESET OPTIONS statement in the DO loop sets the PDIGITS option to the value of i. The final RESET OPTIONS statement restores the default option value, because the value was omitted.

Figure 5.62 Changing the PDIGITS Option Value

```
              n

             0.1

                 n

             0.14286

                   n

           0.142857143
```

Automatic Differentiation

PROC OPTMODEL automatically generates statements to evaluate the derivatives for most objective expressions and nonlinear constraints. PROC OPTMODEL generates analytic derivatives for objective and constraint expressions written in terms of the procedure's mathematical operators and most standard SAS library functions.

NOTE: Some functions, such as ABS, FLOOR, and SIGN, and some operators, such as IF-THEN, <> (element minimum operator), and >< (element maximum operator), must be used carefully in modeling

expressions because functions including such components are not continuously differentiable or even continuous.

Expressions that reference user-defined functions or some SAS library functions might require numerical approximation of derivatives. PROC OPTMODEL uses either forward-difference approximation or central-difference approximation as specified by the FD= option (see the section "PROC OPTMODEL Statement" on page 56).

NOTE: The numerical gradient approximations are significantly slower than automatically generated derivatives when the number of optimization variables is large.

Forward-Difference Approximations

The FD=FORWARD option requests the use of forward-difference derivative approximations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x)}{h_i}$$

Notice that up to n additional function calls are needed here. The step lengths h_i, $i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/2}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $n(n + 3)/2$ extra function calls (Dennis and Schnabel 1983, pp. 80, 104):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{f(x + h_i e_i) - 2f(x) + f(x - h_i e_i)}{h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

Notice that the diagonal of the Hessian uses a central-difference approximation (Abramowitz and Stegun 1972, p. 884). The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Central-Difference Approximations

The FD=CENTRAL option requests the use of central-difference derivative approximations. Generally, central-difference approximations are more accurate than forward-difference approximations, but they require more function evaluations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x - e_i h_i)}{2h_i}$$

Notice that up to $2n$ additional function calls are needed here. The step lengths h_i, $i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $2n(n + 1)$ extra function calls (Abramowitz and Stegun 1972, p. 884):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}$$

The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Conversions

Numeric values are implicitly converted to strings when needed for function arguments or operands to the string concatenation operator (∥). A warning message is generated when the conversion is applied to a function argument. The conversion uses BEST12. format. Unlike the DATA step, the conversion trims blanks.

Implicit conversion of strings to numbers is not permitted. Use the INPUT function to explicitly perform such conversions.

More on Index Sets

Dummy parameters behave like parameters but are assigned values only when an index set is evaluated. You can reference the declared dummy parameters from index set expressions that follow the index set item. You can also reference the dummy parameters in the expression or statement controlled by the index set. As the members of the set expression of an index set item are enumerated, the element values of the members are assigned to the local dummy parameters.

The number of names in a dummy parameter declaration must match the element length of the corresponding set expression in the index set item. A single name is allowed when the set member type is scalar (numeric or string). If the set members are tuples that have $n > 1$ elements, then n names are required between the angle brackets ($<\ >$) that precede the IN keyword.

Multiple index set items in an index set are nominally processed in a left-to-right order. That is, a set expression from an index set item is evaluated as though the index set items that precede it have already been evaluated. The left-hand index set items can assign values to local dummy parameters that are used by the set expressions that follow them. After each member from the set expression is enumerated, any index set items to the right are reevaluated as needed. The actual order in which index set items are evaluated can vary, if necessary, to allow more efficient enumeration. PROC OPTMODEL generates the same set of values in any case, although possibly in a different order than strict left-to-right evaluation.

You can view the element combinations that are generated from an index set as tuples. This is especially true for index set expressions (see the section "Index Set Expression" on page 105). However, in most cases no tuple set is actually formed, and the element values are assigned only to local dummy parameters.

You can specify a selection expression following a colon (:). The index set generates only those combinations of values for which the selection expression is true. For example, the following statements produce a set of upper triangular indices:

```
proc optmodel;
    put (setof {i in 1..3, j in 1..3 : j >= i} <i, j>);
```

These statements produce the output in Figure 5.63.

Figure 5.63 Upper Triangular Index Set

```
{<1,1>,<1,2>,<1,3>,<2,2>,<2,3>,<3,3>}
```

You can use the left-to-right evaluation of index set items to express the previous set more compactly. The following statements produce the same output as the previous statements:

```
proc optmodel;
    put ({i in 1..3, i..3});
```

In this example, the first time the second index set item is evaluated, the value of the dummy parameter i is 1, so the item produces the set {1,2,3}. At the second evaluation the value of i is 2, so the second item produces the set {2,3}. At the final evaluation the value of i is 3, so the second item produces the set {3}.

In many cases it is useful to combine the SLICE operator with index sets. A special form of index set item uses the SLICE operator implicitly. Normally an index set item that is applied to a set of tuples of length greater than one must be of the form

    < name-1 [*,* ... *name-n*] *>* **IN** *set-expression*

In the special form, one or more of the name elements are replaced by expressions. The expressions select tuple elements by using the SLICE operator. An expression that consists of a single name must be enclosed in parentheses to distinguish it from a dummy parameter. The remaining names are the dummy parameters for the index set item that is applied to the SLICE result. The following example demonstrates the use of implicit set slicing:

```
proc optmodel;
   number N = 3;
   set<num,str> S = {<1,'a'>,<2,'b'>,<3,'a'>,<4,'b'>};
   put ({i in 1..N, <(i),j> in S});
   put ({i in 1..N, j in slice(<i,*>, S)});
```

The two PUT statements in this example are equivalent.

Threaded Processing

The OPTMODEL procedure can take advantage of the multiple CPUs that are available in many computers. PROC OPTMODEL automatically uses multithreaded execution to divide problem generation among the multiple CPUs of the computer that is running the procedure. Hessian and Jacobian matrix evaluation is automatically parallelized across threads of execution on multiple CPUs. Threading can decrease the amount of clock time required to perform a task, although the total CPU time required might increase.

If you use the PERFORMANCE statement and specify an NTHREADS option, and the statement does not request distributed computing, then threading in the OPTMODEL procedure is controlled by the NTHREADS option. Otherwise, threading in the OPTMODEL procedure is controlled by the following SAS system options:

CPUCOUNT=*number* | **ACTUAL**
> specifies the maximum number of CPUs that can be used.

THREADS | NOTHREADS
> enables or disables the use of threading.

Good performance is usually obtained with the default option settings (THREADS and CPU-COUNT=ACTUAL). See the option descriptions in *SAS System Options: Reference* for more details.

The PERFORMANCE statement and the SAS system options set the maximum number of threads. The number of threads that PROC OPTMODEL actually uses depends on the characteristics of the problem that is being solved. In particular, threading is not used when the problem is simple enough that threading offers no advantage.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure creates a macro variable named _OROPTMODEL_. You can inspect the execution of the most recently invoked solver from the value of the macro variable. The macro variable is defined at the start of the procedure and updated after each SOLVE statement is executed. The OPTMODEL procedure also updates the macro variable when an error is detected.

The _OROPTMODEL_ value is a string that consists of several "KEYWORD=*value*" items in sequence, separated by blanks; for example:

```
STATUS=OK SOLUTION_STATUS=OPTIMAL OBJECTIVE=9 ITERATIONS=1
PRESOLVE_TIME=0 SOLUTION_TIME=0
```

The information contained in _OROPTMODEL_ varies according to which solver was last called. For lists of keywords and possible values, see the individual solver chapters.

If a value has not been computed, then the corresponding element is not included in the value of the macro variable. When PROC OPTMODEL starts, for example, the macro variable value is set to "STATUS=OK" because no SOLVE statement has been executed. If the STATUS= indicates an error, then the other values from the solver might not be available, depending on when the error occurred.

STATUS and _SOLUTION_STATUS_ Parameters

In addition to generating the macro variable _OROPTMODEL_, the OPTMODEL procedure generates the predeclared string parameters _STATUS_ and _SOLUTION_STATUS_.

The value of _STATUS_ is equal to the STATUS= component of the _OROPTMODEL_ macro variable. The value of _STATUS_ is initially "OK". The value is updated during the SOLVE statement and after statement execution errors.

The value of _SOLUTION_STATUS_ is equal to the SOLUTION_STATUS= component of the _OROPT-MODEL_ macro variable. The value is initially an empty string. The value is updated during the SOLVE statement.

Macro and Statement Evaluation Order

PROC OPTMODEL reads a complete statement, such as a DO statement, before executing any code in it. But macro language statements are processed as the code is read. So you must be careful when using the _OROPTMODEL_ macro variable in code that involves SOLVE statements nested in loops or DO statements. The following statements demonstrate one example of this behavior:

```
proc optmodel;
   var x, y;
   min z=x**2 + (x*y-1)**2;
   for {n in 1..3} do;
      fix x=n;
      solve;
      %put Line 1 &_OROPTMODEL_;
      put 'Line 2 ' (symget("_OROPTMODEL_"));
   end;
quit;
```

In the preceding statements the %PUT statement is executed once, before any SOLVE statements are executed. It displays PROC OPTMODEL's initial setting of the macro variable. But the PUT statement is executed after each SOLVE statement and indicates the expected solution status.

Examples: OPTMODEL Procedure

Example 5.1: Matrix Square Root

This example demonstrates the use of PROC OPTMODEL array parameters and variables. The following statements create a randomized positive definite symmetric matrix and define an optimization model to find

the matrix square root of the generated matrix:

```
proc optmodel;
   number n = 5;   /* size of matrix */
   /* random original array */
   number A{1..n, 1..n} = 10 - 20*ranuni(-1);
   /* compute upper triangle of the
    * symmetric matrix A*transpose(A) */
   /* should be positive def unless A is singular */
   number P{i in 1..n, j in i..n};
   for {i in 1..n, j in i..n}
      P[i,j] = sum{k in 1..n} A[i,k]*A[j,k];
   /* coefficients of square root array
    * (upper triangle of symmetric matrix) */
   var q{i in 1..n, i..n};
   /* The default initial value q[i,j]=0 is
    * a local minimum of the objective,
    * so you must move it away from that point. */
   q[1,1] = 1;
   /* minimize difference of square of q from P */
   min r = sum{i in 1..n, j in i..n}
           (   sum{k in 1..i} q[k,i]*q[k,j]
             + sum{k in i+1..j} q[i,k]*q[k,j]
             + sum{k in j+1..n} q[i,k]*q[j,k]
             - P[i,j] )**2;
   solve;
   print q;
```

These statements define a random array **A** of size $n \times n$. The product **P** is defined as the matrix product AA^T. The product is symmetric, so the declaration of the parameter P gives it upper triangular indexing. The matrix represented by P should be positive definite unless **A** is singular. But singularity is unlikely because of the random generation of **A**. If **P** is positive definite, then it has a well-defined square root, **Q**, such that $P = QQ^T$.

The objective r simply minimizes the sum of squares of the coefficients as

$$r = \sum_{1 \le i \le j \le n} R_{i,j}^2$$

where $R = QQ^T - P$. (This technique for computing matrix square roots is intended only for the demonstration of PROC OPTMODEL capabilities. Better methods exist.)

Output 5.1.1 shows part of the output from running these statements. The values that are actually displayed depend on the random numbers generated.

Output 5.1.1 Matrix Square Root Results

	q				
	1	2	3	4	5
1	12.67867	−8.14753	−6.43848	−0.87666	1.46609
2		−2.45955	−8.23167	4.22369	−8.64930
3			9.20976	2.70390	2.31570
4				−2.41761	−2.44853
5					7.22670

Example 5.2: Reading From and Creating a Data Set

This example demonstrates how to use the READ DATA statement to read parameters from a SAS data set. The objective is the Bard function, which is the following least squares problem with $I = \{1, 2, \ldots, 15\}$:

$$f(x) = \frac{1}{2} \sum_{k \in I} \left[y_k - \left(x_1 + \frac{k}{v_k x_2 + w_k x_3} \right) \right]^2$$

$$x = (x_1, x_2, x_3), \quad y = (y_1, y_2, \ldots, y_{15})$$

where $v_k = 16 - k$, $w_k = \min(k, v_k)$ $(k \in I)$, and

$$y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39)$$

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. This problem is identical to the example "Using the DATA= Option" in Chapter 7, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The following statements use the READ DATA statement to input parameter values and the CREATE DATA statement to save the solution in a SAS data set:

```
data bard;
   input y @@;
   datalines;
.14  .18  .22  .25  .29  .32  .35  .39
.37  .58  .73  .96 1.34 2.10 4.39
;
proc optmodel;
   set I = 1..15;
   number y{I};
   read data bard into [_n_] y;
   number v{k in I} = 16 - k;
   number w{k in I} = min(k, v[k]);
   var x{1..3} init 1;
   min f = 0.5*
      sum{k in I}
         (y[k] - (x[1] + k /
                 (v[k]*x[2] + w[k]*x[3])))**2;
   solve;
   print x;
   create data xdata from [i] xd=x;
```

In these statements the values for parameter y are read from the BARD data set. The set I indexes the terms of the objective in addition to the y array.

The preceding statements define two utility parameters that contain coefficients used in the objective function. These coefficients could have been defined in the expression for the objective, f, but it was convenient to give them names and simplify the objective expression.

The result is shown in Output 5.2.1.

Output 5.2.1 Bard Function Solution

```
            [1]          x

            1        0.08241
            2        1.13303
            3        2.34370
```

The final CREATE DATA statement saves the solution values determined by the solver into the data set XDATA. The data set contains an observation for each x index. Each observation contains two variables. The output variable i contains the index, while xd contains the value for the indexed entry in the array x. The resulting data can be seen by using the PRINT procedure as follows:

```
proc print data=xdata;
run;
```

The output from PROC PRINT is shown in Output 5.2.2.

Output 5.2.2 Output Data Set Contents

```
          Obs    i      xd

           1     1    0.08241
           2     2    1.13303
           3     3    2.34370
```

Example 5.3: Model Construction

This example uses PROC OPTMODEL features to simplify the construction of a mathematically formulated model. The model is based on the example "An Assignment Problem" in Chapter 5, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). A single invocation of PROC OPTMODEL replaces several steps in the PROC LP statements.

The model assigns production of various grades of cloth to a set of machines in order to maximize profit while meeting customer demand. Each machine has different capacities to produce the various grades of cloth. (See the PROC LP example "An Assignment Problem" for more details.) The mathematical formulation, where x_{ijk} represents the amount of cloth of grade j to produce on machine k for customer i, follows:

$$
\begin{aligned}
\text{max} \qquad & \textstyle\sum_{ijk} r_{ijk} x_{ijk} \\
\text{subject to} \qquad & \textstyle\sum_{k} x_{ijk} = d_{ij} && \text{for all } i \text{ and } j \\
& \textstyle\sum_{ij} c_{jk} x_{ijk} \leq a_{k} && \text{for all } k \\
& x_{ijk} \geq 0 && \text{for all } i, j, \text{ and } k
\end{aligned}
$$

The OBJECT, DEMAND, and RESOURCE data sets are the same as in the PROC LP example. A new data set, GRADE, is added to help separate 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;

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 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 PROC 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 ensures that only the nonzero parts of the solution appear in the SOLUTION data set. In the PROC LP example, the creation of this data set required postprocessing of the PROC LP output data set.

The solver produces the following problem summary and solution summary:

Output 5.3.1 LP Solver Result

```
                    An Assignment Problem

                      Problem Summary

          Objective Sense           Maximization
          Objective Function          TotalReturn
          Objective Type                   Linear

          Number of Variables                 120
          Bounded Above                         0
          Bounded Below                       120
          Bounded Below and Above               0
          Free                                  0
          Fixed                                 0

          Number of Constraints                34
          Linear LE (<=)                        4
          Linear EQ (=)                        30
          Linear GE (>=)                        0
          Linear Range                          0

          Constraint Coefficients             220
```

Output 5.3.1 *continued*

```
                    Solution Summary

         Solver                          LP
         Algorithm              Dual Simplex
         Objective Function       TotalReturn
         Solution Status             Optimal
         Objective Value       871426.03763

         Primal Infeasibility               0
         Dual Infeasibility                 0
         Bound Infeasibility                0

         Iterations                        60
         Presolve Time                   0.00
         Solution Time                   0.00
```

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;
```

These statements produce the table shown in Output 5.3.2.

Output 5.3.2 An Assignment Problem

```
                            An Assignment Problem


    ----------------------------------------------------------------------
    |                   |                      grade                      |
    |                   |-------------------------------------------------|
    |                   |  grade1   |  grade2   |  grade3   |  grade4     |
    |-------------------+-----------+-----------+-----------+-------------|
    |machine |customer  |           |           |           |             |
    |--------+----------|           |           |           |             |
    |1       |1         |        . |    100.00|    150.00|    150.00|
    |        |----------+-----------+-----------+-----------+-------------|
    |        |2         |        . |        . |    300.00|        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |3         |        . |        . |    256.72|    210.31|
    |        |----------+-----------+-----------+-----------+-------------|
    |        |4         |        . |        . |    750.00|        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |5         |        . |     92.27|        . |        . |
    |--------+----------+-----------+-----------+-----------+-------------|
    |2       |3         |        . |        . |    143.28|        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |5         |        . |        . |    300.00|        . |
    |--------+----------+-----------+-----------+-----------+-------------|
    |3       |2         |        . |        . |        . |    275.00|
    |        |----------+-----------+-----------+-----------+-------------|
    |        |3         |        . |        . |        . |    289.69|
    |        |----------+-----------+-----------+-----------+-------------|
    |        |4         |        . |        . |        . |    750.00|
    |        |----------+-----------+-----------+-----------+-------------|
    |        |5         |        . |        . |        . |        . |
    |--------+----------+-----------+-----------+-----------+-------------|
    |4       |1         |    100.00|        . |        . |        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |2         |    300.00|    125.00|        . |        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |3         |    400.00|        . |        . |        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |4         |    250.00|        . |        . |        . |
    |        |----------+-----------+-----------+-----------+-------------|
    |        |5         |        . |    507.73|        . |        . |
    ----------------------------------------------------------------------


    (Continued)
```

Output 5.3.2 *continued*

```
                        An Assignment Problem

        ------------------------------------------------
        |                  |              grade         |
        |                  |----------------------------|
        |                  |   grade5    |   grade6     |
        |-----------------+-----------+--------------|
        |machine |customer |             |              |
        |--------+---------|             |              |
        |1        |1        |     175.00|      250.00|
        |         |---------+-----------+--------------|
        |         |2        |          .|           .|
        |         |---------+-----------+--------------|
        |         |3        |          .|           .|
        |         |---------+-----------+--------------|
        |         |4        |          .|           .|
        |         |---------+-----------+--------------|
        |         |5        |          .|           .|
        |--------+---------+-----------+--------------|
        |2        |3        |     340.00|           .|
        |         |---------+-----------+--------------|
        |         |5        |          .|           .|
        |--------+---------+-----------+--------------|
        |3        |2        |     310.00|      325.00|
        |         |---------+-----------+--------------|
        |         |3        |          .|           .|
        |         |---------+-----------+--------------|
        |         |4        |          .|           .|
        |         |---------+-----------+--------------|
        |         |5        |     210.00|      360.00|
        |--------+---------+-----------+--------------|
        |4        |1        |          .|           .|
        |         |---------+-----------+--------------|
        |         |2        |          .|           .|
        |         |---------+-----------+--------------|
        |         |3        |          .|           .|
        |         |---------+-----------+--------------|
        |         |4        |          .|           .|
        |         |---------+-----------+--------------|
        |         |5        |          .|           .|
        ------------------------------------------------
```

Example 5.4: Set Manipulation

This example demonstrates PROC OPTMODEL set manipulation operators. These operators are used to compute the set of primes up to a given limit. This example does not solve an optimization problem, but similar set manipulation could be used to set up an optimization model. Here are the statements:

```
proc optmodel;
   number maxprime;  /* largest number to consider */
   set composites =
       union {i in 3..sqrt(maxprime) by 2} i*i..maxprime by 2*i;
```

```
    set primes = {2} union (3..maxprime by 2 diff composites);
    maxprime = 500;
    put primes;
```

The set composites contains the odd composite numbers up to the value of the parameter maxprime. The even numbers are excluded here to reduce execution time and memory requirements. The UNION aggregation operation is used in the definition to combine the sets of odd multiples of i for $i = 3, 5, \ldots$. Any composite number less than the value of the parameter maxprime has a divisor $\leq \sqrt{\text{maxprime}}$, so the range of i can be limited. The set of multiples of i can also be started at $i \times i$ since smaller multiples are found in the set of multiples for a smaller index.

You can then define the set primes. The odd primes are determined by using the DIFF operator to remove the composites from the set of odd numbers no greater than the parameter maxprime. The UNION operator adds the single even prime, 2, to the resulting set of primes.

The PUT statement produces the result in Output 5.4.1.

Output 5.4.1 Primes less than or equal to 500

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,
107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,
223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,
337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,
457,461,463,467,479,487,491,499}
```

Note that you were able to delay the definition of the value of the parameter maxprime until just before the PUT statement. Since the defining expressions of the SET declarations are handled symbolically, the value of maxprime is not necessary until you need the value of the set primes. Because the sets composites and primes are defined symbolically, their values reflect any changes to the parameter maxprime. You can see this update by appending the following statements to the preceding statements:

```
    maxprime = 50;
    put primes;
```

The additional statements produce the results in Output 5.4.2. The value of the set primes has been recomputed to reflect the change to the parameter maxprime.

Output 5.4.2 Primes less than or equal to 50

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
```

Example 5.5: Multiple Subproblems

Many important optimization problems cannot be solved directly using a standard solver, either because the problem has constraints that cannot be modeled directly or because the resulting model would be too large to be practical. For these types of problems, you can use PROC OPTMODEL to synthesize solution methods by using a combination of the existing solvers and the modeling language programming constructs. This example demonstrates the use of multiple subproblems to solve the cutting stock problem.

The cutting stock problem determines how to efficiently cut raw stock into finished widths based on the demands for the final product. Consider the example from page 195 of Chvátal (1983), where raw stock has a width of 100 inches and the demands are shown in Table 5.15.

Table 5.15   Cutting Stock Demand

Finished Width	Demand
45 inches	97
35 inches	610
31 inches	395
14 inches	211

A portion of the demand can be satisfied using a cutting pattern. For example, with the demands in Table 5.15 a possible pattern cuts one final of width 35 inches, one final of width 31 inches, and two finals of width 14 inches. This gives:

$$100 = 0 * 45 + 1 * 35 + 1 * 31 + 2 * 14 + \text{waste}.$$

The cutting stock problem can be formulated as follows, where x_j represents the number of times pattern j appears, a_{ij} represents the number of times demand item i appears in pattern j, d_i is the demand for item i, w_i is the width of item i, N represents the set of patterns, M represents the set of items, and W is the width of the raw stock:

$$
\begin{array}{ll}
\text{minimize} & \sum_{j \in N} x_j \\
\text{subject to} & \sum_{j \in N} a_{ij} x_j \geq d_i \quad \text{for all } i \in M \\
& x_j \text{ integer}, \ \geq 0 \qquad \text{for all } j \in N
\end{array}
$$

Also for each feasible pattern j you must have:

$$\sum_{i \in M} w_i a_{ij} \leq W$$

The difficulty with this formulation is that the number of patterns can be very large, with too many columns x_j to solve efficiently. But you can use column generation, as described on page 198 of Chvátal (1983), to generate a smaller set of useful patterns, starting from an initial feasible set.

The dual variables, π_i, of the demand constraints are used to price out the columns. From linear programming (LP) duality theory, a column that improves the primal solution must have a negative reduced cost. For this problem the reduced cost for column x_j is

$$1 - \sum_{i \in M} \pi_i a_{ij}$$

Using this observation produces a knapsack subproblem:

$$
\begin{array}{ll}
\text{minimize} & 1 - \sum_{i \in M} \pi_i a_i \\
\text{subject to} & \sum_{i \in M} w_i a_i \leq W \\
& a_i \text{ integer}, \ \geq 0 \qquad \text{for all } j \in N
\end{array}
$$

where the objective is equivalent to:

maximize $\sum_{i \in M} \pi_i a_i$

The pattern is useful if the associated reduced cost is negative:

$$1 - \sum_{i \in M} \pi_i a_i^* < 0$$

So you can use the following steps to generate the patterns and solve the cutting stock problem:

1. Initialize a set of trivial (one item) patterns.

2. Solve the problem using the LP solver.

3. Minimize the reduced cost using a knapsack solver.

4. Include the new pattern if the reduced cost is negative.

5. Repeat steps 2 through 4 until there are no more negative reduced cost patterns.

These steps are implemented in the following statements. Since adding columns preserves primal feasibility, the statements use the primal simplex solver to take advantage of a warm start. The statements also solve the LP relaxation of the problem, but you want the integer solution. So the statements finish by using the MILP solver with the integer restriction applied. The result is not guaranteed to be optimal, but lower and upper bounds can be provided for the optimal objective.

```
/* cutting-stock problem */
/* uses delayed column generation from
   Chvatal's Linear Programming (1983), page 198 */

%macro csp(capacity);
proc optmodel printlevel=0;
   /* declare parameters and sets */
   num capacity = &capacity;
   set ITEMS;
   num demand {ITEMS};
   num width  {ITEMS};
   num num_patterns init card(ITEMS);
   set PATTERNS = 1..num_patterns;
   num a {ITEMS, PATTERNS};
   num c {ITEMS} init 0;
   num epsilon = 1E-6;

   /* read input data */
   read data indata into ITEMS=[_N_] demand width;

   /* generate trivial initial columns */
   for {i in ITEMS, j in PATTERNS}
      a[i,j] = (if (i = j) then floor(capacity/width[i]) else 0);
```

```
/* define master problem */
var x {PATTERNS} >= 0 integer;
minimize NumberOfRaws = sum {j in PATTERNS} x[j];
con demand_con {i in ITEMS}:
   sum {j in PATTERNS} a[i,j] * x[j] >= demand[i];
problem Master include x NumberOfRaws demand_con;

/* define column generation subproblem */
var y {ITEMS} >= 0 integer;
maximize KnapsackObjective = sum {i in ITEMS} c[i] * y[i];
con knapsack_con:
   sum {i in ITEMS} width[i] * y[i] <= capacity;
problem Knapsack include y KnapsackObjective knapsack_con;

/* main loop */
do while (1);
   print _page_ a;

   /* master problem */
   /* minimize sum_j x[j]
      subj. to sum_j a[i,j] * x[j] >= demand[i]
              x[j] >= 0 and integer */
   use problem Master;
   put "solve master problem";
   solve with lp relaxint /
      presolver=none solver=ps basis=warmstart printfreq=1;
   print x;
   print demand_con.dual;
   for {i in ITEMS} c[i] = demand_con[i].dual;

   /* knapsack problem */
   /* maximize sum_i c[i] * y[i]
      subj. to sum_i width[i] * y[i] <= capacity
              y[i] >= 0 and integer */
   use problem Knapsack;
   put "solve column generation subproblem";
   solve with milp / printfreq=0;
   for {i in ITEMS} y[i] = round(y[i]);
   print y;
   print KnapsackObjective;

   if KnapsackObjective <= 1 + epsilon then leave;

   /* include new pattern */
   num_patterns = num_patterns + 1;
   for {i in ITEMS} a[i,num_patterns] = y[i];
end;

/* solve IP, using rounded-up LP solution as warm start */
use problem Master;
for {j in PATTERNS} x[j] = ceil(x[j].sol);
put "solve (restricted) master problem as IP";
solve with milp / primalin;
```

```
        /* cleanup solution and save to output data set */
        for {j in PATTERNS} x[j] = round(x[j].sol);
        create data solution from [pattern]={j in PATTERNS: x[j] > 0}
            raws=x {i in ITEMS} <col('i'||i)=a[i,j]>;
    quit;
    %mend csp;

    /* Chvatal, p.199 */
    data indata;
        input demand width;
        datalines;
    78 25.5
    40 22.5
    30 20
    30 15
    ;
    run;
    %csp(91)
    /* LP solution is integer */

    /* Chvatal, p.195 */
    data indata;
        input demand width;
        datalines;
     97 45
    610 36
    395 31
    211 14
    ;
    run;
    %csp(100)
    /* LP solution is fractional */
```

The contents of the output data set for the second problem instance are shown in Output 5.5.1.

Output 5.5.1 Cutting Stock Solution

Obs	pattern	raws	i1	i2	i3	i4
1	1	49	2	0	0	0
2	2	100	0	2	0	0
3	5	106	0	2	0	2
4	6	198	0	1	2	0

Example 5.6: Traveling Salesman Problem

This example demonstrates the use of the SUBMIT statement to execute a block of SAS statements from within a PROC OPTMODEL session. In this case, the SUBMIT block calls the GPLOT procedure to display intermediate results during the solution of an instance of the traveling salesman problem (TSP). The problem is described in Example 7.4. See the "Examples" section in Chapter 2, "The OPTNET Procedure" (*SAS/OR*

User's Guide: Network Optimization Algorithms). for an example of how to use PROC OPTNET to solve the TSP.

The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "&tsplib";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;
```

The following macro generates plots of the solution and objective value:

```
%macro plotTSP;
   /* create Annotate data set to draw subtours */
   data anno(drop=xi yi xj yj);
      %SYSTEM(2, 2, 2);
      set solData(keep=xi yi xj yj);
      %LINE(xi, yi, xj, yj, *, 1, 1);
   run;

   title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
   title2;

%* sas;
   %put iteration i: &i.;
%* tex;
   proc gplot data=tspData anno=anno;
%* sas;
   proc gplot data=tspData anno=anno gout=tspg;
      axis1 label=none;
      symbol1 value=dot interpol=none
      pointlabel=("#var1" nodropcollisions height=1) cv=black;
      plot var3*var2 / haxis=axis1 vaxis=axis1;
   run;
   quit;
%mend plotTSP;

%annomac;
```

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints. The SUBMIT statement calls the **%plotTSP** macro to plot the solution and objective value at each stage.

```
/* iterative solution using the subtour formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i > j};
   num xc {VERTICES};
   num yc {VERTICES};

   num numsubtour init 0;
```

```
    set SUBTOUR {1..numsubtour};

    /* read in the instance and customer coordinates (xc, yc) */
    read data tspData into VERTICES=[var1] xc=var2 yc=var3;

    /* the cost is the euclidean distance rounded to the nearest integer */
    num c {<i,j> in EDGES}
       init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

    var x {EDGES} binary;

    /* minimize the total cost */
    min obj =
        sum {<i,j> in EDGES} c[i,j] * x[i,j];

    /* each vertex has exactly one in-edge and one out-edge */
    con two_match {i in VERTICES}:
       sum {j in VERTICES: i > j} x[i,j]
       + sum {j in VERTICES: i < j} x[j,i] = 2;

    /* no subtours (these constraints are generated dynamically) */
    con subtour_elim {s in 1..numsubtour}:
       sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
          or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

    /* this starts the algorithm to find violated subtours */
    set <num,num> EDGES1;
    set INITVERTICES = setof{<i,j> in EDGES1} i;
    set VERTICES1;
    set NEIGHBORS;
    set <num,num> CLOSURE;
    num component {INITVERTICES};
    num numcomp  init 2;
    num iter     init 1;
    call symput('i',trim(left(put(round(iter),best.))));
    num numiters init 1;

    /* initial solve with just matching constraints */
    solve;
    call symput(compress('obj'||put(iter,best.)),
                trim(left(put(round(obj),best.))));

    /* create a data set for use by PROC GPLOT */
    create data solData from
       [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
       xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];
    submit;
       %plotTSP;
    endsubmit;
```

```
   /* while the solution is disconnected, continue */
   do while (numcomp > 1);
      iter = iter + 1;
      call symput('i',trim(left(put(round(iter),best.))));

      /* find connected components of support graph   */
      EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
      EDGES1 = EDGES1 union {setof {<i,j> in EDGES1} <j,i>};
      VERTICES1 = INITVERTICES;
      CLOSURE = EDGES1;

      for {i in INITVERTICES} component[i] = 0;
      for {i in VERTICES1} do;
         NEIGHBORS = slice(<i,*>,CLOSURE);
         CLOSURE = CLOSURE union (NEIGHBORS cross NEIGHBORS);
      end;

      numcomp = 0;
      do while (card(VERTICES1) > 0);
         numcomp = numcomp + 1;
         for {i in VERTICES1} do;
            NEIGHBORS = slice(<i,*>,CLOSURE);
            for {j in NEIGHBORS} component[j] = numcomp;
            VERTICES1 = VERTICES1 diff NEIGHBORS;
            leave;
         end;
      end;

      if numcomp = 1 then leave;

      numiters = iter;
      numsubtour = numsubtour + numcomp;
      for {comp in 1..numcomp} do;
         SUBTOUR[numsubtour-numcomp+comp]
            = {i in VERTICES: component[i] = comp};
      end;

      solve;
      call symput(compress('obj'||put(iter,best.)),
                  trim(left(put(round(obj),best.))));

      /* create a data set for use by PROC GPLOT */
      create data solData from
         [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
         xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];

      call symput('numiters',put(numiters,best.));
      submit;
         %plotTSP;
      endsubmit;
   end;
quit;
```

The plot in Output 5.6.1 shows the solution and objective value at each stage. Each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

Output 5.6.1 Iterative Solution of Traveling Salesman Problem
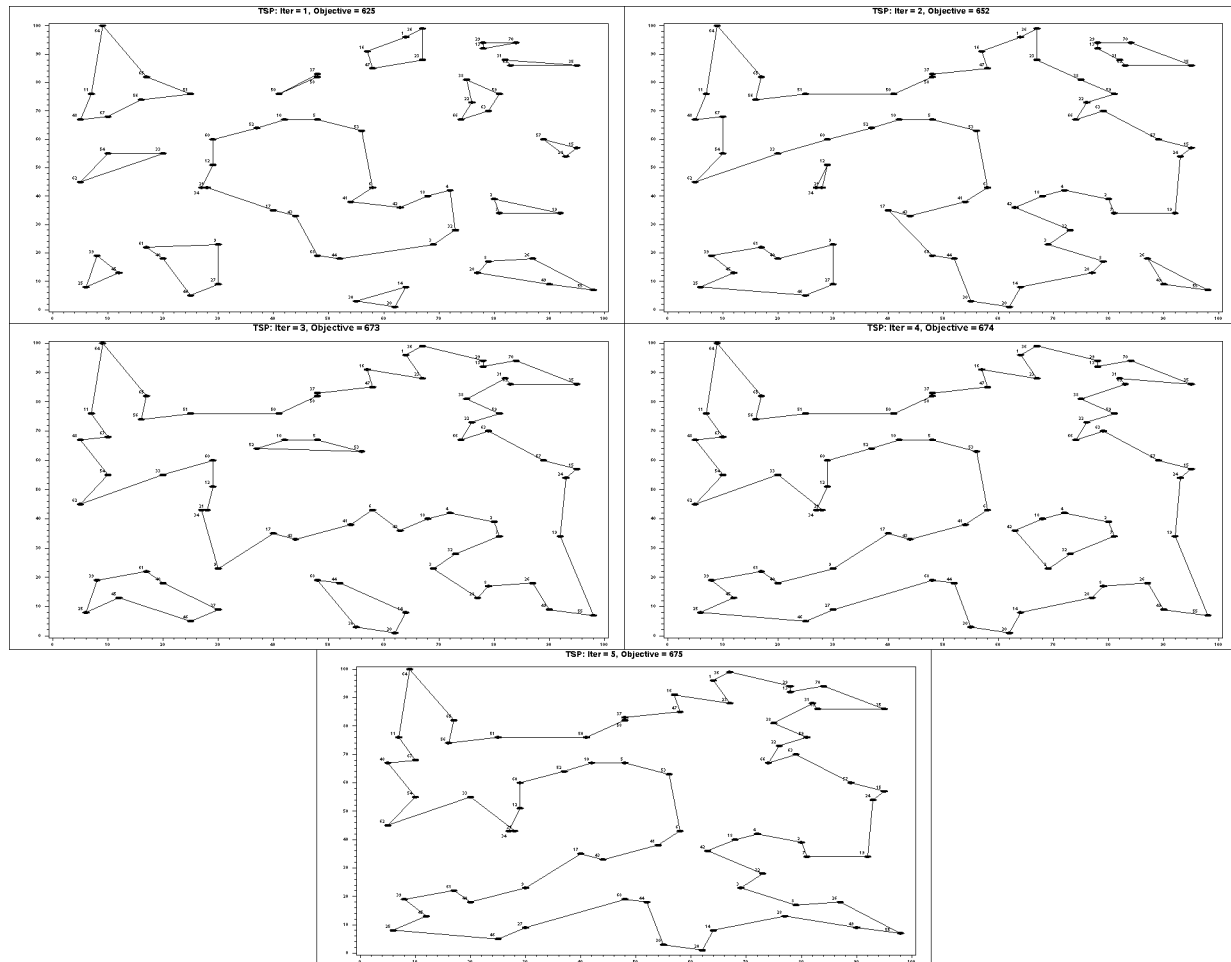
Rewriting PROC NLP Models for PROC OPTMODEL

This section covers techniques for converting PROC NLP models to PROC OPTMODEL models.

To illustrate the basics, consider the following first version of the PROC NLP model for the example "Simple Pooling Problem" in Chapter 7, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*):

```
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 were split into individual declarations because PROC OPT-MODEL 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 PROC 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, a coefficient can be a named parameter 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.

In the original PROC NLP statements 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 statements, 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 PROC 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 this consists of messages from the solver (such as the termination reason) and 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 the example "Chemical Equilibrium" in Chapter 7, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), is used to illustrate how to convert PROC NLP code that handles 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 PROC 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 5.6.2.

Output 5.6.2 PROC OPTMODEL Output

```
                        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


                  [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 5.6.3. The indices of variable x are now strings that represent the actual compounds.

Output 5.6.3 PROC OPTMODEL Output with Strings

```
         [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
```

References

Abramowitz, M. and Stegun, I. A. (1972), *Handbook of Mathematical Functions*, New York: Dover Publications.

Bazaraa, M. S., Sherali, H. D., and Shetty, C. M. (1993), *Nonlinear Programming: Theory and Algorithms*, New York: John Wiley & Sons.

Chvátal, V. (1983), *Linear Programming*, New York: W. H. Freeman.

Dennis, J. E. and Schnabel, R. B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall.

Fletcher, R. (1987), *Practical Methods of Optimization*, 2nd Edition, Chichester, UK: John Wiley & Sons.

Nocedal, J. and Wright, S. J. (1999), *Numerical Optimization*, New York: Springer-Verlag.

Chapter 6
The Linear Programming Solver

Contents

Overview: LP Solver

The OPTMODEL procedure provides a framework for specifying and solving linear programs (LPs). A standard linear program has the following formulation:

$$
\begin{array}{rl}
\min & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\
\text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{array}
$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables

The following LP solvers are available in the OPTMODEL procedure:

- primal simplex solver

- dual simplex solver

- network simplex solver

- interior point solver

The primal and dual simplex solvers implement the two-phase simplex method. In phase I, the solver tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the solver enters phase II to solve the original LP. The network simplex solver extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point solver implements a primal-dual predictor-corrector interior point algorithm. If any of the decision variables are constrained to be integer-valued, then the relaxed version of the problem is solved.

Getting Started: LP Solver

The following example illustrates how you can use the OPTMODEL procedure to solve linear programs. Suppose you want to solve the following problem:

$$
\begin{array}{rlrlrlrlcr}
\max & x_1 & + & x_2 & + & x_3 & & & & \\
\text{subject to} & 3x_1 & + & 2x_2 & - & x_3 & & & \leq & 1 \\
& -2x_1 & - & 3x_2 & + & 2x_3 & & & \leq & 1 \\
& & & x_1, & & x_2, & & x_3 & \geq & 0
\end{array}
$$

You can use the following statements to call the OPTMODEL procedure for solving linear programs:

```
proc optmodel;
   var x{i in 1..3} >= 0;
   max f =     x[1] +   x[2] +   x[3];
   con c1:  3*x[1] + 2*x[2] -   x[3] <= 1;
   con c2: -2*x[1] - 3*x[2] + 2*x[3] <= 1;
   solve with lp / algorithm = ps presolver = none logfreq = 1;
   print x;
quit;
```

The optimal solution and the optimal objective value are displayed in Figure 6.1.

Figure 6.1 Solution Summary

```
                    The OPTMODEL Procedure

                         Problem Summary

           Objective Sense              Maximization
           Objective Function                      f
           Objective Type                     Linear

           Number of Variables                     3
           Bounded Above                           0
           Bounded Below                           3
           Bounded Below and Above                 0
           Free                                    0
           Fixed                                   0

           Number of Constraints                   2
           Linear LE (<=)                          2
           Linear EQ (=)                           0
           Linear GE (>=)                          0
           Linear Range                            0

           Constraint Coefficients                 6


                      Performance Information

            Execution Mode        Single-Machine
            Number of Threads                   1
```

Figure 6.1 *continued*

```
                    Solution Summary

          Solver                            LP
          Algorithm               Primal Simplex
          Objective Function                   f
          Solution Status                Optimal
          Objective Value                      8

          Primal Infeasibility                 0
          Dual Infeasibility                   0
          Bound Infeasibility                  0

          Iterations                           5
          Presolve Time                     0.00
          Solution Time                     0.05



                    [1]     x

                     1       0
                     2       3
                     3       5
```

The iteration log displaying problem statistics, progress of the solution, and the optimal objective value is shown in Figure 6.2.

Figure 6.2 Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 2 linear constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                      Objective              Entering     Leaving
      Phase Iteration    Value        Time    Variable    Variable
       P 1          1   0.000000E+00     0
       P 2          2   0.000000E+00     0     x[3]        c2 (S)
       P 2          3   5.000000E-01     0     x[2]        c1 (S)
       P 2          4   8.000000E+00     0
       P 2          5   8.000000E+00     0
NOTE: Optimal.
NOTE: Objective = 8.
NOTE: The Primal Simplex solve time is 0.05 seconds.
```

Syntax: LP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH LP *< / options >* **;**

Functional Summary

Table 6.1 summarizes the list of options available for the SOLVE WITH LP statement, classified by function.

Table 6.1 Options for the LP Solver

Description	Option
Solver Options	
Specifies the type of solver	ALGORITHM=
Specifies the type of solver called after network simplex	ALGORITHM2=
Enables or disables IIS detection	IIS=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Specifies units of CPU time or real time	TIMETYPE=
Simplex Algorithm Options	
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=
Specifies the queue size for determining entering variable	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Interior Point Algorithm Options	
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=
Decomposition Algorithm Options	
Enables decomposition algorithm and specifies general control options	DECOMP=()
Specifies options for the master problem	DECOMP_MASTER=()
Specifies options for the subproblem	DECOMP_SUBPROB=()

LP Solver Options

This section describes the options recognized by the LP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the LP solver is explicitly specified using a WITH clause.

If the LP solver terminates before reaching an optimal solution, an intermediate solution is available. You can access this solution by using the .sol variable suffix in the OPTMODEL procedure. See the section "Suffixes" on page 131 for details.

Solver Options

IIS=*number* | *string*

      specifies whether the LP solver attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 6.2 describes the valid values of the IIS= option.

Table 6.2   Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

If an IIS is found, information about the infeasibilities can be found in the .status values of the constraints and variables. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 200 for details about the IIS= option. See "Suffixes" on page 131 for details about the .status suffix.

ALGORITHM=*option*

SOLVER=*option*

SOL=*option*

      specifies one of the following LP solvers:

Option	**Description**
PRIMAL (PS)	Uses primal simplex solver.
DUAL (DS)	Uses dual simplex solver.
NETWORK (NS)	Uses network simplex solver.
INTERIORPOINT (IP)	Uses interior point solver.
CONCURRENT (CON) (experimental)	Uses several different algorithms in parallel.

The valid abbreviated value for each option is indicated in parentheses. By default, the dual simplex solver is used.

ALGORITHM2=*option*

SOLVER2=*option*

      specifies one of the following LP solvers if ALGORITHM=NS:

Option	**Description**
PRIMAL (PS)	Uses primal simplex solver (after network simplex).
DUAL (DS)	Uses dual simplex solver (after network simplex).

The valid abbreviated value for each option is indicated in parentheses. By default, the LP solver decides which algorithm is best to use after calling the network simplex solver on the extracted network.

Presolve Options

PRESOLVER=*number | string*

specifies one of the following presolve options:

number	string	Description
0	NONE	Disables presolver.
−1	AUTOMATIC	Applies presolver by using default setting.
1	BASIC	Performs basic presolve like removing empty rows, columns, and fixed variables.
2	MODERATE	Performs basic presolve and apply other inexpensive presolve techniques.
3	AGGRESSIVE	Performs moderate presolve and apply other aggressive (but expensive) presolve techniques.

The default option is AUTOMATIC. See the section "Presolve" on page 192 for details.

Control Options

FEASTOL=ϵ

specifies the feasibility tolerance, $\epsilon \in$[1E–9, 1E–4], for determining the feasibility of a variable. The default value is 1E–6.

LOGFREQ=*k*

PRINTFREQ=*k*

specifies that the printing of the solution progress to the iteration log is to occur after every *k* iterations. The print frequency, *k*, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

The value $k = 0$ disables the printing of the progress of the solution. If the primal or dual simplex algorithms are used, the default value of this option is determined dynamically according to the problem size. If the network simplex algorithm is used, the default value of this option is 10,000. If the interior point algorithm is used, the default value of this option is 1.

LOGLEVEL=*number | string*

PRINTLEVEL2=*number | string*

controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. Table 6.6 describes the valid values for this option.

Table 6.6 Values for LOGLEVEL= Option

number	string	Description
0	NONE	Turns off all solver-related messages to SAS log.
1	BASIC	Displays a solver summary after stopping.
2	MODERATE	Prints a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

Table 6.6   (continued)

number	string	Description
3	AGGRESSIVE	Prints a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

The default value is MODERATE.

MAXITER=*k*

specifies the maximum number of iterations. The value *k* can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the solver called after network simplex (either primal or dual simplex).

MAXTIME=*t*

specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*ϵ*

specifies the optimality tolerance, $\epsilon \in [1E{-}9, 1E{-}4]$, for declaring optimality. The default value is 1E–6.

TIMETYPE=*number* | *string*

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 6.7 describes the valid values of the TIMETYPE= option.

Table 6.7   Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The "Optimization Statistics" table, an output of the OPTMODEL procedure if you specify the PRINTLEVEL=2 option in the PROC OPTMODEL statement, also includes the same time units for "Presolver Time" and "Solver Time." The other times (such as "Problem Generation Time") in the "Optimization Statistics" table are always CPU times.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement of the OPTMODEL procedure. See the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and NODES= options.

If you specify a value greater than 1 for either the NTHREADS= or NODES= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and NODES= options, the default value of the TIMETYPE= option is CPU.

Simplex Algorithm Options

BASIS=_number | string_

specifies the following options for generating an initial basis:

number	string	Description
0	CRASH	Generate an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
1	SLACK	Generate an initial basis by using all slack variables.
2	WARMSTART	Start the primal and dual simplex solvers with available basis.

The default option for the primal simplex solver is CRASH (0). The default option for the dual simplex solver is SLACK(1). For network simplex, this option has no effect.

PRICETYPE=_number | string_

specifies one of the following pricing strategies for the primal and dual simplex solvers:

number	string	Description
0	HYBRID	Use hybrid Devex and steepest-edge pricing strategies. Available for primal simplex solver only.
1	PARTIAL	Use partial pricing strategy. Optionally, you can specify QUEUESIZE=. Available for primal simplex solver only.
2	FULL	Use the most negative reduced cost pricing strategy.
3	DEVEX	Use Devex pricing strategy.
4	STEEPESTEDGE	Use steepest-edge pricing strategy.

The default pricing strategy for the primal simplex solver is HYBRID and that for the dual simplex solver is STEEPESTEDGE. For the network simplex solver, this option applies only to the solver specified by the ALGORITHM2= option. See the section "Pricing Strategies for the Primal and Dual Simplex Solvers" on page 192 for details.

QUEUESIZE=k

specifies the queue size, $k \in [1, n]$, where n is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

SCALE=_number | string_

specifies one of the following scaling options:

number	string	Description
0	NONE	Disable scaling.
−1	AUTOMATIC	Automatically apply scaling procedure if necessary.

The default option is AUTOMATIC.

Interior Point Algorithm Options

CROSSOVER=*number | string*

specifies whether to convert the interior point solution to a basic simplex solution. The values of this option are:

number	string	Description
0	OFF	Disable crossover.
1	ON	Apply the crossover algorithm to the interior point solution.

If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution. The default value of the CROSSOVER= option is OFF.

STOP_DG=δ

specifies the desired relative duality gap, $\delta \in$ [1E–9, 1E–4]. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 193 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 193 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in$[1E–9, 1E–4]. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 193 for details.

Decomposition Algorithm Options

The following options are available for the decomposition algorithm in the LP solver. For information about the decomposition algorithm, see Chapter 13, "The Decomposition Algorithm."

DECOMP=(*options***)**

enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

DECOMP_MASTER=(*options***)**

specifies options for the master problem. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

DECOMP_SUBPROB=(*options***)**
> specifies option for the subproblem. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

Details: LP Solver

Presolve

Presolve in the simplex LP solvers of PROC OPTMODEL uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.

Pricing Strategies for the Primal and Dual Simplex Solvers

Several pricing strategies for the primal and dual simplex solvers are available. Pricing strategies determine which variable enters the basis at each simplex pivot. These can be controlled by specifying the PRICETYPE= option.

The primal simplex solver has the following five pricing strategies:

PARTIAL   scans a queue of decision variables to find an entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.

FULL   uses Dantzig's most violated reduced cost rule (Dantzig 1963). It compares the reduced cost of all decision variables, and selects the variable with the most violated reduced cost as the entering variable.

DEVEX   implements the Devex pricing strategy developed by Harris (1973).

STEEPESTEDGE   uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).

HYBRID   uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex solver has only three pricing strategies available: FULL, DEVEX, and STEEPESTEDGE.

The Network Simplex Algorithm

The network simplex solver in PROC OPTMODEL attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.

2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.

3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex solver on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where N denotes the nodes and A denotes the arcs of the graph. The decision variable x_{ij} denotes the amount of flow sent between node i and node j. The cost per unit of flow on the arcs is designated by c_{ij}, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at each node is designated as b_i, where $b_i > 0$ denotes a supply node and $b_i < 0$ denotes a demand node. The corresponding linear programming problem is as follows:

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i \quad \forall i \in N \\
& x_{ij} \leq u_{ij} \quad \forall (i,j) \in A \\
& x_{ij} \geq l_{ij} \quad \forall (i,j) \in A.
\end{aligned}
$$

The network simplex algorithm used in PROC OPTMODEL is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph G is disconnected. In this case, the problem can be decomposed into its weakly connected components, and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the solver to use after network simplex. However, you can override this selection with the ALGORITHM2= option.

The network simplex algorithm can be more efficient than the other solvers on problems that have a large network substructure. The size of this network structure can be seen in the log.

The Interior Point Algorithm

The interior point LP solver in PROC OPTMODEL implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

$$
\begin{aligned}
\min \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\
& \mathbf{x} \geq \mathbf{0}
\end{aligned}
$$

The corresponding dual is as follows:

$$
\begin{aligned}
\max \quad & \mathbf{b}^{\mathrm{T}}\mathbf{y} \\
\text{subject to} \quad & \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} = \mathbf{c} \\
& \mathbf{y} \geq \mathbf{0} \\
& \mathbf{w} \geq \mathbf{0}
\end{aligned}
$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$
\begin{aligned}
\mathbf{Ax - s} &= \mathbf{b} \quad \text{(Primal Feasibility)} \\
\mathbf{A}^T\mathbf{y + w} &= \mathbf{c} \quad \text{(Dual Feasibility)} \\
\mathbf{WXe} &= \mathbf{0} \quad \text{(Complementarity)} \\
\mathbf{SYe} &= \mathbf{0} \quad \text{(Complementarity)} \\
\mathbf{x,\ y,\ w,\ s} &\geq \mathbf{0}
\end{aligned}
$$

where $\mathbf{e} \equiv (1, \ldots, 1)^T$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding x, y, w, and s on the main diagonal and zero elsewhere, as in the following example:

$$
\mathbf{X} \equiv \begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations representing the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

$$
\begin{bmatrix}
\mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\
\mathbf{A}^T & -\mathbf{X}^{-1}\mathbf{W}
\end{bmatrix}
\begin{bmatrix}
\Delta \mathbf{y} \\
\Delta \mathbf{x}
\end{bmatrix}
=
\begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
$$

where $\Delta \mathbf{x}$ and $\Delta \mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively; Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The interior point solver uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point solver is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore it is of interest to observe the following four measures:

- Relative primal infeasibility measure α:

$$
\alpha = \frac{\|\mathbf{Ax - b - s}\|_2}{\|\mathbf{b}\|_2 + 1}
$$

- Relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{c} - \mathbf{A}^\mathrm{T}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- Relative duality gap δ:

$$\delta = \frac{|\mathbf{c}^\mathrm{T}\mathbf{x} - \mathbf{b}^\mathrm{T}\mathbf{y}|}{|\mathbf{c}^\mathrm{T}\mathbf{x}| + 1}$$

- Absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

where $\|v\|_2$ is the Euclidean norm of the vector v. These measures are displayed in the iteration log.

Iteration Log for the Primal and Dual Simplex Solvers

The primal and dual simplex solvers implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When the LOGFREQ= option has a value of 1, the following information is printed in the iteration log:

Algorithm	indicates which simplex method is running by printing the letter P (primal) or D (dual).
Phase	indicates whether the solver is in phase I or phase II of the simplex method.
Iteration	indicates the iteration number.
Objective Value	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)". If the entering nonbasic variable has distinct, finite lower and upper bounds, then a "bound swap" can take place in the primal simplex method.
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)". The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value larger than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.

The behavior of objective values in the iteration log depends on both the current phase and the chosen solver. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method, phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality

for the perturbed problem, the LP solver solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual simplex method to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

Iteration Log for the Network Simplex Solver

After finding the embedded network and formulating the appropriate relaxation, the network simplex solver uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log will show the progress of the simplex algorithm. The following information is displayed in the iteration log:

Iteration	indicates the iteration number.
PrimalObj	indicates the primal objective value of the current solution.
Primal Infeas	indicates the maximum primal infeasibility of the current solution.
Time	indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the LOGFREQ= option. The default value of the LOGFREQ= option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

Component	indicates the component number being processed.
Nodes	indicates the number of nodes in this component.
Arcs	indicates the number of arcs in this component.
Iterations	indicates the number of simplex iterations needed to solve this component.
Time	indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the LOGFREQ= option. In this case, the default value of the LOGFREQ= option is determined by the size of the network.

The LOGLEVEL= option adjusts the amount of detail shown. By default, LOGLEVEL=MODERATE and reports as in the preceding description. If LOGLEVEL=NONE, no information is shown. If LOGLEVEL=BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If LOGLEVEL=AGGRESSIVE, in the case of one component, the log displays as in the preceding description; in the case of multiple components, for each component, a separate simplex iteration log is displayed.

Iteration Log for the Interior Point Solver

The interior point solver implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter	indicates the iteration number
Complement	indicates the (absolute) complementarity
Duality Gap	indicates the (relative) duality gap
Primal Infeas	indicates the (relative) primal infeasibility measure
Bound Infeas	indicates the (relative) bound infeasibility measure
Dual Infeas	indicates the (relative) dual infeasibility measure

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the option MAXITER= or MAXTIME=. If the complementarity and/or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Iteration Log for the Crossover Algorithm

The crossover algorithm takes an optimal solution from the interior point solver and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When LOGFREQ=1, the following information is printed in the iteration log:

Phase	indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.
Iteration	indicates the iteration number.
Objective Value	indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.
Time	indicates the time elapsed (in seconds) since the beginning of the crossover algorithm.
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)".
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)".

When you omit the LOGFREQ= option or specify a value greater than 1, only the phase, iteration, objective value, and time information are printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.

Concurrent LP (Experimental)

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The LP solver automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the solver runs all four standard algorithms. When the first algorithm finishes, the LP solver returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of DETERMINISTIC for the PARALLELMODE= option in the PERFORMANCE statement in the OPTMODEL procedure, the algorithm for which the results are returned is not necessarily the one that finished first. The LP solver deterministically selects the algorithm for which the results are returned. For more information about the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 28. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. See the section "Iteration Log for the Primal and Dual Simplex Solvers" on page 195 for more information about this iteration log. Upon termination, the solver displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the LP solver displays the iteration logs for all algorithms that were run concurrently.

If you specify PRINTLEVEL=2 in the PROC OPTMODEL statement and ALGORITHM=CON in the SOLVE WITH LP statement, the LP solver produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table "ProblemStatistics" to be generated when the LP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 6.3 demonstrates the contents of the ODS table "ProblemStatistics."

Figure 6.3 ODS Table ProblemStatistics

```
                    The OPTMODEL Procedure

                     Problem Statistics

        Number of Constraint Matrix Nonzeros          6
        Maximum Constraint Matrix Coefficient         3
        Minimum Constraint Matrix Coefficient         1
        Average Constraint Matrix Coefficient    2.1666666667

        Number of Objective Nonzeros                  3
        Maximum Objective Coefficient                 1
        Minimum Objective Coefficient                 1
        Average Objective Coefficient                 1

        Number of RHS Nonzeros                        2
        Maximum RHS                                   1
        Minimum RHS                                   1
        Average RHS                                   1

        Maximum Number of Nonzeros per Column         2
        Minimum Number of Nonzeros per Column         2
        Average Number of Nonzeros per Column         2

        Maximum Number of Nonzeros per Row            3
        Minimum Number of Nonzeros per Row            3
        Average Number of Nonzeros per Row            3
```

Variable and Constraint Status

Upon termination of the LP solver, the .status suffix of each decision variable and constraint stores information about the status of that variable or constraint. For more information about suffixes in the OPTMODEL procedure, see the section "Suffixes" on page 131.

Variable Status

The .status suffix of a decision variable specifies the status of that decision variable. The suffix can take one of the following values:

B   basic variable

L   nonbasic variable at its lower bound

U   nonbasic variable at its upper bound

F   free variable

S   superbasic variable (a nonbasic variable with a value strictly between its bounds)

I   LP model infeasible (all decision variables have .status equal to I)

For the interior point solver with IIS= OFF, .status is blank.

The following values can appear only if IIS= ON. See the section "Irreducible Infeasible Set" on page 200 for details.

I_L   the lower bound of the variable is violated

I_U   the upper bound of the variable is violated

I_F   the fixed bound of the variable is violated

Constraint Status

The .status suffix of a constraint specifies the status of the slack variable for that constraint. The suffix can take one of the following values:

B   basic variable

L   nonbasic variable at its lower bound

U   nonbasic variable at its upper bound

F   free variable

S   superbasic variable (a nonbasic variable with a value strictly between its bounds)

I   LP model infeasible (all decision variables have .status equal to I)

The following values can appear only if option IIS= ON. See the section "Irreducible Infeasible Set" on page 200 for details.

I_L   the "GE" ($\geq$) condition of the constraint is violated

I_U   the "LE" ($\leq$) condition of the constraint is violated

I_F   the "EQ" (=) condition of the constraint is violated

Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help to isolate the structural infeasibility in an LP.

The IIS=ON option directs the LP solver to search for an IIS in a given LP. You should specify the OPTMODEL option PRESOLVER=NONE when you specify IIS=ON; otherwise the IIS results can be incomplete. The LP presolver is not applied to the problem during the IIS search. If the LP solver detects an IIS, it updates the .status suffix of the decision variables and constraints, and then it stops. The number of iterations that are reported in the macro variable and the ODS table is the total number of simplex iterations. This includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the .status suffixes of variables and constraints. (See the section "Variable and Constraint Status" on page 199 for more information.) For constraints, a status of "I_L", "I_U", or "I_F" indicates, respectively, the "GE" ($\geq$), "LE" ($\leq$), or "EQ" (=) condition is violated. For range constraints, a status of "I_L" or "I_U" indicates, respectively, that the lower or upper bound of the constraint

is violated. For variables, a status of "I_L", "I_U", or "I_F" indicates, respectively, the lower, upper, or fixed bound of the variable is violated. From this information, you can identify names of the constraints (variables) in the IIS as well as the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount will removes the infeasibility; however, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to ∞ or $-\infty$. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can rerun the LP solver with IIS=ON after removing the infeasibility from an IIS. Repeat this process until the LP solver no longer detects an IIS. The resulting problem is feasible. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

The IIS= option in the LP solver uses two different methods to identify an IIS. Based on the result of the initial solve, the *sensitivity filter* removes several constraints and variable bounds at once while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS. After that, the *deletion filter* removes each remaining constraint and variable bound one by one to check which of them are needed to get an infeasible system. This second phase is more time consuming, but it ensures that the IIS set returned by the LP solver is indeed irreducible. The progress of the deletion filter is reported at regular intervals. Occasionally, the sensitivity filter might be called again during the deletion filter to improve performance.

See Example 6.4 for an example demonstrating the use of the IIS= option in locating and removing infeasibilities.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the LP solver is called are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK   The solver terminated normally.

SYNTAX_ERROR   Incorrect syntax was used.

DATA_ERROR   The input data were inconsistent.

OUT_OF_MEMORY   Insufficient memory was allocated to the procedure.

IO_ERROR   A problem occurred in reading or writing data.

SEMANTIC_ERROR   An evaluation error, such as an invalid operand type, occurred.

ERROR   The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

PS   The primal simplex algorithm produced the solution data.

DS   The dual simplex algorithm produced the solution data.

NS   The network simplex algorithm produced the solution data.

IP   The interior point algorithm produced the solution data.

DECOMP   The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or pre-processing.
FEASIBLE	The problem is feasible.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FUNCTION_CALL_LIMIT_REACHED	The solver reached its limit on function evaluations.
INTERRUPTED	The solver was interrupted externally.
FAILED	The solver failed to converge, possibly due to numerical issues.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERA-TION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY

indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point solver, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY
> indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point solver, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY
> indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the lower or upper bounds by the primal solution. For the interior point solver, this term indicates the relative violation of the lower or upper bounds by the primal solution.

DUALITY_GAP
> indicates the (relative) duality gap. This term appears only if the option ALGO-RITHM=INTERIORPOINT is specified in the SOLVE statement.

COMPLEMENTARITY
> indicates the (absolute) complementarity. This term appears only if the option ALGO-RITHM=INTERIORPOINT is specified in the SOLVE statement.

ITERATIONS
> indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2
> indicates the number of simplex iterations performed by the secondary solver. The network simplex solver selects the secondary solver automatically unless a value has been specified for the ALGO-RITHM2= option. When crossover is enabled, the secondary solver is selected automatically. This term appears only if the network simplex solver is used or if crossover is enabled.

PRESOLVE_TIME
> indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
> indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERA-TION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: LP Solver

Example 6.1: Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 6.12.

Table 6.12 Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories	90	120	106	97	130	180

The following SAS code creates the data set fooddata of Table 6.12:

```
data fooddata;
   infile datalines;
   input name $  cost  prot  fat  carb  cal;
   datalines;
Bread   2    4    1    15    90
Milk    3.5  8    5    11.7  120
Cheese  8    7    9    0.4   106
Potato  1.5  1.3  0.1  22.6  97
Fish    11   8    7    0     130
Yogurt  1    9.2  1    17    180
;
```

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can model the problem and solve it by using PROC OPTMODEL as follows:

```
proc optmodel;
   /* declare index set */
   set<str> FOOD;

   /* declare variables */
   var diet{FOOD} >= 0;

   /* objective function */
   num cost{FOOD};
   min f=sum{i in FOOD}cost[i]*diet[i];

   /* constraints */
   num prot{FOOD};
   num fat{FOOD};
   num carb{FOOD};
   num cal{FOOD};
   num min_cal, max_prot, min_carb, min_fat;
   con cal_con: sum{i in FOOD}cal[i]*diet[i] >= 300;
   con prot_con: sum{i in FOOD}prot[i]*diet[i] <= 10;
   con carb_con: sum{i in FOOD}carb[i]*diet[i] >= 10;
   con fat_con: sum{i in FOOD}fat[i]*diet[i] >= 8;

   /* read parameters */
   read data fooddata into FOOD=[name] cost prot fat carb cal;
```

```
/* bounds on variables */
diet['Fish'].lb = 0.5;
diet['Milk'].ub = 1.0;

/* solve and print the optimal solution */
solve with lp/logfreq=1; /* print each iteration to log */
print diet;
```

The optimal solution and the optimal objective value are displayed in Output 6.1.1.

Output 6.1.1 Optimal Solution to the Diet Problem

```
                    The OPTMODEL Procedure

                       Problem Summary

           Objective Sense            Minimization
           Objective Function                    f
           Objective Type                   Linear

           Number of Variables                   6
           Bounded Above                         0
           Bounded Below                         5
           Bounded Below and Above               1
           Free                                  0
           Fixed                                 0

           Number of Constraints                 4
           Linear LE (<=)                        1
           Linear EQ (=)                         0
           Linear GE (>=)                        3
           Linear Range                          0

           Constraint Coefficients              23


                   Performance Information

            Execution Mode       Single-Machine
            Number of Threads                  1


                      Solution Summary

           Solver                               LP
           Algorithm                  Dual Simplex
           Objective Function                    f
           Solution Status                 Optimal
           Objective Value            12.081337881

           Primal Infeasibility                  0
           Dual Infeasibility                    0
           Bound Infeasibility                   0

           Iterations                            7
           Presolve Time                      0.00
           Solution Time                      0.03
```

Output 6.1.1 *continued*

```
        [1]              diet

        Bread     0.000000
        Cheese    0.449499
        Fish      0.500000
        Milk      0.053599
        Potato    1.865168
        Yogurt    0.000000
```

Example 6.2: Reoptimizing the Diet Problem Using BASIS=WARMSTART

After an LP is solved, you might want to change a set of the parameters of the LP and solve the problem again. This can be done efficiently in PROC OPTMODEL. The warm start technique uses the optimal solution of the solved LP as a starting point and solves the modified LP problem faster than it can be solved again from scratch. This example illustrates reoptimizing the diet problem described in Example 6.1.

Assume the optimal solution is found by the SOLVE statement. Instead of quitting the OPTMODEL procedure, you can continue to solve several variations of the original problem.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. You can change the parameters and solve the modified problem by submitting the following code:

```
cost['Cheese']=10; cost['Fish']=7;
solve with lp/presolver=none
             basis=warmstart
             algorithm=ps
             logfreq=1;
print diet;
```

Note that the primal simplex solver is preferred because the primal solution to the last-solved LP is still feasible for the modified problem in this case. The solutions to the original diet problem and the modified problem are shown in Output 6.2.1.

Output 6.2.1 Optimal Solutions to the Original Diet Problem and the Diet Problem with Modified Objective
Function

```
                        The OPTMODEL Procedure

                           Problem Summary

          Objective Sense              Minimization
          Objective Function                      f
          Objective Type                     Linear

          Number of Variables                     6
          Bounded Above                           0
          Bounded Below                           5
          Bounded Below and Above                 1
          Free                                    0
          Fixed                                   0

          Number of Constraints                   4
          Linear LE (<=)                          1
          Linear EQ (=)                           0
          Linear GE (>=)                          3
          Linear Range                            0

          Constraint Coefficients                23


                       Performance Information

            Execution Mode        Single-Machine
            Number of Threads                   1


                          Solution Summary

            Solver                             LP
            Algorithm                 Dual Simplex
            Objective Function                  f
            Solution Status               Optimal
            Objective Value           12.081337881

            Primal Infeasibility                0
            Dual Infeasibility                  0
            Bound Infeasibility                 0

            Iterations                          7
            Presolve Time                    0.00
            Solution Time                    0.03


                   [1]            diet

                   Bread      0.000000
                   Cheese     0.449499
                   Fish       0.500000
                   Milk       0.053599
                   Potato     1.865168
                   Yogurt     0.000000
```

Output 6.2.1 *continued*

```
                        Problem Summary

        Objective Sense            Minimization
        Objective Function                    f
        Objective Type                   Linear

        Number of Variables                   6
        Bounded Above                         0
        Bounded Below                         5
        Bounded Below and Above               1
        Free                                  0
        Fixed                                 0

        Number of Constraints                 4
        Linear LE (<=)                        1
        Linear EQ (=)                         0
        Linear GE (>=)                        3
        Linear Range                          0

        Constraint Coefficients              23


                  Performance Information

         Execution Mode        Single-Machine
         Number of Threads                  1


                     Solution Summary

        Solver                               LP
        Algorithm                 Primal Simplex
        Objective Function                    f
        Solution Status                 Optimal
        Objective Value            10.980335514

        Primal Infeasibility        1.776357E-15
        Dual Infeasibility                    0
        Bound Infeasibility                   0

        Iterations                            1
        Presolve Time                      0.00
        Solution Time                      0.03


              [1]             diet

              Bread     0.000000
              Cheese    0.449499
              Fish      0.500000
              Milk      0.053599
              Potato    1.865168
              Yogurt    0.000000
```

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified

problem by using BASIS=WARMSTART, since the optimal solution to the original problem remains optimal after the objective function is changed.

Output 6.2.2 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
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 6 variables, 4 constraints, and 23 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     0.000000E+00        0
        D 2          7     9.503132E+00        0
NOTE: Optimal.
NOTE: Objective = 9.5031323959.
NOTE: The Dual Simplex solve time is 0.05 seconds.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                          Objective                  Entering         Leaving
      Phase Iteration       Value          Time      Variable         Variable
        P 2          1     1.098034E+01        0
NOTE: Optimal.
NOTE: Objective = 10.980335514.
NOTE: The Primal Simplex solve time is 0.03 seconds.
```

Next, restore the original coefficients of the objective function and consider the case that you need a diet that supplies at least 150 calories. You can change the parameters and solve the modified problem by submitting the following code:

```
cost['Cheese']=8; cost['Fish']=11;cal_con.lb=150;
solve with lp/presolver=none
              basis=warmstart
              algorithm=ds
              logfreq=1;
print diet;
```

Note that the dual simplex solver is preferred because the dual solution to the last-solved LP is still feasible for the modified problem in this case. The solution is shown in Output 6.2.3.

Output 6.2.3 Optimal Solution to the Diet Problem with Modified RHS

```
                        The OPTMODEL Procedure

                          Problem Summary

        Objective Sense              Minimization
        Objective Function                      f
        Objective Type                     Linear

        Number of Variables                     6
        Bounded Above                           0
        Bounded Below                           5
        Bounded Below and Above                 1
        Free                                    0
        Fixed                                   0

        Number of Constraints                   4
        Linear LE (<=)                          1
        Linear EQ (=)                           0
        Linear GE (>=)                          3
        Linear Range                            0

        Constraint Coefficients                23


                       Performance Information

          Execution Mode        Single-Machine
          Number of Threads                   1


                         Solution Summary

          Solver                             LP
          Algorithm                 Dual Simplex
          Objective Function                  f
          Solution Status               Optimal
          Objective Value          9.1744131985

          Primal Infeasibility                0
          Dual Infeasibility                  0
          Bound Infeasibility                 0

          Iterations                          6
          Presolve Time                    0.00
          Solution Time                    0.05


                    [1]           diet

                  Bread     0.00000
                  Cheese    0.18481
                  Fish      0.50000
                  Milk      0.56440
                  Potato    0.14702
                  Yogurt    0.00000
```

The following iteration log indicates that it takes the LP solver just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.

Output 6.2.4 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: No basis information is available.  The BASIS=WARMSTART option is ignored.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective            Entering         Leaving
     Phase Iteration        Value       Time    Variable        Variable
       D 1           1    0.000000E+00      0
       D 2           2    5.500000E+00      0    diet[Milk]
     fat_con (S)
       D 2           3    8.650000E+00      0    diet[Cheese]
     prot_con (S)
       D 2           4    8.925676E+00      0    diet[Potato]
     carb_con (S)
       D 2           5    9.174413E+00      0
       D 2           6    9.174413E+00      0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.05 seconds.
```

Next, restore the original constraint on calories and consider the case that you need a diet that supplies no more than 550 mg of sodium per day. The following row is appended to Table 6.12.

	Bread	Milk	Cheese	Potato	Fish	Yogurt
sodium, mg	148	122	337	186	56	132

You can change the parameters, add the new constraint, and solve the modified problem by submitting the following code:

```
cal_con.lb=300;
num sod{FOOD}=[148 122 337 186 56 132];
con sodium: sum{i in FOOD}sod[i]*diet[i] <= 550;
solve with lp/presolver=none
            basis=warmstart
            logfreq=1;
print diet;
```

The solution is shown in Output 6.2.5.

Output 6.2.5 Optimal Solution to the Diet Problem with Additional Constraint

```
                    The OPTMODEL Procedure

                      Problem Summary

     Objective Sense              Minimization
     Objective Function                      f
     Objective Type                     Linear

     Number of Variables                     6
     Bounded Above                           0
     Bounded Below                           5
     Bounded Below and Above                 1
     Free                                    0
     Fixed                                   0

     Number of Constraints                   5
     Linear LE (<=)                          2
     Linear EQ (=)                           0
     Linear GE (>=)                          3
     Linear Range                            0

     Constraint Coefficients                29


                   Performance Information

       Execution Mode        Single-Machine
       Number of Threads                    1


                      Solution Summary

       Solver                              LP
       Algorithm                 Dual Simplex
       Objective Function                   f
       Solution Status                Optimal
       Objective Value          12.081337881

       Primal Infeasibility                 0
       Dual Infeasibility                   0
       Bound Infeasibility                  0

       Iterations                           7
       Presolve Time                     0.00
       Solution Time                     0.03


                  [1]             diet

               Bread       0.000000
               Cheese      0.449499
               Fish        0.500000
               Milk        0.053599
               Potato      1.865168
               Yogurt      0.000000
```

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 6.2.6 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 linear constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: No basis information is available.  The BASIS=WARMSTART option is ignored.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective               Entering         Leaving
      Phase Iteration           Value         Time    Variable         Variable
        D 1            1    0.000000E+00          0
        D 2            2    5.500000E+00          0    diet[Milk]
      fat_con (S)
        D 2            3    8.650000E+00          0    diet[Yogurt]
      cal_con (S)
        D 2            4    8.894231E+00          0    diet[Potato]
      prot_con (S)
        D 2            5    1.155221E+01          0    diet[Cheese]
      diet[Yogurt]
        D 2            6    1.208134E+01          0
        D 2            7    1.208134E+01          0
NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.06 seconds.
```

Example 6.3: Two-Person Zero-Sum Game

Consider a two-person zero-sum game (where one person wins what the other person loses). The players make moves simultaneously, and each has a choice of actions. There is a *payoff* matrix that indicates the amount one player gives to the other under each combination of actions:

$$
\begin{array}{c}
\text{Player II plays } j \\
\begin{array}{cccc}
1 & 2 & 3 & 4
\end{array}
\end{array}
$$

$$
\text{Player I plays } i \quad
\begin{array}{c}
1 \\
2 \\
3
\end{array}
\begin{pmatrix}
-5 & 3 & 1 & 8 \\
5 & 5 & 4 & 6 \\
-4 & 6 & 0 & 5
\end{pmatrix}
$$

If player I makes move i and player II makes move j, then player I wins (and player II loses) a_{ij}. What is the best strategy for the two players to adopt? This example is simple enough to be analyzed from observation. Suppose player I plays 1 or 3; the best response of player II is to play 1. In both cases, player I loses and player II wins. So the best action for player I is to play 2. In this case, the best response for player II is to play 3, which minimizes the loss. In this case, (2, 3) is a *pure-strategy Nash equilibrium* in this game.

For illustration, consider the following mixed strategy case. Assume that player I selects i with probability p_i, $i = 1, 2, 3$, and player II selects j with probability q_j, $j = 1, 2, 3, 4$. Consider player II's problem of minimizing the maximum expected payout:

$$\min_{q} \left\{ \max_{i} \sum_{j=1}^{4} a_{ij} q_j \right\} \quad \text{subject to} \quad \sum_{j=1}^{4} q_{ij} = 1, \quad q \geq 0$$

This is equivalent to

$$\min_{q,v} v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} q_j \leq v \quad \forall i$$
$$\sum_{j=1}^{4} q_j = 1$$
$$q \geq 0$$

The problem can be transformed into a more standard format by making a simple change of variables: $x_j = q_j/v$. The preceding LP formulation now becomes

$$\min_{x,v} v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} x_j \leq 1 \quad \forall i$$
$$\sum_{j=1}^{4} x_j = 1/v$$
$$q \geq 0$$

which is equivalent to

$$\max_{x} \sum_{j=1}^{4} x_j \quad \text{subject to} \quad Ax \leq 1, \quad x \geq 0$$

where A is the payoff matrix and $\mathbf{1}$ is a vector of 1's. It turns out that the corresponding optimization problem from player I's perspective can be obtained by solving the dual problem, which can be written as

$$\min_{y} \sum_{i=1}^{3} y_i \quad \text{subject to} \quad A^T y \geq 1, \quad y \geq 0$$

You can model the problem and solve it by using PROC OPTMODEL as follows:

```
proc optmodel;
   num a{1..3, 1..4}=[-5 3 1 8
                       5 5 4 6
                      -4 6 0 5];
   var x{1..4} >= 0;
   max f = sum{i in 1..4}x[i];
   con c{i in 1..3}: sum{j in 1..4}a[i,j]*x[j] <= 1;
   solve with lp / algorithm = ps presolver = none logfreq = 1;
   print x;
   print c.dual;
quit;
```

The optimal solution is displayed in .

Output 6.3.1 Optimal Solutions to the Two-Person Zero-Sum Game

```
                    The OPTMODEL Procedure

                      Problem Summary

        Objective Sense            Maximization
        Objective Function                    f
        Objective Type                   Linear

        Number of Variables                   4
        Bounded Above                         0
        Bounded Below                         4
        Bounded Below and Above               0
        Free                                  0
        Fixed                                 0

        Number of Constraints                 3
        Linear LE (<=)                        3
        Linear EQ (=)                         0
        Linear GE (>=)                        0
        Linear Range                          0

        Constraint Coefficients              11


                   Performance Information

         Execution Mode        Single-Machine
         Number of Threads                  1


                     Solution Summary

        Solver                               LP
        Algorithm                 Primal Simplex
        Objective Function                    f
        Solution Status                 Optimal
        Objective Value                    0.25

        Primal Infeasibility                  0
        Dual Infeasibility                    0
        Bound Infeasibility                   0

        Iterations                            4
        Presolve Time                      0.00
        Solution Time                      0.05


                   [1]         x

                    1        0.00
                    2        0.00
                    3        0.25
                    4        0.00
```

Output 6.3.1 *continued*

```
                           [1]      c.DUAL

                            1        0.00
                            2        0.25
                            3        0.00
```

The optimal solution $\mathbf{x}^* = (0, 0, 0.25, 0)$ with an optimal value of 0.25. Therefore the optimal strategy for player II is $\mathbf{q}^* = \mathbf{x}^*/0.25 = (0, 0, 1, 0)$. You can check the optimal solution of the dual problem by using the constraint suffix ".dual". So $\mathbf{y}^* = (0, 0.25, 0)$ and player I's optimal strategy is $(0, 1, 0)$. The solution is consistent with our intuition from observation.

Example 6.4: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

$$
\begin{array}{llccccccll}
\min & x_1 & + & x_2 & + & x_3 & & & & \text{(cost)} \\
\text{subject to} & x_1 & + & x_2 & & & \geq & 10 & & \text{(con1)} \\
& x_1 & & & + & x_3 & \leq & 4 & & \text{(con2)} \\
4 \leq & & & x_2 & + & x_3 & \leq & 5 & & \text{(con3)} \\
& & & x_1, & x_2 & & \geq & 0 & & \\
& & 0 & \leq & x_3 & & \leq & 3 & &
\end{array}
$$

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem:

$$
\begin{array}{ccccccll}
x_1 & + & x_2 & & & \geq & 10 & \text{(con1)} \\
x_1 & & & + & x_3 & \leq & 4 & \text{(con2)} \\
& & x_2 & + & x_3 & \leq & 5 & \text{(con3)} \\
& & & & x_3 & \geq & 0 &
\end{array}
$$

You can formulate the problem and call the LP solver by using the following statements:

```
proc optmodel presolver=none;
   /* declare variables */
   var x{1..3} >=0;

   /* upper bound on variable x[3] */
   x[3].ub = 3;

   /* objective function */
   min obj = x[1] + x[2] + x[3];
```

```
/* constraints */
con c1: x[1] + x[2] >= 10;
con c2: x[1] + x[3] <= 4;
con c3: 4 <= x[2] + x[3] <= 5;

solve with lp / iis = on;

print x.status;
print c1.status c2.status c3.status;
```

The notes printed in the log appear in Output 6.4.1.

Output 6.4.1 Finding an IIS: Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The IIS option is enabled.
                          Objective
      Phase Iteration        Value            Time
        P 1            1    1.400000E+01          0
        P 1            4    1.000000E+00          0
NOTE: The IIS option found the problem to be infeasible.
NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.
      Processed      Removed       Time
          0             0           0
          1             0           0
          2             0           0
          3             0           0
NOTE: Processing variable bounds.
      Processed      Removed       Time
          0             0           0
          1             0           0
          2             0           0
          3             0           0
NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS option found the problem to be infeasible.
NOTE: The IIS option found an irreducible infeasible set with 1 variables and 3
      constraints.
NOTE: The IIS solve time is 0.02 seconds.
```

The output of the PRINT statements appears in Output 6.4.2. The value of the .status suffix for the variables x[1] and x[2] is "I," which indicates an infeasible problem. The value I is not one of those assigned by the IIS= option to members of the IIS, however, so the variable bounds for x[1] and x[2] are not in the IIS.

Output 6.4.2 Solution Summary, Variable Status, and Constraint Status

```
                    The OPTMODEL Procedure

                       Solution Summary

          Solver                                LP
          Algorithm               Primal Simplex
          Objective Function                    obj
          Solution Status            Infeasible

          Iterations                            14
          Presolve Time                       0.00
          Solution Time                       0.05


                    [1]     x.STATUS

                     1
                     2
                     3       I_L


          c1.STATUS      c2.STATUS      c3.STATUS

          I_L            I_U            I_U
```

The value of c3.status is I_U, which indicates that $x_2 + x_3 \leq 5$ is an element of the IIS. The original constraint is c3, a range constraint with a lower bound of 4. If you choose to remove the constraint $x_2 + x_3 \leq 5$, you can change the value of c3.ub to the largest positive number representable in your operating environment. You can specify this number by using the MIN aggregation expression in the OPTMODEL procedure. See "MIN Aggregation Expression" on page 106 for details.

The modified LP problem is specified and solved by adding the following lines to the original PROC OPTMODEL call.

```
/* relax upper bound on constraint c3 */
c3.ub = min{{}}0;

solve with lp / iis = on;

/* print solution */
print x;
```

Because one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets.

The notes shown in Output 6.4.3 are printed to the log.

Output 6.4.3 Infeasibility Removed: Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The IIS option is enabled.
                          Objective
      Phase Iteration       Value         Time
       P 1           1     1.400000E+01       0
       P 1           3     0.000000E+00       0
NOTE: The IIS option found the problem to be feasible.
NOTE: The IIS solve time is 0.05 seconds.
```

The solution summary and primal solution are displayed in Output 6.4.4.

Output 6.4.4 Infeasibility Removed: Solution

```
                    The OPTMODEL Procedure

                       Solution Summary

            Solver                          LP
            Algorithm           Primal Simplex
            Objective Function             obj
            Solution Status           Feasible

            Iterations                       3
            Presolve Time                 0.00
            Solution Time                 0.03


                       [1]     x

                        1      0
                        2      0
                        3      0
```

Example 6.5: Using the Network Simplex Solver

This example demonstrates how you can use the network simplex solver to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 6.4, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 6.4 Minimum Cost Network Flow Problem: Data
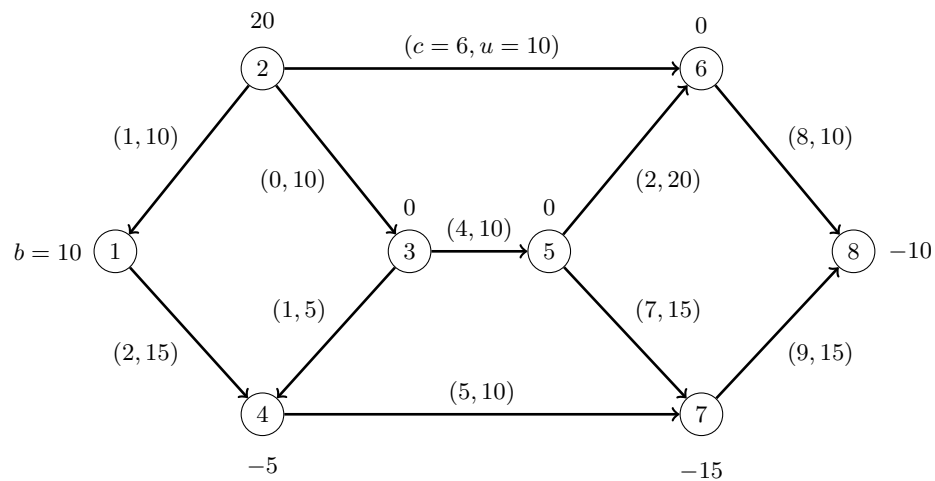
You can use the following SAS statements to create the input data sets nodedata and arcdata:

```
data nodedata;
   input _node_ $ _sd_;
   datalines;
1    10
2    20
3     0
4    -5
5     0
6     0
7   -15
8   -10
;

data arcdata;
   input _tail_ $ _head_ $ _lo_ _capac_ _cost_;
   datalines;
1    4    0    15    2
2    1    0    10    1
2    3    0    10    0
2    6    0    10    6
3    4    0     5    1
3    5    0    10    4
4    7    0    10    5
5    6    0    20    2
5    7    0    15    7
6    8    0    10    8
7    8    0    15    9
;
```

You can use the following call to PROC OPTMODEL to find the minimum-cost flow:

```
proc optmodel;
   set <str> NODES;
   num supply_demand {NODES};

   set <str,str> ARCS;
   num arcLower   {ARCS};
   num arcUpper   {ARCS};
   num arcCost    {ARCS};

   read data arcdata into ARCS=[_tail_ _head_]
      arcLower=_lo_ arcUpper=_capac_ arcCost=_cost_;
   read data nodedata into NODES=[_node_] supply_demand=_sd_;

   var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
   min obj = sum {<i,j> in ARCS} arcCost[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]
         = supply_demand[i];
   solve with lp / algorithm=ns scale=none logfreq=1;
   print flow;
quit;
%put &_OROPTMODEL_;
```

The optimal solution is displayed in Output 6.5.1.

Output 6.5.1 Network Simplex Solver: Primal Solution Output

```
                    The OPTMODEL Procedure

                        Problem Summary

         Objective Sense            Minimization
         Objective Function                  obj
         Objective Type                   Linear

         Number of Variables                  11
         Bounded Above                         0
         Bounded Below                         0
         Bounded Below and Above              11
         Free                                  0
         Fixed                                 0

         Number of Constraints                 8
         Linear LE  (<=)                        0
         Linear EQ  (=)                         8
         Linear GE  (>=)                        0
         Linear Range                          0

         Constraint Coefficients              22
```

Output 6.5.1 *continued*

```
                    Performance Information

        Execution Mode          Single-Machine
        Number of Threads                    1


                    Solution Summary

        Solver                             LP
        Algorithm           Network Simplex
        Objective Function              obj
        Solution Status             Optimal
        Objective Value                 270

        Primal Infeasibility              0
        Dual Infeasibility                0
        Bound Infeasibility               0

        Iterations                        8
        Iterations2                       0
        Presolve Time                  0.00
        Solution Time                  0.00


                [1]     [2]     flow

                 1       4       10
                 2       1        0
                 2       3       10
                 2       6       10
                 3       4        5
                 3       5        5
                 4       7       10
                 5       6        0
                 5       7        5
                 6       8       10
                 7       8        0
```

The optimal solution is represented graphically in Figure 6.5.

Figure 6.5 Minimum Cost Network Flow Problem: Optimal Solution
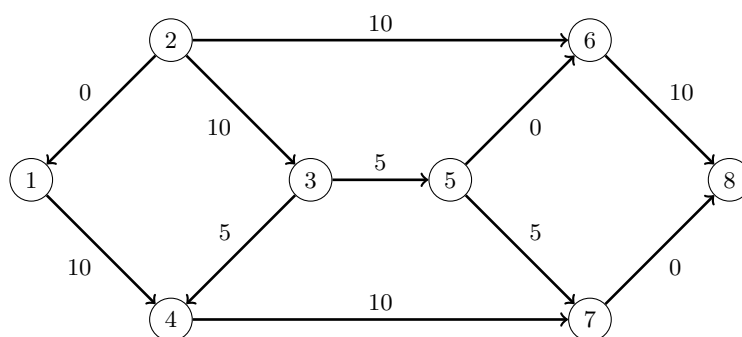
The iteration log is displayed in Output 6.5.2.

Output 6.5.2 Log: Solution Progress

```
NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 22 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem is a pure network instance; PRESOLVER=NONE is used.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 8 rows (100.00%), 11 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                          Primal          Primal            Dual
       Iteration         Objective     Infeasibility    Infeasibility      Time
               1                 0      20.0000000       89.0000000        0.00
               2                 0      20.0000000       89.0000000        0.00
               3         5.0000000      15.0000000       84.0000000        0.00
               4         5.0000000      15.0000000       83.0000000        0.00
               5        75.0000000      15.0000000       83.0000000        0.00
               6        75.0000000      15.0000000       79.0000000        0.00
               7       130.0000000      10.0000000       76.0000000        0.00
               8       270.0000000               0                0        0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=270
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=8
ITERATIONS2=0 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00
```

Now, suppose there is a budget on the flow that comes out of arc 2: the total arc cost of flow that comes out of arc 2 cannot exceed 50. You can use the following call to PROC OPTMODEL to find the minimum-cost flow:

```
proc optmodel;
   set <str> NODES;
   num supply_demand {NODES};

   set <str,str> ARCS;
   num arcLower  {ARCS};
   num arcUpper  {ARCS};
   num arcCost   {ARCS};

   read data arcdata into ARCS=[_tail_ _head_]
      arcLower=_lo_ arcUpper=_capac_ arcCost=_cost_;
   read data nodedata into NODES=[_node_] supply_demand=_sd_;

   var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
```

```
      min obj = sum {<i,j> in ARCS} arcCost[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]
            = supply_demand[i];
      con budgetOn2:
         sum {<i,j> in ARCS: i='2'} arcCost[i,j] * flow[i,j] <= 50;
      solve with lp / algorithm=ns scale=none logfreq=1;
      print flow;
   quit;
   %put &_OROPTMODEL_;
```

The optimal solution is displayed in Output 6.5.3.

Output 6.5.3 Network Simplex Solver: Primal Solution Output

```
                        The OPTMODEL Procedure

                           Problem Summary

              Objective Sense              Minimization
              Objective Function                     obj
              Objective Type                      Linear

              Number of Variables                    11
              Bounded Above                           0
              Bounded Below                           0
              Bounded Below and Above                11
              Free                                    0
              Fixed                                   0

              Number of Constraints                   9
              Linear LE (<=)                          1
              Linear EQ (=)                           8
              Linear GE (>=)                          0
              Linear Range                            0

              Constraint Coefficients                24


                        Performance Information

                Execution Mode        Single-Machine
                Number of Threads                  1
```

Output 6.5.3 *continued*

```
                    Solution Summary

         Solver                              LP
         Algorithm               Network Simplex
         Objective Function                  obj
         Solution Status                 Optimal
         Objective Value                     274

         Primal Infeasibility                  0
         Dual Infeasibility                    0
         Bound Infeasibility                   0

         Iterations                            7
         Iterations2                           3
         Presolve Time                      0.02
         Solution Time                      0.05


                 [1]     [2]     flow

                  1       4       12
                  2       1        2
                  2       3       10
                  2       6        8
                  3       4        3
                  3       5        7
                  4       7       10
                  5       6        2
                  5       7        5
                  6       8       10
                  7       8        0
```
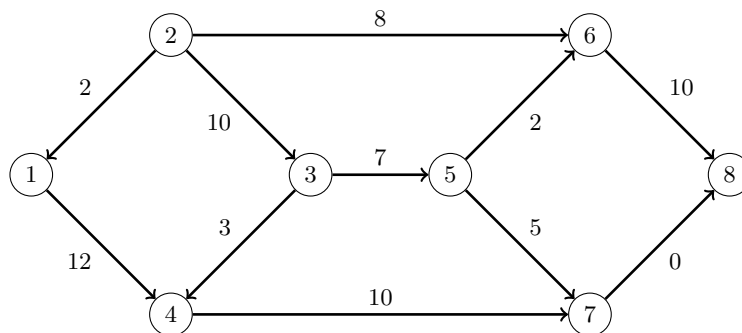
The optimal solution is represented graphically in Figure 6.6.

Figure 6.6 Minimum Cost Network Flow Problem: Optimal Solution (with Budget Constraint)

The iteration log is displayed in Output 6.5.4. Note that the network simplex solver extracts a subnetwork in this case.

Output 6.5.4 Log: Solution Progress

```
NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (1 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 LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 4 variables and 4 constraints.
NOTE: The LP presolver removed 7 constraint coefficients.
NOTE: The presolved problem has 7 variables, 5 constraints, and 17 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 4 rows (80.00%), 7 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                        Primal          Primal           Dual
        Iteration      Objective    Infeasibility   Infeasibility      Time
                1      80.1500000      10.0600000      55.0000000       0.00
                2     105.3000000       5.0300000      54.0000000       0.00
                3     105.3000000       5.0300000      54.0000000       0.00
                4     135.3000000       0.0300000      49.0000000       0.00
                5     135.6300000               0      47.0000000       0.00
                6     135.6300000               0               0       0.00
                7     270.0000000               0               0       0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
NOTE: The Dual Simplex algorithm is used.
                             Objective               Entering          Leaving
        Phase Iteration        Value         Time     Variable         Variable
         D 2          1      2.700000E+02        0   flow['5','6']
        budgetOn2 (S)
         D 2          2      2.740000E+02        0
         D 2          3      2.740000E+02        0
NOTE: Optimal.
NOTE: Objective = 274.
NOTE: The Simplex solve time is 0.03 seconds.
STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=274
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=7
ITERATIONS2=3 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.03
```

Example 6.6: Migration to OPTMODEL: Generalized Networks

The following example shows how to use PROC OPTMODEL to solve the example "Generalized Networks: Using the EXCESS= Option" in Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data sets are the same as in the PROC NETFLOW example.

```
title 'Generalized Networks';

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
;
```

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 GENETOUT:

```
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 = min {r in {}} r;
      /* 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_[i,j]" in the second sum. The balance constraint is initially declared as an equality, but to mimic the EXCESS=SUPPLY option in PROC NETFLOW, the sense of this constraint is changed to "≤" by relaxing the constraint's lower bound for supply nodes. The output data set is displayed in Output 6.6.1.

Output 6.6.1 Optimal Solution with Excess Supply

Obs	_from_	_to_	_cost_	_capac_	_lo_	_mult_	_supply_	_demand_	_flow_	_fcost_
1	s1	d1	1	.	0	1.0	5	5	5	5
2	s1	d2	8	.	0	1.0	5	10	0	0
3	s2	d1	4	.	0	2.0	20	5	0	0
4	s2	d2	2	.	0	2.0	20	10	5	10
5	s2	d3	1	.	0	2.0	20	20	10	10
6	s3	d2	5	.	0	0.5	10	10	0	0
7	s3	d3	4	.	0	0.5	10	20	0	0

The log is displayed in Output 6.6.2.

Output 6.6.2 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.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           4    2.500000E+01        0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.05 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 the PROC NETFLOW example:

```
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 example, except for the balance constraint. The balance constraint 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 = min {r in {}} r;
    /* 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 is displayed in Output 6.6.3.

Output 6.6.3 Optimal Solution with Excess Demand

Obs	_from_	_to_	_cost_	_capac_	_lo_	_mult_	_supply_	_demand_	_flow_	_fcost_
1	s1	d1	1	.	0	0.5	5	5	5	5
2	s1	d2	8	.	0	0.5	5	10	0	0
3	s2	d1	4	.	0	1.0	20	5	0	0
4	s2	d2	2	.	0	1.0	20	10	5	10
5	s2	d3	1	.	0	1.0	20	20	15	15
6	s3	d2	5	.	0	0.5	10	10	0	0
7	s3	d3	4	.	0	0.5	10	20	10	40

The log is displayed in Output 6.6.4.

Output 6.6.4 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            5   7.000000E+01        0
NOTE: Optimal.
NOTE: Objective = 70.
NOTE: The Dual Simplex solve time is 0.05 seconds.
NOTE: The data set WORK.GNETOUT1 has 7 observations and 10 variables.
```

Example 6.7: Migration to OPTMODEL: Maximum Flow

The following example shows how to use PROC OPTMODEL to solve the example "Maximum Flow Problem" in Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data set is the same as in that example.

```
title 'Maximum Flow Problem';

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 . .
  ;
```

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 is displayed in Output 6.7.1.

Output 6.7.1 Optimal Solution

Obs	_from_	_to_	_cost_	_capac_	_lo_	_supply_	_demand_	_flow_	_fcost_
1	S	a	0	.	0	.	.	12	0
2	S	b	0	.	0	.	.	13	0
3	a	c	1	7	0	.	.	7	7
4	b	c	2	9	0	.	.	8	16
5	a	d	3	5	0	.	.	5	15
6	b	d	4	8	0	.	.	5	20
7	c	e	5	15	0	.	.	15	75
8	d	f	6	20	0	.	.	10	60
9	e	g	7	11	0	.	.	3	21
10	f	g	8	6	0	.	.	6	48
11	e	h	9	12	0	.	.	12	108
12	f	h	10	4	0	.	.	4	40
13	g	T	0	.	0	.	.	9	0
14	h	T	0	.	0	.	.	16	0

The log is displayed in Output 6.7.2.

Output 6.7.2 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
        P 2         6   2.500000E+01      0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.03 seconds.
NOTE: The data set WORK.GOUT3 has 14 observations and 9 variables.
```

Example 6.8: Migration to OPTMODEL: Production, Inventory, Distribution

The following example shows how to use PROC OPTMODEL to solve the example "Production, Inventory, Distribution Problem" in Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data sets are the same as in that example.

```
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   .
;
```

The following PROC OPTMODEL statements 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 = min {r in {}} r;
   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 PROC OPTMODEL 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 PROC 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 the NETFLOW example, including reduced cost, basis status, and dual values. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to the PROC NETFLOW example:

```
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 are displayed in Output 6.8.1.

Output 6.8.1 Output Data Sets

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_
1	fact1_1	f1_mar_1	127.90	500	50	prod f1 19 mar
2	fact1_1	f1_apr_1	78.60	600	50	prod f1 19 apl
3	fact1_1	f1_may_1	95.10	400	50	
4	f1_mar_1	f1_apr_1	15.00	50	0	
5	f1_apr_1	f1_may_1	12.00	50	0	
6	f1_apr_1	f1_mar_1	28.00	20	0	back f1 19 apl
7	f1_may_1	f1_apr_1	28.00	20	0	back f1 19 may
8	f1_mar_1	f2_mar_1	11.00	.	0	
9	f1_apr_1	f2_apr_1	11.00	.	0	
10	f1_may_1	f2_may_1	16.00	.	0	
11	f1_mar_1	shop1_1	−327.65	250	0	
12	f1_apr_1	shop1_1	−300.00	250	0	
13	f1_may_1	shop1_1	−285.00	250	0	
14	f1_mar_1	shop2_1	−362.74	250	0	
15	f1_apr_1	shop2_1	−300.00	250	0	
16	f1_may_1	shop2_1	−245.00	250	0	
17	fact2_1	f2_mar_1	88.00	450	35	prod f2 19 mar
18	fact2_1	f2_apr_1	62.40	480	35	prod f2 19 apl
19	fact2_1	f2_may_1	133.80	250	35	
20	f2_mar_1	f2_apr_1	18.00	30	0	
21	f2_apr_1	f2_may_1	20.00	30	0	
22	f2_apr_1	f2_mar_1	17.00	15	0	back f2 19 apl
23	f2_may_1	f2_apr_1	25.00	15	0	back f2 19 may
24	f2_mar_1	f1_mar_1	10.00	40	0	

Obs	_supply_	_demand_	_flow_	_fcost_
1	1000	.	345	44125.50
2	1000	.	600	47160.00
3	1000	.	50	4755.00
4	.	.	0	0.00
5	.	.	50	600.00
6	.	.	20	560.00
7	.	.	0	0.00
8	.	.	0	0.00
9	.	.	30	330.00
10	.	.	100	1600.00
11	.	900	155	−50785.75
12	.	900	250	−75000.00
13	.	900	0	0.00
14	.	900	250	−90685.00
15	.	900	250	−75000.00
16	.	900	0	0.00
17	850	.	290	25520.00
18	850	.	480	29952.00
19	850	.	35	4683.00
20	.	.	0	0.00
21	.	.	15	300.00
22	.	.	0	0.00
23	.	.	0	0.00
24	.	.	40	400.00

Output 6.8.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_
25	f2_apr_1	f1_apr_1	11.00	40	0	
26	f2_may_1	f1_may_1	13.00	40	0	
27	f2_mar_1	shop1_1	−297.40	250	0	
28	f2_apr_1	shop1_1	−290.00	250	0	
29	f2_may_1	shop1_1	−292.00	250	0	
30	f2_mar_1	shop2_1	−272.70	250	0	
31	f2_apr_1	shop2_1	−312.00	250	0	
32	f2_may_1	shop2_1	−299.00	250	0	
33	fact1_2	f1_mar_2	217.90	400	40	prod f1 25 mar
34	fact1_2	f1_apr_2	174.50	550	50	prod f1 25 apl
35	fact1_2	f1_may_2	133.30	350	40	
36	f1_mar_2	f1_apr_2	20.00	40	0	
37	f1_apr_2	f1_may_2	18.00	40	0	
38	f1_apr_2	f1_mar_2	32.00	30	0	back f1 25 apl
39	f1_may_2	f1_apr_2	41.00	15	0	back f1 25 may
40	f1_mar_2	f2_mar_2	23.00	.	0	
41	f1_apr_2	f2_apr_2	23.00	.	0	
42	f1_may_2	f2_may_2	26.00	.	0	
43	f1_mar_2	shop1_2	−559.76	.	0	
44	f1_apr_2	shop1_2	−524.28	.	0	
45	f1_may_2	shop1_2	−475.02	.	0	
46	f1_mar_2	shop2_2	−623.89	.	0	
47	f1_apr_2	shop2_2	−549.68	.	0	
48	f1_may_2	shop2_2	−460.00	.	0	

Obs	_supply_	_demand_	_flow_	_fcost_
25	.	.	0	0.00
26	.	.	0	0.00
27	.	900	250	−74350.00
28	.	900	245	−71050.00
29	.	900	0	0.00
30	.	900	0	0.00
31	.	900	250	−78000.00
32	.	900	150	−44850.00
33	1000	.	400	87160.00
34	1000	.	550	95975.00
35	1000	.	40	5332.00
36	.	.	0	0.00
37	.	.	0	0.00
38	.	.	30	960.00
39	.	.	15	615.00
40	.	.	0	0.00
41	.	.	0	0.00
42	.	.	0	0.00
43	.	900	0	0.00
44	.	900	0	0.00
45	.	900	25	−11875.50
46	.	1450	455	−283869.95
47	.	1450	535	−294078.80
48	.	1450	0	0.00

Output 6.8.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_
49	fact2_2	f2_mar_2	182.00	650	35	prod f2 25 mar
50	fact2_2	f2_apr_2	196.70	680	35	prod f2 25 apl
51	fact2_2	f2_may_2	201.40	550	35	
52	f2_mar_2	f2_apr_2	28.00	50	0	
53	f2_apr_2	f2_may_2	38.00	50	0	
54	f2_apr_2	f2_mar_2	31.00	15	0	back f2 25 apl
55	f2_may_2	f2_apr_2	54.00	15	0	back f2 25 may
56	f2_mar_2	f1_mar_2	20.00	25	0	
57	f2_apr_2	f1_apr_2	21.00	25	0	
58	f2_may_2	f1_may_2	43.00	25	0	
59	f2_mar_2	shop1_2	−567.83	500	0	
60	f2_apr_2	shop1_2	−542.19	500	0	
61	f2_may_2	shop1_2	−461.56	500	0	
62	f2_mar_2	shop2_2	−542.83	500	0	
63	f2_apr_2	shop2_2	−559.19	500	0	
64	f2_may_2	shop2_2	−489.06	500	0	

Obs	_supply_	_demand_	_flow_	_fcost_
49	1500	.	645	117390.00
50	1500	.	680	133756.00
51	1500	.	35	7049.00
52	.	.	0	0.00
53	.	.	0	0.00
54	.	.	0	0.00
55	.	.	15	810.00
56	.	.	25	500.00
57	.	.	0	0.00
58	.	.	0	0.00
59	.	900	500	−283915.00
60	.	900	375	−203321.25
61	.	900	0	0.00
62	.	1450	120	−65139.60
63	.	1450	320	−178940.80
64	.	1450	20	−9781.20
				===========
				−1281110.35

Output 6.8.1 *continued*

Obs	_rcost_	_status_	diagonal	factory	key_id	mth_made
1	.	B	19	1	production	March
2	−0.65	U	19	1	production	April
3	0.85	L	19	1	production	May
4	63.65	L	19	1	storage	March
5	−3.00	U	19	1	storage	April
6	−20.65	U	19	1	backorder	April
7	43.00	L	19	1	backorder	May
8	50.90	L	19	.	f1_to_2	March
9	.	B	19	.	f1_to_2	April
10	.	B	19	.	f1_to_2	May
11	.	B	19	1	sales	March
12	−21.00	U	19	1	sales	April
13	9.00	L	19	1	sales	May
14	−46.09	U	19	1	sales	March
15	−32.00	U	19	1	sales	April
16	38.00	L	19	1	sales	May
17	.	B	19	2	production	March
18	−27.85	U	19	2	production	April
19	23.55	L	19	2	production	May
20	15.75	L	19	2	storage	March
21	.	B	19	2	storage	April
22	19.25	L	19	2	backorder	April
23	45.00	L	19	2	backorder	May
24	−29.90	U	19	.	f2_to_1	March
25	22.00	L	19	.	f2_to_1	April
26	29.00	L	19	.	f2_to_1	May
27	−9.65	U	19	2	sales	March
28	.	B	19	2	sales	April
29	18.00	L	19	2	sales	May
30	4.05	L	19	2	sales	March
31	−33.00	U	19	2	sales	April
32	.	B	19	2	sales	May
33	−45.16	U	25	1	production	March
34	−14.35	U	25	1	production	April
35	2.11	L	25	1	production	May
36	94.21	L	25	1	storage	March
37	75.66	L	25	1	storage	April
38	−42.21	U	25	1	backorder	April
39	−16.66	U	25	1	backorder	May
40	104.06	L	25	.	f1_to_2	March
41	13.49	L	25	.	f1_to_2	April
42	28.96	L	25	.	f1_to_2	May
43	47.13	L	25	1	sales	March
44	8.40	L	25	1	sales	April
45	.	B	25	1	sales	May
46	.	B	25	1	sales	March
47	.	B	25	1	sales	April
48	32.02	L	25	1	sales	May
49	.	B	25	2	production	March
50	−1.66	U	25	2	production	April
51	73.17	L	25	2	production	May
52	11.64	L	25	2	storage	March

Output 6.8.1 *continued*

Obs	_rcost_	_status_	diagonal	factory	key_id	mth_made
53	108.13	L	25	2	storage	April
54	47.36	L	25	2	backorder	April
55	−16.13	U	25	2	backorder	May
56	−61.06	U	25	.	f2_to_1	March
57	30.51	L	25	.	f2_to_1	April
58	40.04	L	25	.	f2_to_1	May
59	−42.00	U	25	2	sales	March
60	.	B	25	2	sales	April
61	10.50	L	25	2	sales	May
62	.	B	25	2	sales	March
63	.	B	25	2	sales	April
64	.	B	25	2	sales	May

Obs	_node_	_supdem_	_dual_
1	fact1_1	1000	0.00
2	fact2_1	850	0.00
3	fact1_2	1000	0.00
4	fact2_2	1500	0.00
5	shop1_1	−900	199.75
6	shop2_1	−900	188.75
7	shop1_2	−900	343.83
8	shop2_2	−1450	360.83
9	f1_mar_1	.	−127.90
10	f1_apr_1	.	−79.25
11	f1_may_1	.	−94.25
12	f2_mar_1	.	−88.00
13	f2_apr_1	.	−90.25
14	f2_may_1	.	−110.25
15	f1_mar_2	.	−263.06
16	f1_apr_2	.	−188.85
17	f1_may_2	.	−131.19
18	f2_mar_2	.	−182.00
19	f2_apr_2	.	−198.36
20	f2_may_2	.	−128.23

The log is displayed in

Output 6.8.2 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          33  -1.281110E+06        0
NOTE: Optimal.
NOTE: Objective = -1281110.35.
NOTE: The Dual Simplex solve time is 0.05 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 6.9: Migration to OPTMODEL: Shortest Path

The following example shows how to use PROC OPTMODEL to solve the example "Shortest Path Problem" in Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data set is the same as in that example.

```
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
    ;
```

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 index 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 the PROC NETFLOW example, including reduced cost and basis status. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to the PROC NETFLOW example:

```
proc print data=spath;
    sum _fcost_;
run;
```

The output is displayed in .

Output 6.9.1 Output Data Set

Obs	from	to	_cost_	_capac_	_lo_	_supply_	_demand_	_flow_	_fcost_	_rcost_	_status_
1	Honolulu	Chicago	105	.	0	1	.	0	0	.	B
2	Honolulu	San Francisco	75	.	0	1	.	0	0	.	B
3	Honolulu	Los Angeles	68	.	0	1	.	1	68	.	B
4	Chicago	Boston	45	.	0	.	.	0	0	61	L
5	Chicago	New York	56	.	0	.	.	0	0	49	L
6	San Francisco	Boston	71	.	0	.	.	0	0	57	L
7	San Francisco	New York	48	.	0	.	.	0	0	11	L
8	San Francisco	Atlanta	63	.	0	.	.	0	0	37	L
9	Los Angeles	New York	44	.	0	.	.	1	44	.	B
10	Los Angeles	Atlanta	57	.	0	.	.	0	0	24	L
11	Boston	Heathrow London	88	.	0	.	1	0	0	.	B
12	New York	Heathrow London	65	.	0	.	1	1	65	.	B
13	Atlanta	Heathrow London	76	.	0	.	1	0	0	.	B
									===		
									177		

The log is displayed in Output 6.9.2.

Output 6.9.2 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.

Andersen, E. D. and Andersen, K. D. (1995), "Presolving in Linear Programming," *Mathematical Programming*, 71, 221–245.

Dantzig, G. B. (1963), *Linear Programming and Extensions*, Princeton, NJ: Princeton University Press.

Forrest, J. J. and Goldfarb, D. (1992), "Steepest-Edge Simplex Algorithms for Linear Programming," *Mathematical Programming*, 5, 1–28.

Gondzio, J. (1997), "Presolve Analysis of Linear Programs prior to Applying an Interior Point Method," *INFORMS Journal on Computing*, 9, 73–91.

Harris, P. M. J. (1973), "Pivot Selection Methods in the Devex LP Code," *Mathematical Programming*, 57, 341–374.

Maros, I. (2003), *Computational Techniques of the Simplex Method*, Boston: Kluwer Academic.

Chapter 7
The Mixed Integer Linear Programming Solver

Contents

Overview: MILP Solver

The OPTMODEL procedure provides a framework for specifying and solving mixed integer linear programs (MILPs). A standard mixed integer linear program has the formulation

$$
\begin{array}{rll}
\min & \mathbf{c}^T \mathbf{x} & \\
\text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} & \text{(MILP)} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} & \\
& \mathbf{x}_i \in \mathbb{Z} \quad \forall i \in \mathcal{S} &
\end{array}
$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of structural variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of technological coefficients
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables
$\mathcal{S}$			is a nonempty subset of the set $\{1 \ldots, n\}$ of indices

The MILP solver, available in the OPTMODEL procedure, implements an linear-programming-based branch-and-bound algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The MILP solver also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The MILP solver provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned. It is also possible to input an incumbent solution; see the section "Warm Start Option" on page 252 for details.

Getting Started: MILP Solver

The following example illustrates how you can use the OPTMODEL procedure to solve mixed integer linear programs. For more examples, see the section "Examples: MILP Solver" on page 270. Suppose you want to solve the following problem:

$$
\begin{array}{rrrrrrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & \\
\text{s.t.} & & - & 2x_2 & - & 3x_3 & \geq & -5 \quad \text{(R1)} \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 \quad \text{(R2)} \\
& x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 \quad \text{(R3)} \\
& x_1, & & x_2, & & x_3 & \geq & 0 \\
& x_1, & & x_2, & & x_3 & \in \mathbb{Z} &
\end{array}
$$

You can use the following statements to call the OPTMODEL procedure for solving mixed integer linear programs:

```
proc optmodel;
   var x{1..3} >= 0 integer;

   min f = 2*x[1] - 3*x[2] - 4*x[3];

   con r1: -2*x[2] - 3*x[3] >= -5;
   con r2: x[1] + x[2] + 2*x[3] <= 4;
   con r3: x[1] + 2*x[2] + 3*x[3] <= 7;

   solve with milp / presolver = automatic heuristics = automatic;
   print x;
quit;
```

The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value, AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal value of x is shown in Figure 7.1.

Figure 7.1 Solution Output

```
                The OPTMODEL Procedure

                      [1]    x

                       1     0
                       2     1
                       3     1
```

The solution summary stored in the macro variable _OROPTMODEL_ can be viewed by issuing the following statement:

```
%put &_OROPTMODEL_;
```

This statement produces the output shown in Figure 7.2.

Figure 7.2 Macro Output

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=1 ITERATIONS=5 PRESOLVE_TIME=0.00
SOLUTION_TIME=0.05
```

Syntax: MILP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH MILP < */ options* > **;**

Functional Summary

Table 7.1 summarizes the options available for the SOLVE WITH MILP statement, classified by function.

Table 7.1 Options for the MILP Solver

Description	Option
Presolve Option	
Specifies the type of presolve	PRESOLVER=

Table 7.1 (continued)

Description	Option
Warm Start Option	
Specifies the input primal solution (warm start)	PRIMALIN
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the node log	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Enables use of variable priorities	PRIORITY=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=

Table 7.1 (continued)

Description	Option
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=
Decomposition Algorithm Options	
Enables decomposition algorithm and specifies general control options	DECOMP=()
Specifies options for the master problem	DECOMP_MASTER=()
Specifies options for the master problem solved as a MILP	DECOMP_MASTER_IP=()
Specifies options for the subproblem	DECOMP_SUBPROB=()

MILP Solver Options

This section describes the options that are recognized by the MILP solver in PROC OPTMODEL. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the MILP solver is explicitly specified using a WITH clause. For example, the following line could appear in PROC OPTMODEL statements:

```
solve with milp / allcuts=aggressive maxnodes=10000 primalin;
```

Presolve Option

PRESOLVER=*number* | *string*
> specifies a presolve *string* or its corresponding value *number*, as listed in Table 7.2.

Table 7.2 Values for PRESOLVER= Option

number	string	Description
−1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default value is AUTOMATIC.

Warm Start Option

PRIMALIN

enables you to input a starting solution in PROC OPTMODEL before invoking the MILP solver. Adding the PRIMALIN option to the SOLVE statement requests that the MILP solver use the current variable values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. It is possible to set a variable value to the missing value '.' to mark a variable for repair. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

NOTE: If the MILP solver produces a feasible solution, the variable values from that run can be used as the warm start solution for a subsequent run. If the warm start solution is not feasible for the subsequent run, the solver automatically tries to repair it.

Control Options

ABSOBJGAP=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the objective of the best remaining node falls below the value of *number*, the solver stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=*number*

cuts off any nodes in a minimization (maximization) problem with an objective value above (below) *number*. The value of *number* can be any number; the default value is the positive (negative) number that has the largest absolute value representable in your operating environment.

EMPHASIS=*number* | *string*

specifies a search emphasis *string* or its corresponding value *number* as listed in Table 7.3.

Table 7.3 Values for EMPHASIS= Option

number	string	Description
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

The default value is BALANCE.

FEASTOL=*number*

specifies the tolerance used to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default value is 1E–6.

If the MILP solver fails to find a feasible solution within this tolerance but does find a solution with a slightly larger violation, then the solver ends with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMODEL_ " on page 267).

INTTOL=_number_

    specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 0.0 and 1.0; the default value is 1E–5. The MILP solver attempts to find an optimal solution with integer infeasibility less than *number*. If you assign a value smaller than 1E–10 to *number* and the best solution found by the solver has integer infeasibility between *number* and 1E–10, then the solver terminates with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMODEL_ " on page 267).

LOGFREQ=_number_

PRINTFREQ=_number_

    specifies how often information is printed in the node log. The value of *number* can be any nonnegative number up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=_number_ | *string*

PRINTLEVEL2=_number_ | *string*

    controls the amount of information displayed in the SAS log by the MILP solver, from a short description of presolve information and summary to details at each node. Table 7.4 describes the valid values for this option.

Table 7.4   Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all solver-related messages to SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval dictated by the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval dictated by the LOGFREQ= option

    The default value is MODERATE.

MAXNODES=_number_

    specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXSOLS=_number_

    specifies a stopping criterion. If *number* solutions have been found, then the solver stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXTIME=_t_

    specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed.

The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=number

specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default is 1E–6.

PROBE=number | string

specifies a probing *string* or its corresponding value *number*, as listed in the following table:

Table 7.5 Values for PROBE= Option

number	string	Description
−1	AUTOMATIC	Uses the probing strategy determined by the MILP solver
0	NONE	Disables probing
1	MODERATE	Uses probing moderately
2	AGGRESSIVE	Uses probing aggressively

The default value is AUTOMATIC.

RELOBJGAP=number

specifies a stopping criterion based on the best integer objective (BestInteger) and the objective of the best remaining node (BestBound). The relative objective gap is equal to

$$| \text{BestInteger} - \text{BestBound} | \, / \, (1\text{E}{-}10 + | \text{BestBound} |)$$

When this value becomes smaller than the specified gap size *number*, the solver stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

SCALE=option

indicates whether to scale the problem matrix. SCALE= can take either of the values AUTOMATIC (–1) and NONE (0). SCALE=AUTOMATIC scales the matrix as determined by the MILP solver; SCALE=NONE disables scaling. The default value is AUTOMATIC.

TARGET=number

specifies a stopping criterion for minimization (maximization) problems. If the best integer objective is better than or equal to *number*, the solver stops. The value of *number* can be any number; the default value is the negative (positive) number that has the largest absolute value representable in your operating environment.

TIMETYPE=string | number

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 7.6 describes the valid values of the TIMETYPE= option.

Table 7.6 Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The Optimization Statistics table, an output of PROC OPTMODEL if option PRINTLEVEL=2 is specified in the PROC OPTMODEL statement, also includes the same time units for "Presolver Time" and "Solver Time." The other times (such as "Problem Generation Time") in the Optimization Statistics table are always CPU times.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement of the OPTMODEL procedure. See the section "PERFOR-MANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and NODES= options.

If you specify a value greater than 1 for either the NTHREADS= or the NODES= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and NODES= options, the default value of the TIMETYPE= option is CPU.

Heuristics Option

HEURISTICS=*number* | *string*

controls the level of primal heuristics applied by the MILP solver. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. The values of *string* and the corresponding values of *number* are listed in Table 7.7.

Table 7.7   Values for HEURISTICS= Option

number	*string*	**Description**
−1	AUTOMATIC	Applies default level of heuristics, similar to MOD-ERATE
0	NONE	Disables all primal heuristics
1	BASIC	Applies basic primal heuristics at low frequency
2	MODERATE	Applies most primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all primal heuristics at high frequency

Setting HEURISTICS=NONE does not disable the heuristics that repair an infeasible input solution that is specified by using the PRIMALIN option.

The default value is AUTOMATIC. For details about primal heuristics, see the section "Primal Heuristics" on page 264.

Search Options

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search performed by the MILP solver. Conflict finds clauses resulting from infeasible subproblems that arise in the search tree. The values of *string* and the corresponding values of *number* are listed in Table 7.8.

Table 7.8   Values for CONFLICTSEARCH= Option

number	string	Description
−1	AUTOMATIC	Performs conflict search based on a strategy determined by the MILP solver
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

The default value is AUTOMATIC.

NODESEL=*number* | *string*

specifies the node selection strategy *string* or its corresponding value *number* as listed in Table 7.9.

Table 7.9   Values for NODESEL= Option

number	string	Description
−1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node with the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node with the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

The default value is AUTOMATIC. For details about node selection, see the section "Node Selection" on page 261.

PRIORITY=0 | **1**

indicates whether to use specified branching priorities for integer variables. PRIORITY=0 ignores variable priorities; PRIORITY=1 uses priorities when they exist. The default value is 1. See the section "Branching Priorities" on page 262 for details.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations performed for each variable in the candidate list when the strong branching variable selection strategy is used. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the MILP solver uses the default value; this value is calculated automatically.

STRONGLEN=*number* | **AUTOMATIC**

specifies the number of candidates used when the strong branching variable selection strategy is performed. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the MILP solver uses the default value; this value is calculated automatically.

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. The values of *string* and the corresponding values of *number* are listed in Table 7.10.

Table 7.10 Values for VARSEL= Option

number	string	Description
−1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable with maximum infeasibility
1	MININFEAS	Chooses the variable with minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses strong branching variable selection strategy

The default value is AUTOMATIC. For details about variable selection, see the section "Variable Selection" on page 262.

Cut Options

Table 7.11 describes the *string* and *number* values for the cut options in the OPTMODEL procedure.

Table 7.11 Values for Individual Cut Options

number	string	Description
−1	AUTOMATIC	Generates cutting planes based on a strategy determined by the MILP solver
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

You can specify the CUTSTRATEGY= option to set the overall aggressiveness of the cut generation in the MILP solver. Alternatively, you can use the ALLCUTS= option to set all cut types to the same level. You can override the ALLCUTS= value by using the options that correspond to particular cut types. For example, if you want the MILP solver to generate only Gomory cuts, specify ALLCUTS=NONE and CUTGOMORY=AUTOMATIC. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set ALLCUTS=AGGRESSIVE and CUTLAP=NONE.

ALLCUTS=_number | string_

provides a shorthand way of setting all the cuts-related options in one setting. In other words, ALL-CUTS=*number* is equivalent to setting each of the individual cuts parameters to the same value *number*. Thus, ALLCUTS=−1 has the effect of setting CUTCLIQUE=−1, CUTFLOWCOVER=−1, CUTFLOWPATH=−1, . . . , CUTMIR=−1, and CUTZEROHALF=−1. Table 7.11 lists the values that can be assigned to *option* and *number*. In addition, you can override levels for individual cuts with the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIM-PLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, and CUTZEROHALF= options. If the ALLCUTS= option is not specified, then all the cuts-related options are either at their individually specified values (if the corresponding option is specified) or at their default values (if that option is not specified).

CUTCLIQUE=_number | string_

specifies the level of clique cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTCLIQUE= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWCOVER=*number | string*

specifies the level of flow cover cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTFLOWCOVER= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWPATH=*number | string*

specifies the level of flow path cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTFLOWPATH= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGOMORY=*number | string*

specifies the level of Gomory cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTGOMORY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGUB=*number | string*

specifies the level of generalized upper bound (GUB) cover cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTGUB= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTIMPLIED=*number | string*

specifies the level of implied bound cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTIMPLIED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTKNAPSACK=*number | string*

specifies the level of knapsack cover cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTKNAPSACK= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTLAP=*number | string*

specifies the level of lift-and-project (LAP) cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTLAP= option overrides the ALLCUTS= option. The default value is NONE.

CUTMILIFTED=*number | string*

specifies the level of mixed lifted 0-1 cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTMILIFTED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMIR=*number | string*

specifies the level of mixed integer rounding (MIR) cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*. The CUTMIR= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTSFACTOR=*number*

specifies a row multiplier factor for cuts. The number of cuts that are added is limited to *number* times the original number of rows. The value of *number* can be any nonnegative number less than or equal to 100; the default value is automatically calculated by the MILP solver.

CUTSTRATEGY=*number | string*

CUTS=*number | string*

> specifies the overall aggressiveness of the cut generation in the solver. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is basic, moderate, or aggressive compared to the default value.

CUTZEROHALF=*number | string*

> specifies the level of zero-half cuts that are generated by the MILP solver. Table 7.11 lists the values that can be assigned to *option* and *number*.The CUTZEROHALF= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

Decomposition Algorithm Options

The following options are available for the decomposition algorithm in the MILP solver. For information about the decomposition algorithm, see Chapter 13, "The Decomposition Algorithm."

DECOMP=(*options***)**

> enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

DECOMP_MASTER=(*options***)**

> specifies options for the master problem. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

DECOMP_MASTER_IP=(*options***)**

> specifies options for the (restricted) master problem solved as a MILP with the current set of columns in an effort to obtain an integer feasible solution. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

DECOMP_SUBPROB=(*options***)**

> specifies option for the subproblem. For more information about this option, see Chapter 13, "The Decomposition Algorithm."

Details: MILP Solver

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how to enhance its progress by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP^0) associated with the root is identical to the original problem, which is called (MILP), given in the section "Overview: MILP Solver" on page 247.

The linear programming relaxation (LP^0) of (MILP^0) can be written as

$$
\begin{aligned}
\min \quad & \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \; \{\geq, =, \leq\} \; \mathbf{b} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{aligned}
$$

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider $\bar{x}^0$, the optimal solution to (LP^0), which is usually obtained by using the dual simplex algorithm. If $\bar{x}_i^0$ is an integer for all $i \in \mathcal{S}$, then $\bar{x}^0$ is an optimal solution to (MILP). Suppose that for some $i \in \mathcal{S}$, $\bar{x}_i^0$ is nonintegral. In that case the algorithm defines two new subproblems (MILP^1) and (MILP^2), descendants of the parent subproblem (MILP^0). The subproblem (MILP^1) is identical to (MILP^0) except for the additional constraint

$$ x_i \leq \lfloor \bar{x}_i^0 \rfloor $$

and the subproblem (MILP^2) is identical to (MILP^0) except for the additional constraint

$$ x_i \geq \lceil \bar{x}_i^0 \rceil $$

The notation $\lfloor y \rfloor$ represents the largest integer that is less than or equal to y, and the notation $\lceil y \rceil$ represents the smallest integer that is greater than or equal to y. The two preceding constraints can be handled by modifying the bounds of the variable x_i rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have $\bar{x}^0$ as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable x_i is called the *branching variable*.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is *integer feasible* (that is, x_i is an integer for all $i \in \mathcal{S}$), then its objective value provides an *upper bound* for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section "Controlling the Branch-and-Bound Algorithm" on page 261 for details.

Upper bounds from integer feasible solutions can be used to *fathom* or *cut off* active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if z is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to z can be discarded.

It is important to realize that mixed integer linear programs are non-deterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case $2^{10} = 1{,}024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case $2^{20} = 1{,}048{,}576$ nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has 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.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the *branch-and-cut algorithm*. Preprocessing can reduce problem size and improve problem solvability. The MILP solver in PROC OPTMODEL employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). The MILP solver in PROC OPTMODEL implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let (LP^k) be the linear programming relaxation of subproblem (MILP^k). Also, let

$$f_i(k) = \bar{x}_i^k - \lfloor \bar{x}_i^k \rfloor$$

where $\bar{x}^k$ is the optimal solution to the relaxation problem (LP^k) solved at node k.

Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

AUTOMATIC enables the MILP solver to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\bar{x}^k$, a relaxed optimal solution of (MILP^k), is used to define two active subproblems. In the following list, "INTTOL" refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section "Control Options" on page 252.

AUTOMATIC   enables the MILP solver to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS   chooses as the branching variable the variable x_i such that i maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

MININFEAS   chooses as the branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

PSEUDO   chooses as the branching variable the variable x_i such that i maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG   chooses as the branching variable the variable x_i such that i maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMODEL by attaching branching priorities to the integer variables in your model by using the .priority suffix. More information about this suffix is available in the section "Integer Variable Suffixes" on page 134 in Chapter 5. For an example in which branching priorities are used, see Example 7.3.

Presolve and Probing

The MILP solver in PROC OPTMODEL includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a *polyhedron*. Every polyhedron in n-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$. After you add the restriction that some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \; \forall i \in \mathcal{S}\}$, no longer forms a polyhedron.

The *convex hull* of a set X is the minimal convex set that contains X. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(\mathcal{F})$, of $\mathcal{F}$. If you can find $\text{conv}(\mathcal{F})$ and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section "Branch-and-Bound Algorithm" on page 259, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, $\text{conv}(\mathcal{F}) \subseteq \mathcal{Q}$.

Cutting plane techniques are used to tighten the linear relaxation to better approximate $\text{conv}(\mathcal{F})$. Assume you are given a solution $\bar{x}$ to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality ($\pi x \leq \pi^0$), is some half-space with the following characteristics:

- The half-space contains $\text{conv}(\mathcal{F})$; that is, every integer feasible solution is feasible for the cut ($\pi x \leq \pi^0, \forall x \in \mathcal{F}$).

- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut ($\pi \bar{x} > \pi^0$).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are *generic cuts* and *structured cuts*. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of the MILP solver. Table 7.12 lists the various types of cutting planes that are built into the MILP solver. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

Table 7.12 Cutting Planes in the MILP Solver

Generic Cutting Planes	Structured Cutting Planes
Gomory mixed integer	Cliques
Lift-and-project	Flow cover
Mixed integer rounding	Flow path
Mixed lifted 0-1	Generalized upper bound cover
Zero-half	Implied bound
	Knapsack cover

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUT-MIR=, and CUTZEROHALF= options. The valid levels for these options are listed in Table 7.11.

The cut level determines the internal strategy that is used by the MILP solver for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in the MILP solver, can take a great deal of CPU time. Typically the additional tightening of the relaxation helps to speed up the overall process, because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of the MILP solver have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

Primal Heuristics

Primal heuristics, an important component of the MILP solver in PROC OPTMODEL, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)

- locating a reasonably good feasible solution when that is sufficient (sometimes a reasonably good feasible solution is the best the solver can produce within certain time or resource limits)

- providing upper bounds for some bound-tightening techniques

The MILP solver implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics applied by the MILP solver. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations allowed in iterative heuristics. The valid values for this option are listed in Table 7.7.

Node Log

The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node   indicates the sequence number of the current node in the search tree.

Active   indicates the current number of active nodes in the branch-and-bound tree.

Sols   indicates the number of feasible solutions found so far.

BestInteger   indicates the best upper bound (assuming minimization) found so far.

BestBound   indicates the best lower bound (assuming minimization) found so far.

Gap   indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.

Time   indicates the elapsed real time.

The LOGFREQ= option can be used to control the amount of information printed in the node log. By default a new entry is included in the log at the first node, at the last node, and at 100-node intervals. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 7.3 shows a sample node log.

Figure 7.3 Sample Node Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 10 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 10 integer variables.
NOTE: The problem has 2 linear constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 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 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 10 variables, 2 constraints, and 20 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger      BestBound      Gap     Time
            0        1       3     85.0000000    178.0000000   52.25%        0
            0        1       3     85.0000000     88.0955497    3.51%        0
            0        1       3     85.0000000     88.0626822    3.48%        0
            0        1       3     85.0000000     87.9666563    3.37%        0
            0        1       3     85.0000000     87.9661593    3.37%        0
            0        1       3     85.0000000     87.8181818    3.21%        0
NOTE: The MILP solver added 2 cuts with 13 cut coefficients at the root.
            3        0       5     87.0000000     87.0000000    0.00%        0
NOTE: Optimal.
NOTE: Objective = 87.
```

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the MILP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 7.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 7.4 ODS Table ProblemStatistics

```
                          ProblemStatistics

 Obs    Label1                                   cValue1              nValue1

  1     Number of Constraint Matrix Nonzeros     8                   8.000000
  2     Maximum Constraint Matrix Coefficient    3                   3.000000
  3     Minimum Constraint Matrix Coefficient    1                   1.000000
  4     Average Constraint Matrix Coefficient    1.875               1.875000
  5                                                                         .
  6     Number of Objective Nonzeros             3                   3.000000
  7     Maximum Objective Coefficient            4                   4.000000
  8     Minimum Objective Coefficient            2                   2.000000
  9     Average Objective Coefficient            3                   3.000000
 10                                                                         .
 11     Number of RHS Nonzeros                   3                   3.000000
 12     Maximum RHS                              7                   7.000000
 13     Minimum RHS                              4                   4.000000
 14     Average RHS                              5.3333333333        5.333333
 15                                                                         .
 16     Maximum Number of Nonzeros per Column    3                   3.000000
 17     Minimum Number of Nonzeros per Column    2                   2.000000
 18     Average Number of Nonzeros per Column    2                   2.000000
 19                                                                         .
 20     Maximum Number of Nonzeros per Row       3                   3.000000
 21     Minimum Number of Nonzeros per Row       2                   2.000000
 22     Average Number of Nonzeros per Row       2                   2.000000
```

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure defines a macro variable named _OROPTMODEL_. This variable contains a character string that indicates the status of the solver upon termination. The contents of the macro variable depend on which solver was invoked. For the MILP solver, the various terms of _OROPTMODEL_ are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK   The solver terminated normally.

SYNTAX_ERROR   Syntax was used incorrectly.

DATA_ERROR   The input data was inconsistent.

OUT_OF_MEMORY   Insufficient memory was allocated to the solver.

IO_ERROR   A problem occurred in reading or writing data.

SEMANTIC_ERROR   An evaluation error, such as an invalid operand type, was found.

ERROR   The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC   The branch-and-cut algorithm produced the solution data.

DECOMP   The decomposition algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of small INTTOL= value.
TARGET	The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by solver.
SOLUTION_LIM	The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified by the MAXTIME= option and found a solution.
TIME_LIM_NOSOL	The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL	The solver was stopped by user but still found a solution.
ABORT_NOSOL	The solver was stopped by user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

RELATIVE_GAP

specifies the relative gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the MILP solver. The relative gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-}10 + | \text{BestBound} |)$$

ABSOLUTE_GAP

specifies the absolute gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the MILP solver. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY

indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY

indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY

indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND

specifies the best LP objective value of all unprocessed nodes on the branch-and-bound tree at the end of execution. A missing value indicates that the MILP solver has processed either all or none of the nodes on the branch-and-bound tree.

NODES

specifies the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

ITERATIONS

indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME

indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME

indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERA-TION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: MILP Solver

This section contains examples that illustrate the options and syntax of the MILP solver in PROC OPT-
MODEL. Example 7.1 illustrates the use of PROC OPTMODEL to solve an employee scheduling problem.
Example 7.2 discusses a multicommodity transshipment problem with fixed charges. Example 7.3 demon-
strates how to warm start the MILP solver. Example 7.4 shows the solution of an instance of the traveling
salesman problem in PROC OPTMODEL. Other examples of mixed integer linear programs, along with
example SAS code, are given in Chapter 11.

Example 7.1: Scheduling

The following example has been adapted from the example "A Scheduling Problem" in Chapter 5, "The LP
Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

Scheduling is a common application area in which mixed integer linear programming techniques are used. In
this example, you have eight one-hour time slots in each of five days. You have to assign four employees to
these time slots so that each slot is covered every day. You allow the employees to specify preference data for
each slot on each day. In addition, the following constraints must be satisfied:

- Each employee has some time slots for which he or she is unavailable (OneEmpPerSlot).

- Each employee must have either time slot 4 or time slot 5 off for lunch (EmpMustHaveLunch).

- Each employee can work at most two time slots in a row (AtMost2ConSlots).

- Each employee can work only a specified number of hours in the week (WeeklyHoursLimit).

To formulate this problem, let i denote a person, j denote a time slot, and k denote a 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. Let h_i denote the number of hours in a week that person i will work. The formulation of
this problem follows:

$$
\begin{aligned}
\max \quad & \sum_{ijk} p_{ijk} x_{ijk} \\
\text{s.t.} \quad & \sum_{i} x_{ijk} & = \; & 1 & \forall j, k & \quad (\text{OneEmpPerSlot}) \\
& x_{i4k} + x_{i5k} & \leq \; & 1 & \forall i, k & \quad (\text{EmpMustHaveLunch}) \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} & \leq \; & 2 & \forall i, k, \text{ and } l \leq 6 & \quad (\text{AtMost2ConSlots}) \\
& \sum_{jk} x_{ijk} & \leq \; & h_i & \forall i & \quad (\text{WeeklyHoursLimit}) \\
& x_{ijk} & = \; & 0 & \forall i, j, k \text{ s.t. } p_{ijk} > 0 \\
& x_{ijk} \in \{0, 1\} & & & \forall i, j, k
\end{aligned}
$$

The following data set preferences gives the preferences 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. The data set maxhours gives the maximum number of hours each
employee can work per week.

```
data preferences;
   input name $ slot mon tue wed thu fri;
   datalines;
marc  1    10 10 10 10 10
marc  2     9  9  9  9  9
marc  3     8  8  8  8  8
marc  4     1  1  1  1  1
marc  5     1  1  1  1  1
marc  6     1  1  1  1  1
marc  7     1  1  1  1  1
marc  8     1  1  1  1  1
mike  1    10  9  8  7  6
mike  2    10  9  8  7  6
mike  3    10  9  8  7  6
mike  4    10  3  3  3  3
mike  5     1  1  1  1  1
mike  6     1  2  3  4  5
mike  7     1  2  3  4  5
mike  8     1  2  3  4  5
bill  1    10 10 10 10 10
bill  2     9  9  9  9  9
bill  3     8  8  8  8  8
bill  4     0  0  0  0  0
bill  5     1  1  1  1  1
bill  6     1  1  1  1  1
bill  7     1  1  1  1  1
bill  8     1  1  1  1  1
bob   1    10  9  8  7  6
bob   2    10  9  8  7  6
bob   3    10  9  8  7  6
bob   4    10  3  3  3  3
bob   5     1  1  1  1  1
bob   6     1  2  3  4  5
bob   7     1  2  3  4  5
bob   8     1  2  3  4  5
;

data maxhours;
   input name $ hour;
   datalines;
marc  20
mike  20
bill  20
bob   20
;
```

Using PROC OPTMODEL, you can model and solve the scheduling problem as follows:

```
proc optmodel;

   /* read in the preferences and max hours from the data sets */
   set <string,num> DailyEmployeeSlots;
   set <string>     Employees;

   set <num>    TimeSlots = (setof {<name,slot> in DailyEmployeeSlots} slot);
   set <string> WeekDays  = {"mon","tue","wed","thu","fri"};

   num WeeklyMaxHours{Employees};
   num PreferenceWeights{DailyEmployeeSlots,Weekdays};
   num NSlots = card(TimeSlots);

   read data preferences into DailyEmployeeSlots=[name slot]
        {day in Weekdays} <PreferenceWeights[name,slot,day] = col(day)>;
   read data maxhours into Employees=[name] WeeklyMaxHours=hour;

   /* declare the binary assignment variable x[i,j,k] */
   var Assign{<name,slot> in DailyEmployeeSlots, day in Weekdays} binary;

   /* for each p[i,j,k] = 0, fix x[i,j,k] = 0 */
   for {<name,slot> in DailyEmployeeSlots, day in Weekdays:
      PreferenceWeights[name,slot,day] = 0}
        fix Assign[name,slot,day] = 0;

   /* declare the objective function */
   max TotalPreferenceWeight =
      sum{<name,slot> in DailyEmployeeSlots, day in Weekdays}
         PreferenceWeights[name,slot,day] * Assign[name,slot,day];

   /* declare the constraints */
   con OneEmpPerSlot{slot in TimeSlots, day in Weekdays}:
      sum{name in Employees} Assign[name,slot,day] = 1;

   con EmpMustHaveLunch{name in Employees, day in Weekdays}:
      Assign[name,4,day] + Assign[name,5,day] <= 1;

   con AtMost2ConsSlots{name in Employees, start in 1..NSlots-2,
                           day in Weekdays}:
      Assign[name,start,day] + Assign[name,start+1,day]
           + Assign[name,start+2,day] <= 2 ;

   con WeeklyHoursLimit{name in Employees}:
      sum{slot in TimeSlots, day in Weekdays} Assign[name,slot,day]
           <= WeeklyMaxHours[name];

   /* solve the model */
   solve with milp;

   /* clean up the solution */
   for {<name,slot> in DailyEmployeeSlots, day in Weekdays}
```

```
        Assign[name,slot,day] = round(Assign[name,slot,day],1e-6);

     create data report from [name slot]={<name,slot> in DailyEmployeeSlots:
        max {day in Weekdays} Assign[name,slot,day] > 0}
           {day in Weekdays} <col(day)=(if Assign[name,slot,day] > 0
           then Assign[name,slot,day] else .)>;
   quit;
```

The following statements demonstrate how to use the TABULATE procedure to display a schedule that shows how the eight time slots are covered for the week:

```
   title 'Reported Solution';
   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;
```

The output from the preceding code is displayed in Output 7.1.1.

Output 7.1.1 Scheduling Reported Solution

```
                          Reported Solution

        ------------------------------------------------------------------
        |                 |  mon   |  tue   |  wed   |  thu   |  fri   |
        |-----------------+--------+--------+--------+--------+--------|
        |slot    |name    |        |        |        |        |        |
        |--------+--------|        |        |        |        |        |
        |1       |marc    |  xxx   |  xxx   |  xxx   |  xxx   |  xxx   |
        |--------+--------+--------+--------+--------+--------+--------|
        |2       |marc    |        |  xxx   |  xxx   |  xxx   |  xxx   |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |  xxx   |        |        |        |        |
        |--------+--------+--------+--------+--------+--------+--------|
        |3       |bill    |        |        |        |  xxx   |  xxx   |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |  xxx   |  xxx   |  xxx   |        |        |
        |--------+--------+--------+--------+--------+--------+--------|
        |4       |bob     |  xxx   |  xxx   |  xxx   |        |        |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |        |        |        |  xxx   |  xxx   |
        |--------+--------+--------+--------+--------+--------+--------|
        |5       |bill    |  xxx   |  xxx   |  xxx   |        |  xxx   |
        |        |--------+--------+--------+--------+--------+--------|
        |        |bob     |        |        |        |  xxx   |        |
        |--------+--------+--------+--------+--------+--------+--------|
        |6       |bob     |        |        |  xxx   |        |  xxx   |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |  xxx   |  xxx   |        |  xxx   |        |
        |--------+--------+--------+--------+--------+--------+--------|
        |7       |bob     |  xxx   |        |  xxx   |  xxx   |  xxx   |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |        |  xxx   |        |        |        |
        |--------+--------+--------+--------+--------+--------+--------|
        |8       |bob     |  xxx   |  xxx   |        |  xxx   |        |
        |        |--------+--------+--------+--------+--------+--------|
        |        |mike    |        |        |  xxx   |        |  xxx   |
        ------------------------------------------------------------------
```

Example 7.2: Multicommodity Transshipment Problem with Fixed Charges

The following example has been adapted from the example "A Multicommodity Transshipment Problem with Fixed Charges" in Chapter 5, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

This example illustrates the use of PROC OPTMODEL to generate a mixed integer linear program to solve a multicommodity network flow model with fixed charges. Consider a network with nodes *N*, arcs *A*, and a set *C* of commodities to be shipped between the nodes. The commodities are defined in the data set COMMODITY_DATA, as follows:

```
title 'Multicommodity Transshipment Problem with Fixed Charges';

data commodity_data;
   do c = 1 to 4;
      output;
   end;
run;
```

Shipping cost s_{ijc} is for each of the four commodities c across each of the arcs (i, j). In addition, there is a fixed charge f_{ij} for the use of each arc (i, j). The shipping costs and fixed charges are defined in the data set ARC_DATA, as follows:

```
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;
```

The supply (positive numbers) or demand (negative numbers) d_{ic} at each of the nodes for each commodity c is shown in the data set SUPPLY_DATA, as follows:

```
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;
```

Let x_{ijc} define the flow of commodity c across arc (i, j). Let $y_{ij} = 1$ if arc (i, j) is used, and 0 otherwise. Since the total flow on an arc (i, j) must be at most the total demand across all nodes $k \in N$, you can define the trivial upper bound u_{ijc} as

$$x_{ijc} \le u_{ijc} = \sum_{k \in N \mid d_{kc} < 0} (-d_{kc})$$

This model can be represented using the following mixed integer linear program:

$$\min \quad \sum_{(i,j) \in A} \sum_{c \in C} s_{ijc} x_{ijc} + \sum_{(i,j) \in A} f_{ij} y_{ij}$$

$$\text{s.t.} \quad \sum_{j \in N \mid (i,j) \in A} x_{ijc} - \sum_{j \in N \mid (j,i) \in A} x_{jic} \quad \leq \quad d_{ic} \qquad \forall i \in N, c \in C \qquad \text{(balance\_con)}$$

$$x_{ijc} \quad \leq \quad u_{ijc} y_{ij} \quad \forall (i,j) \in A, c \in C \quad \text{(fixed\_charge\_con)}$$

$$x_{ijc} \quad \geq \quad 0 \qquad \forall (i,j) \in A, c \in C$$

$$y_{ij} \in \{0,1\} \qquad \forall (i,j) \in A$$

Constraint (balance_con) ensures conservation of flow for both supply and demand. Constraint (fixed_charge_con) models the fixed charge cost by forcing $y_{ij} = 1$ if $x_{ijc} > 0$ for some commodity $c \in C$.

The PROC OPTMODEL statements follow:

```
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 the PROC LP example 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 ensures that only the nonzero parts of the solution appear in the SOLUTION data set.

The problem summary, solution summary, and the output from the two PRINT statements are shown in Output 7.2.1.

Output 7.2.1 Multicommodity Transshipment Problem with Fixed Charges Solution Summary

Multicommodity Transshipment Problem with Fixed Charges

The OPTMODEL Procedure

[1]	[2]	[3]	unit_cost
Chicago	NY	1	75
Chicago	NY	2	75
Chicago	NY	3	75
Chicago	NY	4	75
StLouis	NY	1	80
StLouis	NY	2	80
StLouis	NY	3	80
StLouis	NY	4	80
farm-a	Chicago	1	20
farm-a	Chicago	2	15
farm-a	Chicago	3	17
farm-a	Chicago	4	22
farm-a	StLouis	1	30
farm-a	StLouis	2	25
farm-a	StLouis	3	27
farm-a	StLouis	4	22
farm-b	Chicago	1	15
farm-b	Chicago	2	15
farm-b	Chicago	3	15
farm-b	Chicago	4	30
farm-c	Chicago	1	30
farm-c	Chicago	2	30
farm-c	Chicago	3	10
farm-c	Chicago	4	10
farm-c	StLouis	1	10
farm-c	StLouis	2	9
farm-c	StLouis	3	11
farm-c	StLouis	4	10

Output 7.2.1 *continued*

```
                              fixed_
            [1]         [2]    charge

          Chicago    NY          200
          StLouis    NY          200
          farm-a     Chicago     100
          farm-a     StLouis     150
          farm-b     Chicago      75
          farm-c     Chicago     100
          farm-c     StLouis      75


                    supply
                1       2       3       4

Chicago         0       0       0       0
NY           -150    -200     -50     -75
StLouis         0       0       0       0
farm-a        100     100      40       0
farm-b        100     200      50      50
farm-c         40     100      75     100


                Problem Summary

Objective Sense              Minimization
Objective Function              TotalCost
Objective Type                     Linear

Number of Variables                    35
Bounded Above                           0
Bounded Below                           0
Bounded Below and Above                35
Free                                    0
Fixed                                   0
Binary                                  7
Integer                                 0

Number of Constraints                  52
Linear LE (<=)                         52
Linear EQ (=)                           0
Linear GE (>=)                          0
Linear Range                            0

Constraint Coefficients               112


            Performance Information

  Execution Mode        Single-Machine
  Number of Threads                   1
```

Output 7.2.1 *continued*

```
                    Solution Summary

Solver                                        MILP
Algorithm                          Branch and Cut
Objective Function                       TotalCost
Solution Status        Optimal within Relative Gap
Objective Value                              42825

Relative Gap                          2.3350852E-7
Absolute Gap                                  0.01
Primal Infeasibility                             0
Bound Infeasibility                              0
Integer Infeasibility                            0

Best Bound                                42824.99
Nodes                                            1
Iterations                                      30
Presolve Time                                 0.00
Solution Time                                 0.14


                                       Amount
          [1]        [2]       [3]     Shipped

          Chicago    NY        1          110
          Chicago    NY        2          100
          Chicago    NY        3           50
          Chicago    NY        4           75
          StLouis    NY        1           40
          StLouis    NY        2          100
          StLouis    NY        3            0
          StLouis    NY        4            0
          farm-a     Chicago   1           10
          farm-a     Chicago   2           10
          farm-a     Chicago   3            0
          farm-a     Chicago   4            0
          farm-a     StLouis   1            0
          farm-a     StLouis   2            0
          farm-a     StLouis   3            0
          farm-a     StLouis   4            0
          farm-b     Chicago   1          100
          farm-b     Chicago   2           90
          farm-b     Chicago   3            0
          farm-b     Chicago   4            0
          farm-c     Chicago   1            0
          farm-c     Chicago   2            0
          farm-c     Chicago   3           50
          farm-c     Chicago   4           75
          farm-c     StLouis   1           40
          farm-c     StLouis   2          100
          farm-c     StLouis   3            0
          farm-c     StLouis   4            0
```

Example 7.3: Facility Location

Consider the classic facility location problem. Given a set L of customer locations and a set F of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand d_i must be satisfied, and each facility has a demand capacity limit C. The total cost is the sum of the distances c_{ij} between facility j and its assigned customer i, plus a fixed charge f_j for building a facility at site j. Let $y_j = 1$ represent choosing site j to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer i to facility j, and 0 otherwise. This model can be formulated as the following integer linear program:

$$
\begin{aligned}
\min \quad & \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j \\
\text{s.t.} \quad & \sum_{j \in F} x_{ij} = 1 && \forall i \in L && \text{(assign\_def)} \\
& x_{ij} \leq y_j && \forall i \in L, j \in F && \text{(link)} \\
& \sum_{i \in L} d_i x_{ij} \leq C y_j && \forall j \in F && \text{(capacity)} \\
& x_{ij} \in \{0, 1\} && \forall i \in L, j \in F \\
& y_j \in \{0, 1\} && \forall j \in F
\end{aligned}
$$

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$
\min \quad \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij}
$$

First, construct a random instance of this problem by using the following DATA steps:

```
title 'Facility Location Problem';


%let NumCustomers  = 50;
%let NumSites      = 10;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;


/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
```

```
        demand = ranuni(&seed) * &MaxDemand;
        output;
    end;
run;


/* generate random site locations and fixed charge */
data sdata(drop=i);
    length name $8;
    do i = 1 to &NumSites;
        name = compress('SITE'||put(i,best.));
        x = ranuni(&seed) * &xmax;
        y = ranuni(&seed) * &ymax;
        fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
        output;
    end;
run;
```

The following PROC OPTMODEL statements first generate and solve the model with the no-fixed-charge variant of the cost function. Next, they solve the fixed-charge model. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for the MILP solver. Use the PRIMALIN option to provide an incumbent solution (warm start).

```
proc optmodel;
    set <str> CUSTOMERS;
    set <str> SITES init {};
    /* x and y coordinates of CUSTOMERS and SITES */
    num x {CUSTOMERS union SITES};
    num y {CUSTOMERS union SITES};
    num demand {CUSTOMERS};
    num fixed_charge {SITES};

    /* distance from customer i to site j */
    num dist {i in CUSTOMERS, j in SITES}
        = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

    read data cdata into CUSTOMERS=[name] x y demand;
    read data sdata into SITES=[name] x y fixed_charge;

    var Assign {CUSTOMERS, SITES} binary;
    var Build {SITES} binary;

    min CostNoFixedCharge
        = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
    min CostFixedCharge
        = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

    /* each customer assigned to exactly one site */
    con assign_def {i in CUSTOMERS}:
        sum {j in SITES} Assign[i,j] = 1;

    /* if customer i assigned to site j, then facility must be built at j */
    con link {i in CUSTOMERS, j in SITES}:
        Assign[i,j] <= Build[j];
```

```
   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];

   /* solve the MILP with no fixed charges */
   solve obj CostNoFixedCharge with milp / logfreq = 500;

   /* clean up the solution */
   for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j]);
   for {j in SITES} Build[j] = round(Build[j]);

   call symput('varcostNo',put(CostNoFixedCharge,6.1));

   /* create a data set for use by GPLOT */
   create data CostNoFixedCharge_Data from
      [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
      xi=x[i] yi=y[i] xj=x[j] yj=y[j];

   /* solve the MILP, with fixed charges with warm start */
   solve obj CostFixedCharge with milp / primalin logfreq = 500;

   /* clean up the solution */
   for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j]);
   for {j in SITES} Build[j] = round(Build[j]);

   num varcost = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j].sol;
   num fixcost = sum {j in SITES} fixed_charge[j] * Build[j].sol;
   call symput('varcost', put(varcost,6.1));
   call symput('fixcost', put(fixcost,5.1));
   call symput('totalcost', put(CostFixedCharge,6.1));

   /* create a data set for use by GPLOT */
   create data CostFixedCharge_Data from
      [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
      xi=x[i] yi=y[i] xj=x[j] yj=y[j];
quit;
```

The information printed in the log for the no-fixed-charge model is displayed in Output 7.3.1.

Output 7.3.1 OPTMODEL Log for Facility Location with No Fixed Charges

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000
      constraint coefficients.
NOTE: The MILP solver is called.
         Node  Active    Sols    BestInteger      BestBound      Gap     Time
            0       1       2    972.1737321              0    972.2        0
            0       1       2    972.1737321    961.2403449    1.14%        0
            0       1       2    972.1737321    966.4828342    0.59%        0
            0       1       3    966.4832160    966.4828342    0.00%        0
NOTE: The MILP solver added 3 cuts with 102 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 966.48321599.
```

The results from the warm start approach are shown in Output 7.3.2.

Output 7.3.2 OPTMODEL Log for Facility Location with Fixed Charges, Using Warm Start

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010
      constraint coefficients.
NOTE: The MILP solver is called.
          Node   Active    Sols    BestInteger        BestBound        Gap     Time
             0        1       3    16070.0150023                0      16070        0
             0        1       3    16070.0150023    9946.2514269     61.57%        0
             0        1       3    16070.0150023    9965.3660728     61.26%        0
             0        1       3    16070.0150023    9992.5098441     60.82%        1
             0        1       3    16070.0150023   10008.1168033     60.57%        1
             0        1       5    12714.8421527   10015.7276884     26.95%        1
             0        1       5    12714.8421527   10022.7765711     26.86%        2
             0        1       6    12697.5037984   10031.4554268     26.58%        2
             0        1       6    12697.5037984   10038.3630945     26.49%        2
             0        1       6    12697.5037984   10040.1836986     26.47%        3
             0        1       7    12681.6662283   10041.2236063     26.30%        3
             0        1       7    12681.6662283   10041.8276236     26.29%        3
             0        1       7    12681.6662283   10042.4192876     26.28%        3
             0        1       7    12681.6662283   10042.4192876     26.28%        6
NOTE: The MILP solver added 30 cuts with 862 cut coefficients at the root.
            12       10      10    10982.8321843   10050.5594169      9.28%        7
           113       15      11    10962.4644900   10940.8960590      0.20%       11
           141       16      12    10948.4603467   10942.5795938      0.05%       12
           173       17      13    10948.4603465   10946.1080596      0.02%       12
           181        9      13    10948.4603465   10947.6003284      0.01%       12
NOTE: Optimal within relative gap.
NOTE: Objective = 10948.460346.
```

The following two SAS programs produce a plot of the solutions for both variants of the model, using data sets produced by PROC OPTMODEL:

```
title1 h=1.5 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";

data csdata;
   set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;

/* create Annotate data set to draw line between customer and assigned site */
%annomac;
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostNoFixedCharge_Data(keep=xi yi xj yj);
```

```
      %LINE(xi, yi, xj, yj, *, 1, 1);
   run;

   proc gplot data=csdata anno=anno;
      axis1 label=none order=(0 to &xmax by 10);
      axis2 label=none order=(0 to &ymax by 10);
      symbol1 value=dot interpol=none
         pointlabel=("#name" nodropcollisions height=1) cv=black;
      symbol2 value=diamond interpol=none
         pointlabel=("#name" nodropcollisions color=blue height=1) cv=blue;
      plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
   run;
   quit;
```

The output of the first program is shown in Output 7.3.3.

Output 7.3.3 Solution Plot for Facility Location with No Fixed Charges
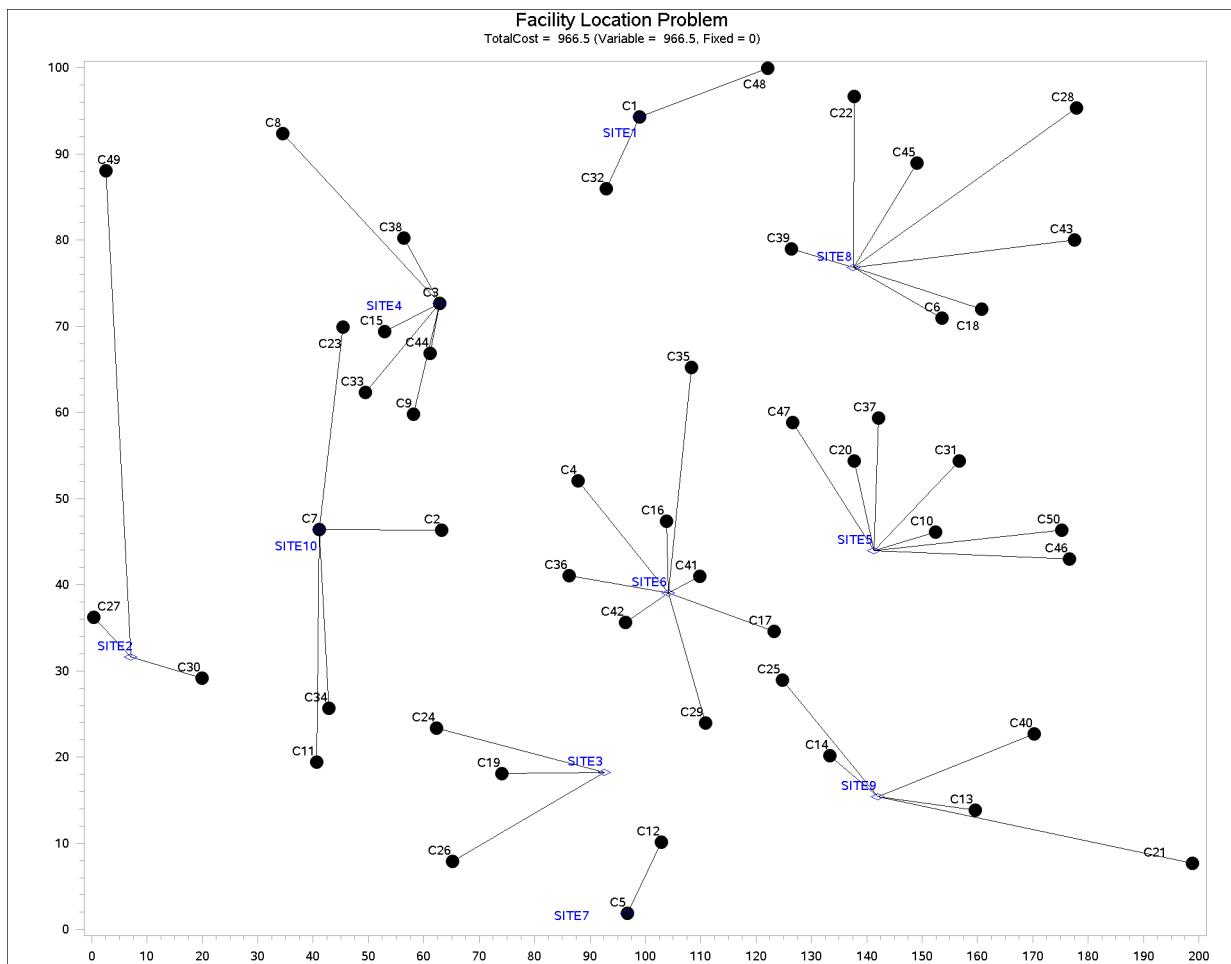
The output of the second program is shown in Output 7.3.4.

```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";

/* create Annotate data set to draw line between customer and assigned site */
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;

proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=1) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=1) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

Output 7.3.4 Solution Plot for Facility Location with Fixed Charges
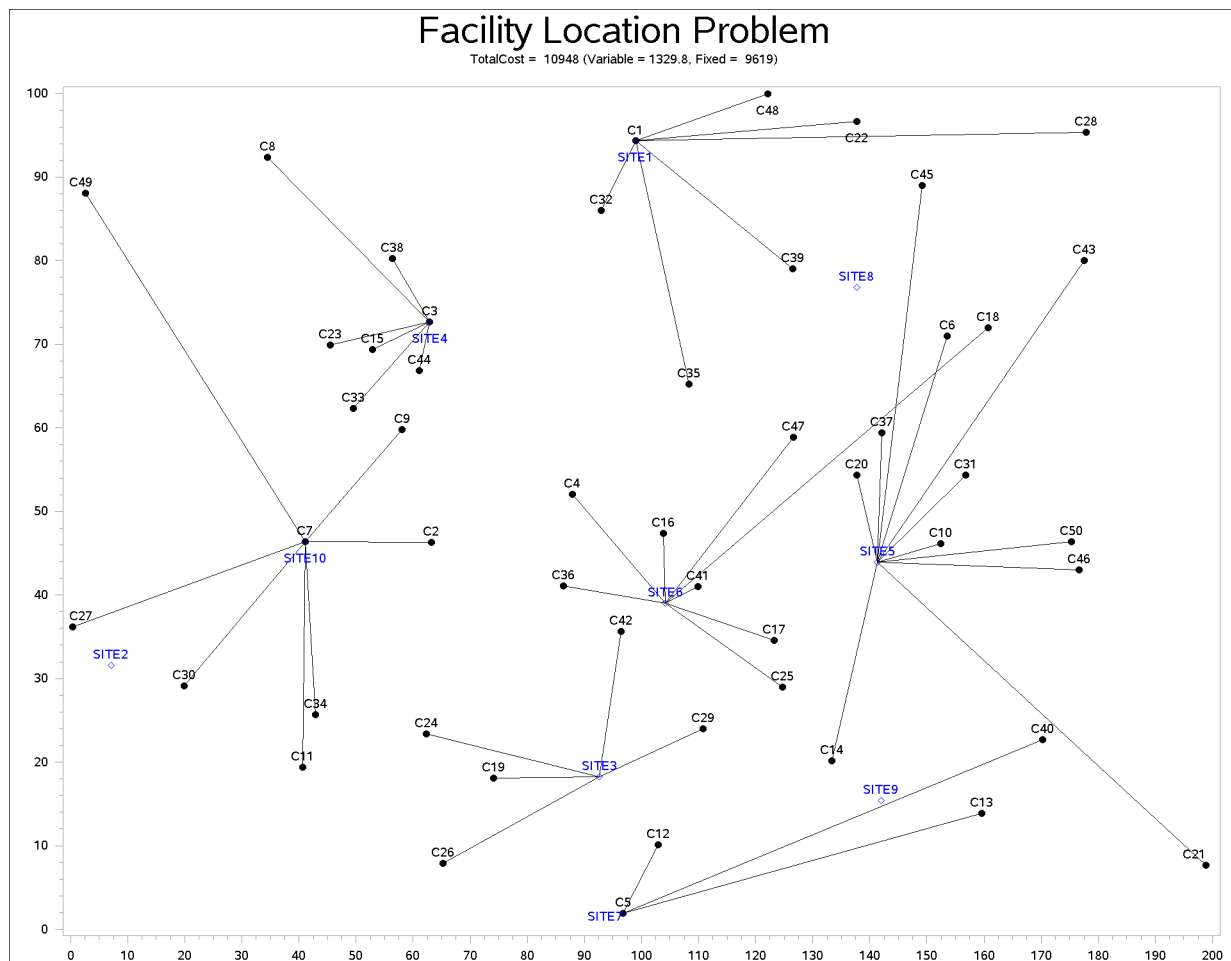
The economic trade-off for the fixed-charge model forces you to build fewer sites and push more demand to each site.

It is possible to expedite the solution of the fixed-charge facility location problem by choosing appropriate branching priorities for the decision variables. Recall that for each site j, the value of the variable y_j determines whether or not a facility is built on that site. Suppose you decide to branch on the variables y_j before the variables x_{ij}. You can set a higher branching priority for y_j by using the .priority suffix for the Build variables in PROC OPTMODEL, as follows:

```
for{j in SITES} Build[j].priority=10;
```

Setting higher branching priorities for certain variables is not guaranteed to speed up the MILP solver, but it can be helpful in some instances. The following program creates and solves an instance of the facility location problem, giving higher priority to the variables y_j. The LOGFREQ= option is used to abbreviate the node log.

```
%let NumCustomers  = 45;
%let NumSites      = 8;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 2345;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      fixed_charge = (abs(&xmax/2-x) + abs(&ymax/2-y)) / 2;
      output;
   end;
run;
```

```
proc optmodel;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
      = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;

   min CostFixedCharge
      = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j]
         + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {j in SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {i in CUSTOMERS, j in SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <= &SiteCapacity * Build[j];

   /* assign priority to Build variables (y) */
   for{j in SITES} Build[j].priority=10;

   /* solve the MILP with fixed charges, using branching priorities */
   solve obj CostFixedCharge with milp / logfreq=1000;
quit;
```

The resulting output is shown in

Output 7.3.5 PROC OPTMODEL Log for Facility Location with Branching Priorities

```
NOTE: There were 45 observations read from the data set WORK.CDATA.
NOTE: There were 8 observations read from the data set WORK.SDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 368 variables (0 free, 0 fixed).
NOTE: The problem has 368 binary and 0 integer variables.
NOTE: The problem has 413 linear constraints (368 LE, 45 EQ, 0 GE, 0 range).
NOTE: The problem has 1448 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 368 variables, 413 constraints, and 1448
      constraint coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger       BestBound        Gap     Time
            0        1       3    2823.1827978               0     2823.2        0
            0        1       3    2823.1827978    1727.0208789     63.47%        0
            0        1       3    2823.1827978    1752.0649013     61.13%        0
            0        1       3    2823.1827978    1764.4145721     60.01%        0
            0        1       3    2823.1827978    1771.3542742     59.38%        1
            0        1       3    2823.1827978    1778.1649446     58.77%        1
            0        1       3    2823.1827978    1779.9319639     58.61%        1
            0        1       3    2823.1827978    1780.4398220     58.57%        1
            0        1       6    1903.7976887    1781.4348074      6.87%        2
            0        1       6    1903.7976887    1781.6818875      6.85%        2
            0        1       6    1903.7976887    1781.9825516      6.84%        2
            0        1       6    1903.7976887    1784.3735066      6.69%        2
            0        1       6    1903.7976887    1785.1171912      6.65%        3
            0        1       6    1903.7976887    1785.1318270      6.65%        3
            0        1       6    1903.7976887    1785.1691077      6.65%        3
            0        1       6    1903.7976887    1785.2000207      6.64%        3
            0        1       6    1903.7976887    1785.2000207      6.64%        3
NOTE: The MILP solver added 28 cuts with 767 cut coefficients at the root.
            4        1       8    1852.5588919    1802.5216297      2.78%        4
           52       46      10    1835.0251598    1804.8458565      1.67%        5
          244      215      11    1834.9126635    1808.0082480      1.49%        7
          296      245      12    1830.7850312    1809.1809114      1.19%        8
          523      282      13    1825.1666003    1812.4410890      0.70%       11
          608      293      14    1825.1666001    1814.3493473      0.60%       13
          642      135      15    1821.5115363    1815.4351037      0.33%       13
          691       46      16    1819.9124342    1818.1426710      0.10%       13
          735        4      16    1819.9124342    1819.7466770      0.01%       14
NOTE: Optimal within relative gap.
NOTE: Objective = 1819.9124342.
```

Example 7.4: Traveling Salesman Problem

The traveling salesman problem (TSP) is that of finding a minimum cost *tour* in an undirected graph G with vertex set $V = \{1, \ldots, |V|\}$ and edge set E. A tour is a connected subgraph for which each vertex has degree two. The goal is then to find a tour of minimum total cost, where the total cost is the sum of the costs of the edges in the tour. With each edge $e \in E$ we associate a binary variable x_e, which indicates whether edge e is part of the tour, and a cost $c_e \in \mathbb{R}$. Let $\delta(S) = \{\{i, j\} \in E \mid i \in S, \ j \notin S\}$. Then an integer linear programming (ILP) formulation of the TSP is as follows:

$$
\begin{aligned}
\min \quad & \sum_{e \in E} c_e x_e \\
\text{s.t.} \quad & \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V && \text{(two\_match)} \\
& \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V,\ 2 \leq |S| \leq |V| - 1 && \text{(subtour\_elim)} \\
& x_e \in \{0, 1\} \quad \forall e \in E
\end{aligned}
$$

The equations (two_match) are the *matching constraints*, which ensure that each vertex has degree two in the subgraph, while the inequalities (subtour_elim) are known as the *subtour elimination constraints* (SECs) and enforce connectivity.

Since there is an exponential number $O(2^{|V|})$ of SECs, it is impossible to explicitly construct the full TSP formulation for large graphs. An alternative formulation of polynomial size was introduced by Miller, Tucker, and Zemlin (1960) (MTZ):

$$
\begin{aligned}
\min \quad & \sum_{(i,j) \in E} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j \in V} x_{ij} = 1 && \forall i \in V && \text{(assign\_i)} \\
& \sum_{i \in V} x_{ij} = 1 && \forall j \in V && \text{(assign\_j)} \\
& u_i - u_j + 1 \leq (|V| - 1)(1 - x_{ij}) && \forall (i, j) \in V, i \neq 1, j \neq 1 && \text{(mtz)} \\
& 2 \leq u_i \leq |V| && \forall i \in \{2, .., |V|\}, \\
& x_{ij} \in \{0, 1\} && \forall (i, j) \in E
\end{aligned}
$$

This formulation uses a directed graph. Constraints (assign_i) and (assign_j) now enforce that each vertex has degree two (one edge in, one edge out). The MTZ constraints (mtz) enforce that no subtours exist.

TSPLIB, located at http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html, is a set of benchmark instances for the TSP. The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "st70.tsp";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;
```

The following PROC OPTMODEL statements attempt to solve the TSPLIB instance st70.tsp by using the MTZ formulation:

```
/* direct solution using the MTZ formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i ne j};
   num xc {VERTICES};
   num yc {VERTICES};

   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[_n_] xc=var2 yc=var3;

   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
       init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

   var x {EDGES} binary;
   var u {i in 2..card(VERTICES)} >= 2 <= card(VERTICES);

   /* each vertex has exactly one in-edge and one out-edge */
   con assign_i {i in VERTICES}:
       sum {j in VERTICES: i ne j} x[i,j] = 1;
   con assign_j {j in VERTICES}:
       sum {i in VERTICES: i ne j} x[i,j] = 1;

   /* minimize the total cost */
   min obj
       = sum {<i,j> in EDGES} (if i > j then c[i,j] else c[j,i]) * x[i,j];

   /* no subtours */
   con mtz {<i,j> in EDGES : (i ne 1) and (j ne 1)}:
       u[i] - u[j] + 1 <= (card(VERTICES) - 1) * (1 - x[i,j]);

   solve with milp / maxtime = 600;
quit;
```

It is well known that the MTZ formulation is much weaker than the subtour formulation. The exponential number of SECs makes it impossible, at least in large instances, to use a direct call to the MILP solver with the subtour formulation. For this reason, if you want to solve the TSP with one SOLVE statement, you must use the MTZ formulation and rely strictly on generic cuts and heuristics. Except for very small instances, this is unlikely to be a good approach.

A much more efficient way to tackle the TSP is to dynamically generate the subtour inequalities as *cuts*. Typically this is done by solving the LP relaxation of the two-matching problem, finding violated subtour cuts, and adding them iteratively. The problem of finding violated cuts is known as the *separation problem*. In this case, the separation problem takes the form of a minimum cut problem, which is nontrivial to implement efficiently. Therefore, for the sake of illustration, an integer program is solved at each step of the process.

The initial formulation of the TSP is the integral two-matching problem. You solve this by using PROC OPTMODEL to obtain an integral matching, which is not necessarily a tour. In this case, the separation problem is trivial. If the solution is a connected graph, then it is a tour, so the problem is solved. If the solution is a disconnected graph, then each component forms a violated subtour constraint. These constraints

are added to the formulation, and the integer program is solved again. This process is repeated until the solution defines a tour.

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints:

```
/* iterative solution using the subtour formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i > j};
   num xc {VERTICES};
   num yc {VERTICES};

   num numsubtour init 0;
   set SUBTOUR {1..numsubtour};

   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[var1] xc=var2 yc=var3;

   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
       init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

   var x {EDGES} binary;

   /* minimize the total cost */
   min obj =
       sum {<i,j> in EDGES} c[i,j] * x[i,j];

   /* each vertex has exactly one in-edge and one out-edge */
   con two_match {i in VERTICES}:
       sum {j in VERTICES: i > j} x[i,j]
     + sum {j in VERTICES: i < j} x[j,i] = 2;

   /* no subtours (these constraints are generated dynamically) */
   con subtour_elim {s in 1..numsubtour}:
       sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
           or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

   /* this starts the algorithm to find violated subtours */
   set <num,num> EDGES1;
   set INITVERTICES = setof{<i,j> in EDGES1} i;
   set VERTICES1;
   set NEIGHBORS;
   set <num,num> CLOSURE;
   num component {INITVERTICES};
   num numcomp  init 2;
   num iter     init 1;
   num numiters init 1;
   set ITERS = 1..numiters;
   num sol {ITERS, EDGES};
```

```
   /* initial solve with just matching constraints */
   solve;
   call symput(compress('obj'||put(iter,best.)),
               trim(left(put(round(obj),best.)))));
   for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);

   /* while the solution is disconnected, continue */
   do while (numcomp > 1);
      iter = iter + 1;

      /* find connected components of support graph   */
      EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
      EDGES1 = EDGES1 union {setof {<i,j> in EDGES1} <j,i>};
      VERTICES1 = INITVERTICES;
      CLOSURE = EDGES1;
      for {i in INITVERTICES} component[i] = 0;
      for {i in VERTICES1} do;
         NEIGHBORS = slice(<i,*>,CLOSURE);
         CLOSURE = CLOSURE union (NEIGHBORS cross NEIGHBORS);
      end;

      numcomp = 0;
      do while (card(VERTICES1) > 0);
         numcomp = numcomp + 1;
         for {i in VERTICES1} do;
            NEIGHBORS = slice(<i,*>,CLOSURE);
            for {j in NEIGHBORS} component[j] = numcomp;
            VERTICES1 = VERTICES1 diff NEIGHBORS;
            leave;
         end;
      end;

      if numcomp = 1 then leave;
      numiters = iter;
      numsubtour = numsubtour + numcomp;
      for {comp in 1..numcomp} do;
         SUBTOUR[numsubtour-numcomp+comp]
            = {i in VERTICES: component[i] = comp};
      end;

      solve;
      call symput(compress('obj'||put(iter,best.)),
                  trim(left(put(round(obj),best.)))));
      for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);
   end;

   /* create a data set for use by gplot */
   create data solData from
      [iter i j]={it in ITERS, <i,j> in EDGES: sol[it,i,j] = 1}
      xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];
   call symput('numiters',put(numiters,best.));
quit;
```

You can generate plots of the solution and objective value at each stage by using the following statements:

```
%macro plotTSP;
   %annomac;
   %do i = 1 %to &numiters;
      /* create annotate data set to draw subtours */
      data anno(drop=iter xi yi xj yj);
         %SYSTEM(2, 2, 2);
         set solData(keep=iter xi yi xj yj);
         where iter = &i;
         %LINE(xi, yi, xj, yj, *, 1, 1);
      run;

      title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
      title2;

         axis1 label=none;
         symbol1 value=dot interpol=none
         pointlabel=("#var1" nodropcollisions height=1) cv=black;
         plot var3*var2 / haxis=axis1 vaxis=axis1;
      run;
      quit;
   %end;
%mend plotTSP;

%plotTSP;
```

The plot in Output 7.4.1 shows the solution and objective value at each stage. Notice that each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

NOTE: An alternative way of approaching the TSP is to use a genetic algorithm. See the "Examples" section in Chapter 4, "The GA Procedure" (*SAS/OR User's Guide: Local Search Optimization*), for an example of how to use PROC GA to solve the TSP.

NOTE: See the "Examples" section in Chapter 2, "The OPTNET Procedure" (*SAS/OR User's Guide: Network Optimization Algorithms*). for an example of how to use PROC OPTNET to solve the TSP.

Output 7.4.1 Traveling Salesman Problem Iterative Solution
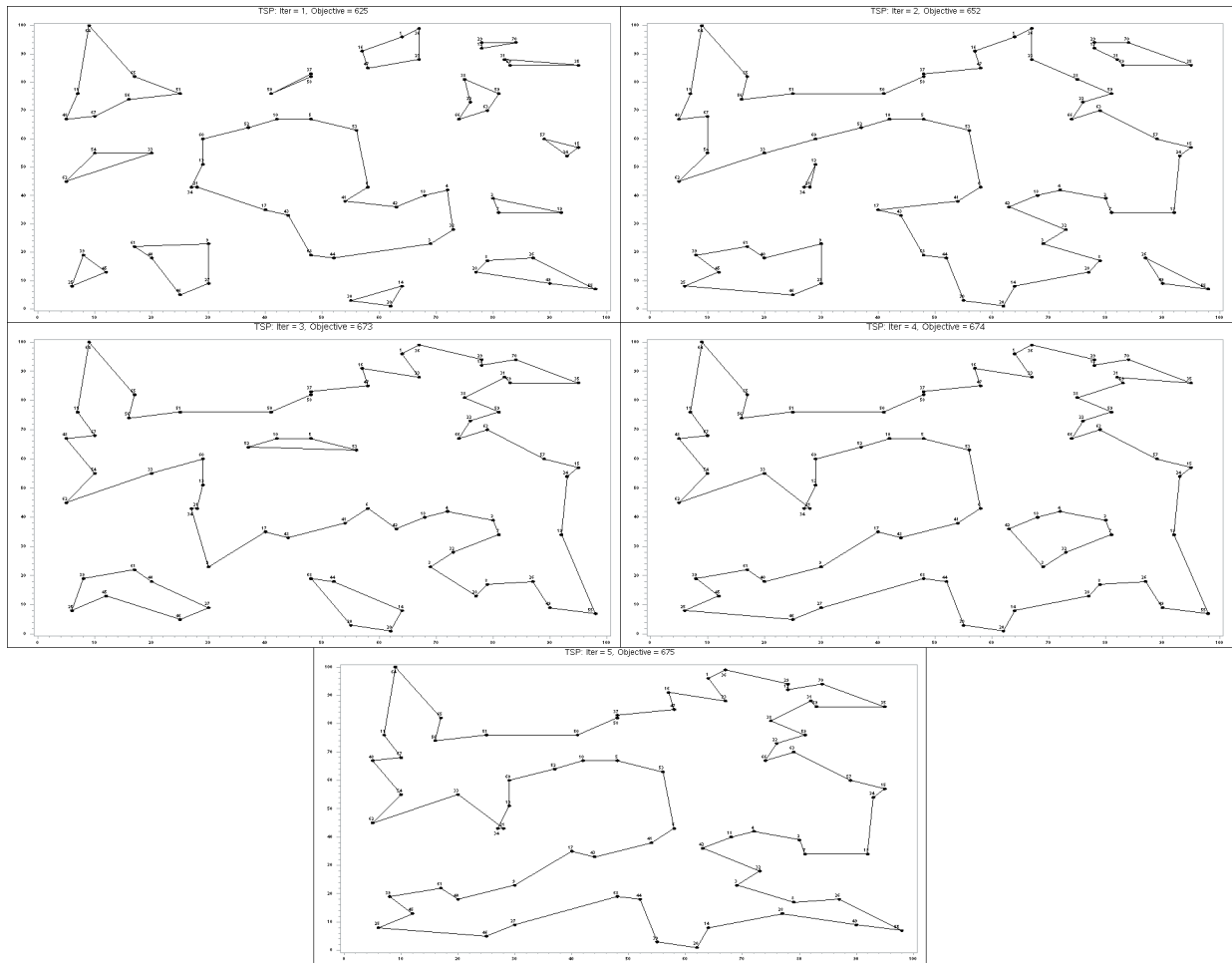
References

Achterberg, T., Koch, T., and Martin, A. (2005), "Branching Rules Revisited," *Operations Research Letters*, 33, 42–54.

Andersen, E. D. and Andersen, K. D. (1995), "Presolving in Linear Programming," *Mathematical Programming*, 71, 221–245.

Atamturk, A. (2004), "Sequence Independent Lifting for Mixed-Integer Programming," *Operations Research*, 52, 487–490.

Dantzig, G. B., Fulkerson, R., and Johnson, S. M. (1954), "Solution of a Large-Scale Traveling Salesman Problem," *Operations Research*, 2, 393–410.

Gondzio, J. (1997), "Presolve Analysis of Linear Programs prior to Applying an Interior Point Method," *INFORMS Journal on Computing*, 9, 73–91.

Land, A. H. and Doig, A. G. (1960), "An Automatic Method for Solving Discrete Programming Problems," *Econometrica*, 28, 497–520.

Linderoth, J. T. and Savelsbergh, M. W. P. (1998), "A Computational Study of Search Strategies for Mixed Integer Programming," *INFORMS Journal on Computing*, 11, 173–187.

Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. (1999), "Cutting Planes in Integer and Mixed Integer Programming," DP 9953, CORE, Université Catholique de Louvain, 1999.

Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960), "Integer Programming Formulations of Traveling Salesman Problems," *Journal of the Association for Computing Machinery*, 7, 326–329.

Savelsbergh, M. W. P. (1994), "Preprocessing and Probing Techniques for Mixed Integer Programming Problems," *ORSA Journal on Computing*, 6, 445–454.

Chapter 8
The Nonlinear Programming Solver

Contents

Overview: NLP Solver

The sparse nonlinear programming (NLP) solver is a component of the OPTMODEL procedure that can solve optimization problems containing both nonlinear equality and inequality constraints. The general nonlinear optimization problem can be defined as

$$
\begin{aligned}
\text{minimize} \quad & f(x) \\
\text{subject to} \quad & h_i(x) = 0, i \in \mathcal{E} = \{1, 2, \ldots, p\} \\
& g_i(x) \geq 0, i \in \mathcal{I} = \{1, 2, \ldots, q\} \\
& l \leq x \leq u
\end{aligned}
$$

where $x \in \mathbb{R}^n$ is the vector of the decision variables; $f : \mathbb{R}^n \mapsto \mathbb{R}$ is the objective function; $h : \mathbb{R}^n \mapsto \mathbb{R}^p$ is the vector of equality constraints—that is, $h = (h_1, \ldots, h_p)$; $g : \mathbb{R}^n \mapsto \mathbb{R}^q$ is the vector of inequality constraints—that is, $g = (g_1, \ldots, g_q)$; and $l, u \in \mathbb{R}^n$ are the vectors of the lower and upper bounds, respectively, on the decision variables.

It is assumed that the functions f, h_i, and g_i are twice continuously differentiable. Any point that satisfies the constraints of the NLP problem is called a *feasible point*, and the set of all those points forms the feasible region of the NLP problem—that is, $\mathcal{F} = \{x \in \mathbb{R}^n : h(x) = 0, g(x) \geq 0, l \leq x \leq u\}$.

The NLP problem can have a unique minimum or many different minima, depending on the type of functions involved. If the objective function is convex, the equality constraint functions are linear, and the inequality constraint functions are concave, then the NLP problem is called a convex program and has a unique minimum. All other types of NLP problems are called nonconvex and can contain more than one minimum, usually called *local minima*. The solution that achieves the lowest objective value of all local minima is called the *global minimum* or *global solution* of the NLP problem. The NLP solver can find the unique minimum of convex programs and a local minimum of a general NLP problem. In addition, the solver is equipped with specific options that enable it to locate the global minimum or a good approximation of it, for those problems that contain many local minima.

The NLP solver implements the following primal-dual methods for finding a local minimum:

- interior point trust-region line-search algorithm

- active-set trust-region line-search algorithm

Both methods can solve small-, medium-, and large-scale optimization problems efficiently and robustly. These methods use exact first and second derivatives to calculate search directions. The memory requirements of both algorithms are reduced dramatically because only nonzero elements of matrices are stored. Convergence of both algorithms is achieved by using a trust-region line-search framework that guides the iterations towards the optimal solution. If a trust-region subproblem fails to provide a suitable step of improvement, a line-search is then used to fine tune the trust-region radius and ensure sufficient decrease in objective function and constraint violations.

The interior point technique implements a primal-dual interior point algorithm in which barrier functions are used to ensure that the algorithm remains feasible with respect to the bound constraints. Interior point methods are extremely useful when the optimization problem contains many inequality constraints and you suspect that most of these constraints will be satisfied as strict inequalities at the optimal solution.

The active-set technique implements an active-set algorithm in which only the inequality constraints that are satisfied as equalities, together with the original equality constraints, are considered. Once that set of constraints is identified, active-set algorithms typically converge faster than interior point algorithms. They converge faster because the size and the complexity of the original optimization problem can be reduced if only few constraints need to be considered.

For optimization problems that contain many local optima, the NLP solver can be run in multistart mode. If the multistart mode is specified, the solver samples the feasible region and generates a number of starting points. Then the local solvers can be called from each of those starting points to converge to different local optima. The local minimum with the smallest objective value is then reported back to the user as the optimal solution.

The NLP solver implements many powerful features that are obtained from recent research in the field of nonlinear optimization algorithms (Akrotirianakis and Rustem 2005; Armand, Gilbert, and Jan-Jégou

2002; Erway, Gill, and Griffin 2007; Forsgren and Gill 1998; Vanderbei 1999; Wächter and Biegler 2006; Yamashita 1998). The term *primal-dual* means that the algorithm iteratively generates better approximations of the decision variables x (usually called *primal* variables) in addition to the dual variables (also referred to as Lagrange multipliers). At every iteration, the algorithm uses a modified Newton's method to solve a system of nonlinear equations. The modifications made to Newton's method are implicitly controlled by the current trust-region radius. The solution of that system provides the direction and the steps along which the next approximation of the local minimum is searched. The active-set algorithm ensures that the primal iterations are always within their bounds—that is, $l \leq x^k \leq u$, for every iteration k. However, the interior approach relaxes this condition by using slack variables, and intermediate iterations might be infeasible.

Getting Started: NLP Solver

The NLP solver consists of two techniques that can solve a wide class of optimization problems efficiently and robustly. In this section two examples that introduce the two techniques of NLP are presented. The examples also introduce basic features of the modeling language of PROC OPTMODEL that is used to define the optimization problem.

The NLP solver can be invoked using the SOLVE statement,

SOLVE WITH NLP *< / options >* **;**

where *options* specify the technique name, termination criteria, and how to display the results in the iteration log. For a detailed description of the *options*, see the section "NLP Solver Options" on page 308.

A Simple Problem

Consider the following simple example of a nonlinear optimization problem:

$$
\begin{aligned}
\text{minimize} \quad & f(x) = (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2 \\
\text{subject to} \quad & x_1 + x_2 + x_3 = 1 \\
& 6x_2 + 4x_3 - x_1^3 - 3 \geq 0 \\
& x_i \geq 0, i = 1, 2, 3
\end{aligned}
$$

The problem consists of a quadratic objective function, a linear equality constraint, and a nonlinear inequality constraint. The goal is to find a local minimum, starting from the point $x^0 = (0.1, 0.7, 0.2)$. You can use the following call to PROC OPTMODEL to find a local minimum:

```
proc optmodel;
   var x{1..3} >= 0;
   minimize f = (x[1] + 3*x[2] + x[3])**2 + 4*(x[1] - x[2])**2;

   con constr1: sum{i in 1..3}x[i] = 1;
   con constr2: 6*x[2] + 4*x[3] - x[1]**3 - 3 >= 0;

   /* starting point */
   x[1] = 0.1;
   x[2] = 0.7;
   x[3] = 0.2;
```

```
    solve with NLP;
    print x;
quit;
```

Because no options have been specified, the default solver (INTERIORPOINT) is used to solve the problem. The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the local minimum. The summaries and the optimal solution are shown in Figure 8.1.

Figure 8.1 Problem Summary, Solution Summary, and the Optimal Solution

```
                        The OPTMODEL Procedure

                           Problem Summary

            Objective Sense              Minimization
            Objective Function                      f
            Objective Type                  Quadratic

            Number of Variables                     3
            Bounded Above                           0
            Bounded Below                           3
            Bounded Below and Above                 0
            Free                                    0
            Fixed                                   0

            Number of Constraints                   2
            Linear LE  (<=)                         0
            Linear EQ  (=)                          1
            Linear GE  (>=)                         0
            Linear Range                            0
            Nonlinear LE  (<=)                      0
            Nonlinear EQ  (=)                       0
            Nonlinear GE  (>=)                      1
            Nonlinear Range                         0


                      Performance Information

             Execution Mode       Single-Machine
             Number of Threads                 4
```

Figure 8.1 *continued*

```
                        Solution Summary

            Solver                              NLP
            Algorithm                 Interior Point
            Objective Function                     f
            Solution Status               Optimal
            Objective Value            1.000017384

            Optimality Error                   5E-7
            Infeasibility           2.5088265E-8

            Iterations                           5
            Presolve Time                     0.00
            Solution Time                     0.03


                    [1]                 x

                     1       0.0003034484
                     2       0.0000042439
                     3       0.9996923328
```

The SAS log shown in Figure 8.2 displays a brief summary of the problem being solved, followed by the iterations that are generated by the solver.

Figure 8.2 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 1 linear constraints (0 LE, 1 EQ, 0 GE, 0 range).
NOTE: The problem has 3 linear constraint coefficients.
NOTE: The problem has 1 nonlinear constraints (0 LE, 0 EQ, 1 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
                  Objective                            Optimality
         Iter          Value      Infeasibility             Error
            0     7.20000000                  0        6.40213404
            1     1.22115550         0.00042385        0.00500000
            2     1.00188693         0.00003290        0.00480263
            3     1.00275609         0.00002123        0.00005000
            4     1.00001702   0.0000000252254        0.00187172
            5     1.00001738   0.0000000250883   0.0000005000000
NOTE: Optimal.
NOTE: Objective = 1.000017384.
```

A Larger Optimization Problem

Consider the following larger optimization problem:

$$
\begin{aligned}
\text{minimize} \quad & f(x) = \sum_{i=1}^{1000} x_i y_i + \tfrac{1}{2} \sum_{j=1}^{5} z_j^2 \\
\text{subject to} \quad & x_k + y_k + \sum_{j=1}^{5} z_j = 5, \ \text{for } k = 1, 2, \ldots, 1000 \\
& \sum_{i=1}^{1000} (x_i + y_i) + \sum_{j=1}^{5} z_j \geq 6 \\
& -1 \leq x_i \leq 1, i = 1, 2, \ldots, 1000 \\
& -1 \leq y_i \leq 1, i = 1, 2, \ldots, 1000 \\
& 0 \leq z_i \leq 2, i = 1, 2, \ldots, 5
\end{aligned}
$$

The problem consists of a quadratic objective function, 1,000 linear equality constraints, and a linear inequality constraint. There are also 2,005 variables. The goal is to find a local minimum by using the ACTIVESET technique. This can be accomplished by issuing the following call to PROC OPTMODEL:

```
proc optmodel;
   number n = 1000;
   number b = 5;
   var x{1..n} >= -1 <= 1 init  0.99;
   var y{1..n} >= -1 <= 1 init -0.99;
   var z{1..b} >=  0 <= 2 init  0.5;
   minimize f = sum {i in 1..n} x[i] * y[i] + sum {j in 1..b} 0.5 * z[j]^2;
   con cons1{k in 1..n}: x[k] + y[k] + sum {j in 1..b} z[j] = b;
   con cons2: sum {i in 1..n} (x[i] + y[i]) + sum {j in 1..b} z[j] >= b + 1;
   solve with NLP / algorithm=activeset logfreq=10;
quit;
```

The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the local minimum. The summaries are shown in .

Figure 8.3 Problem Summary and Solution Summary

```
                    The OPTMODEL Procedure

                        Problem Summary

        Objective Sense            Minimization
        Objective Function                    f
        Objective Type               Quadratic

        Number of Variables                2005
        Bounded Above                         0
        Bounded Below                         0
        Bounded Below and Above            2005
        Free                                  0
        Fixed                                 0

        Number of Constraints              1001
        Linear LE (<=)                        0
        Linear EQ (=)                      1000
        Linear GE (>=)                        1
        Linear Range                          0


                   Performance Information

         Execution Mode        Single-Machine
         Number of Threads                   4


                     Solution Summary

         Solver                            NLP
         Algorithm                  Active Set
         Objective Function                  f
         Solution Status               Optimal
         Objective Value         -996.5000004

         Optimality Error         6.133411E-7
         Infeasibility           5.4803721E-7

         Iterations                          5
         Presolve Time                    0.03
         Solution Time                    0.81
```

The SAS log shown in Figure 8.4 displays a brief summary of the problem that is being solved, followed by the iterations that are generated by the solver.

Figure 8.4 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2005 variables (0 free, 0 fixed).
NOTE: The problem has 1001 linear constraints (0 LE, 1000 EQ, 1 GE, 0 range).
NOTE: The problem has 9005 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using 2 threads for nonlinear evaluation.
NOTE: The NLP solver is called.
NOTE: The Active Set algorithm is used.
                        Objective                          Optimality
          Iter              Value      Infeasibility            Error
             0       -979.47500000         3.50000000       0.50000000
             9       -996.49999988   0.0000002654239   0.0000000000116
NOTE: Optimal.
NOTE: Objective = -996.4999999.
```

An Optimization Problem with Many Local Minima

Consider the following optimization problem:

$$
\begin{aligned}
\text{minimize } f(x) \quad &= \quad e^{\sin(50x)} + \sin(60e^{y}) + \sin(70\sin(x)) + \sin(\sin(80y)) \\
&\quad - \sin(10(x + y)) + (x^2 + y^2)/4 \\
\text{subject to} \quad &\quad -1 \le x \le 1 \\
&\quad -1 \le y \le 1
\end{aligned}
$$

The objective function is highly nonlinear and contains many local minima. The NLP solver provides you with the option of searching the feasible region and identifying local minima of better quality. This is achieved by writing the following SAS program:

```
proc optmodel;
   var x >= -1 <= 1;
   var y >= -1 <= 1;
   min f = exp(sin(50*x)) + sin(60*exp(y)) + sin(70*sin(x)) + sin(sin(80*y))
           - sin(10*(x+y)) + (x^2+y^2)/4;
   solve with nlp / multistart seed=94245 msmaxstarts=30;
quit;
```

The MULTISTART option is specified, which directs the algorithm to start the local solver from many different starting points. The SAS log is shown in Figure 8.5.

Figure 8.5 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in single-machine mode.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 94245 is used.
                   Best        Local  Optimality     Infeasi-  Local  Local
        Start   Objective    Objective      Error      bility  Iters  Status
            1  -1.8567821   -1.8567821        5E-7           0      3  Optimal
            2  -2.9390086   -2.9390086        5E-7           0      3  Optimal
            3  -2.9390086   -2.6561783        5E-7           0      3  Optimal
            4  -3.3068686   -3.3068686        5E-7           0      3  Optimal
            5  -3.3068686   -2.2355863        5E-7           0      3  Optimal
            6  -3.3068686   -1.9014527  6.14439E-7  6.14439E-7      4  Optimal
            7  -3.3068686   -0.4808983        5E-7           0      3  Optimal
            8  -3.3068686   -1.5173191        5E-7           0      4  Optimal
            9  -3.3068686    -2.076195        5E-7           0      4  Optimal
           10  -3.3068686   -2.1930943        5E-7           0      5  Optimal
           11  -3.3068686   -0.7857749        5E-7           0      5  Optimal
           12  -3.3068686   -0.4033346        5E-7           0      3  Optimal
           13  -3.3068686   -0.5926643        5E-7           0      4  Optimal
           14  -3.3068686   -2.0402058        5E-7           0      4  Optimal
           15  -3.3068686   -2.9525781        5E-7           0      4  Optimal
           16 *-3.3068686   -1.3289057        5E-7           0      4  Optimal
           17  -3.3068686   -1.5650191        5E-7           0      7  Optimal
           18  -3.3068686    -1.404132        5E-7           0      3  Optimal
           19  -3.3068686   -2.4632393        5E-7           0      3  Optimal
           20  -3.3068686   -2.4541355        5E-7           0      4  Optimal
           21  -3.3068686   -2.5171432        5E-7           0      5  Optimal
           22  -3.3068686   -1.3559281  8.74345E-7  8.74345E-7      3  Optimal
           23  -3.3068686    -1.031811        5E-7           0      4  Optimal
           24  -3.3068686   -1.0823455        5E-7           0      5  Optimal
           25  -3.3068686    -2.387082        5E-7           0      4  Optimal
           26  -3.3068686   -0.6723829        5E-7           0      4  Optimal
           27  -3.3068686   -1.2265443        5E-7           0      4  Optimal
           28  -3.3068686   -0.6391886        5E-7           0      4  Optimal
           29  -3.3068686   -0.9393803        5E-7           0      4  Optimal
           30  -3.3068686   -0.5519164        5E-7           0      3  Optimal
NOTE: The Multistart algorithm generated 320 sample points.
NOTE: 30 distinct local optima were found.
NOTE: The best objective value found by local solver = -3.306868647.
NOTE: The solution found by local solver with objective = -3.306868647 was
      returned.
```

The SAS log presents additional information when the MULTISTART option is enabled. The first column counts the number of restarts of the local solver. The second column records the best local optimum that has been found so far, and the third through sixth columns record the local optimum to which the solver has converged. The final column records the status of the local solver at every iteration.

The SAS output is shown in Figure 8.6.

Figure 8.6 Problem Summary and Solution Summary

```
                    The OPTMODEL Procedure

                      Problem Summary

          Objective Sense              Minimization
          Objective Function                      f
          Objective Type                  Nonlinear

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

          Number of Constraints                   0


                   Performance Information

            Execution Mode        Single-Machine
            Number of Threads                   4


                      Solution Summary

          Solver                    Multistart NLP
          Algorithm                 Interior Point
          Objective Function                      f
          Solution Status                   Optimal
          Objective Value              -3.306868647

          Number of Starts                       30
          Number of Sample Points               320
          Number of Distinct Optima              30
          Random Seed Used                    94245
          Optimality Error                     5E-7
          Infeasibility                           0

          Presolve Time                        0.00
          Solution Time                        3.57
```

Syntax: NLP Solver

The following PROC OPTMODEL statement is available for the NLP solver:

SOLVE WITH NLP < */ options* > **;**

Functional Summary

Table 8.1 summarizes the options that can be used with the SOLVE WITH NLP statement.

Table 8.1   Options for the NLP Solver

Description	Option
Multistart Options	
Specifies the maximum range of values that each variable can take during the sampling process	MSBNDRANGE=
Specifies the tolerance for local optima to be considered distinct	MSDISTTOL=
Specifies the time limit in multistart mode	MSMAXTIME=
Specifies the number of starting points to be used by multistart	MSMAXSTARTS=
Specifies the seed used to generate random numbers	SEED=
Optimization Options	
Specifies the optimization technique	ALGORITHM=
Directs the local solver to start from multiple initial points	MULTISTART
Output Options	
Specifies the frequency of printing solution progress (local solvers)	LOGFREQ=
Specifies the amount of printing solution progress in multistart mode	MSLOGLEVEL=
Specifies the allowable types of output solution	SOLTYPE=
Solver Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the type of Hessian used by the solver	HESSTYPE=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the upper limit on the objective	OBJLIMIT=
Specifies the convergence tolerance	OPTTOL=
Specifies units of CPU time or real time	TIMETYPE=

NLP Solver Options

This section describes the options that are recognized by the NLP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the NLP solver is explicitly specified using a WITH clause.

Multistart Options

MSBNDRANGE=M

defines the range from which each variable can take values during the sampling process. This option affects only the sampling process that determines starting points for the local solver. It does not affect the bounds of the original nonlinear optimization problem. More specifically, if the ith variable x_i has lower and upper bounds ℓ_i and u_i respectively (that is, $\ell_i \leq x_i \leq u_i$), then an initial point is generated by a sampling process as follows:

For each sample point x, the ith coordinate x_i is generated so that the following bounds hold where x_i^0 is the default starting point or a specified starting point:

$$
\begin{array}{ll}
l_i \leq x_i \leq u_i & \text{if } l_i \text{ and } u_i \text{ are both finite} \\
l_i \leq x_i \leq l_i + M & \text{if only } l_i \text{ is finite} \\
u_i - M \leq x_i \leq u_i & \text{if only } u_i \text{ is finite} \\
x_i^0 - M/2 \leq x_i \leq x_i^0 + M/2 & \text{otherwise}
\end{array}
$$

This option is effective only when the MULTISTART option is specified. The default value is 200 in a shared-memory computing environment and 1,000 in a distributed computing environment.

MSDISTTOL=ϵ

defines the tolerance by which two optimal points are considered distinct. Optimal points are considered distinct if the Euclidean distance between them is at least ϵ. This option is effective only when the MULTISTART option is specified. The default is ϵ=1.0E–6.

MSMAXTIME=T

defines the maximum allowable time T (in seconds) for the NLP solver to locate the best local optimum in multistart mode. The value of the TIMETYPE= option determines the type of units used. The time specified by the MSMAXTIME= option is checked only once after the completion of the local solver. Since the local solver might be called many times, the maximum time specified for multistart is recommended to be greater than the maximum time specified for the local solver (that is, MSMAXTIME $\geq$ MAXTIME). This option is effective only when the MULTISTART option is specified. If you do not specify this option, the procedure does not stop based on the amount of time elapsed.

MSMAXSTARTS=N

defines the maximum number of starting points to be used for local optimization. That is, there will be no more than N local optimization calls in the multistart algorithm. You can specify N to be any nonnegative integer. When $N = 0$, the algorithm uses the default value of this option. In a shared memory computing environment, the default value is 100. In a distributed computing environment, the default value is a number proportional to the number of threads across all the grid nodes (usually more than 100). This option is effective only when the MULTISTART option is specified.

SEED=*N*

>   specifies a positive integer to be used as the seed for generating random number sequences. You can use this option to replicate results from different runs.

Optimization Options

ALGORITHM=*keyword*

TECHNIQUE=*keyword*

TECH=*keyword*

SOLVER=*keyword*

>   specifies the optimization technique to be used to solve the problem. The following *keywords* are valid:

>   **INTERIORPOINT**
>
>   >   uses a primal-dual interior point method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes a large number of inactive constraints.

>   **ACTIVESET**
>
>   >   uses a primal-dual active-set method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes only bound constraints or if the optimal active set can be quickly determined by the solver.

>   **CONCURRENT (experimental)**
>
>   >   runs the INTERIORPOINT and ACTIVESET techniques in parallel, with one thread using the INTERIORPOINT technique and the other thread using the ACTIVESET technique. The solution is returned by the first method that terminates.

>   The default is INTERIORPOINT.

MULTISTART

MS

>   enables multistart mode. In this mode, the local solver solves the problem from multiple starting points, possibly finding a better local minimum as a result. This option is disabled by default. For more information about multistart, see the section "Multistart" on page 317.

Output Options

LOGFREQ=*N*

PRINTFREQ=*N*

>   specifies how often the iterations are displayed in the SAS log. *N* should be an integer between zero and the largest four-byte, signed integer, which is $2^{31} - 1$. If $N \geq 1$, the solver prints only those iterations that are a multiple of *N*. If $N = 0$, no iteration is displayed in the log. The default value is 1.

MSLOGLEVEL=*number*

MSPRINTLEVEL=*number*

defines the amount of information displayed in the SAS log by the MULTISTART option. Table 8.2 describes the valid values of this option.

Table 8.2 Values for MSLOGLEVEL= Option

number	**Description**
0	Turns off all solver-related messages to SAS log
1	Displays multistart summary information when the algorithm terminates
2	Displays multistart iteration log and summary information when the algorithm terminates
3	Displays the same information as MSLOGLEVEL=2 and might display additional information

This option is effective only when the MULTISTART option is specified. The default is 2.

SOLTYPE=0 | 1

specifies whether the NLP solver should return only a solution that is locally optimal. If SOLTYPE=0, the solver returns a locally optimal solution, provided it locates one. If SOLTYPE=1, the solver returns the best feasible solution found, provided its objective value is better than that of the locally optimal solution found. The default is 1.

Solver Options

FEASTOL=ϵ

defines the feasible tolerance. The solver will exit if the constraint violation is less than FEASTOL and the scaled optimality conditions are less than OPTTOL. The default is ϵ=1E–6.

HESSTYPE=FULL | PRODUCT

specifies the type of Hessian to be used by the solver. The valid keywords for this option are FULL and PRODUCT. If HESSTYPE=FULL, the solver uses a full Hessian. If HESSTYPE=PRODUCT, the solver uses only Hessian-vector products, not the full Hessian. When the solver uses only Hessian-vector products to find a search direction, it usually uses much less memory, especially when the problem is large and the Hessian is not sparse. On the other hand, when the full Hessian is used, the algorithm can create a better preconditioner to solve the problem in less CPU time. The default is FULL.

MAXITER=*N*

specifies that the solver take at most *N* major iterations to determine an optimum of the NLP problem. The value of *N* is an integer between zero and the largest four-byte, signed integer, which is $2^{31} - 1$. A major iteration in NLP consists of finding a descent direction and a step size along which the next approximation of the optimum resides. The default is 5,000 iterations.

MAXTIME=*t*

    specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OBJLIMIT=*M*

    specifies an upper limit on the magnitude of the objective value. For a minimization problem, the algorithm terminates when the objective value becomes less than $-M$; for a maximization problem, the algorithm stops when the objective value exceeds M. The algorithm stopping implies that either the problem is unbounded or the algorithm diverges. If optimization were allowed to continue, numerical difficulty might be encountered. The default is M=1E+20. The minimum acceptable value of M is 1E+8. If the specified value of M is less than 1E+8, the value is reset to the default value 1E+20.

OPTTOL=*ϵ*

RELOPTTOL=*ϵ*

    defines the measure by which you can decide whether the current iterate is an acceptable approximation of a local minimum. The value of this option is a positive real number. The NLP solver determines that the current iterate is a local minimum when the norm of the scaled vector of the optimality conditions is less than ϵ and the true constraint violation is less than FEASTOL. The default is ϵ=1E–6.

TIMETYPE=*number* | *string*

    specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 8.3 describes the valid values of the TIMETYPE= option.

Table 8.3   Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The Optimization Statistics table, an output of PROC OPTMODEL if you specify the PRINTLEVEL=2 option in the PROC OPTMODEL statement, also includes the same time units for "Presolver Time" and "Solver Time." The other times (such as "Problem Generation Time") in the Optimization Statistics table are always CPU times.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement of the OPTMODEL procedure. See the section "PERFOR-MANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and NODES= options.

If you set the value of the NODES= option to 0 and the value of the NTHREADS= option to 1, the default value of the TIMETYPE= option is CPU. Otherwise the default value of the TIMETYPE= option is REAL.

Details: NLP Solver

This section presents a brief discussion about the algorithmic details of the NLP solver. First, the notation is defined. Next, an introduction to the fundamental ideas in constrained optimization is presented; the main point of the second section is to present the necessary and sufficient optimality conditions, which play a central role in all optimization algorithms. The section concludes with a general overview of primal-dual interior point and active-set algorithms for nonlinear optimization. A detailed treatment of the preceding topics can be found in Nocedal and Wright (1999), Wright (1997), and Forsgren, Gill, and Wright (2002).

Basic Definitions and Notation

The gradient of a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is the vector of all the first partial derivatives of f and is denoted by

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^{\mathrm{T}}$$

where the superscript T denotes the transpose of a vector.

The Hessian matrix of f, denoted by $\nabla^2 f(x)$, or simply by $H(x)$, is an $n \times n$ symmetric matrix whose (i, j) element is the second partial derivative of $f(x)$ with respect to x_i and x_j. That is, $H_{i,j}(x) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$.

Consider the vector function, $c : \mathbb{R}^n \mapsto \mathbb{R}^{p+q}$, whose first p elements are the equality constraint functions $h_i(x), i = 1, 2, \ldots, p$, and whose last q elements are the inequality constraint functions $g_i(x), i = 1, 2, \ldots, q$. That is,

$$c(x) = (h(x) : g(x))^{\mathrm{T}} = (h_1(x), \ldots, h_p(x) : g_1(x), \ldots, g_q(x))^{\mathrm{T}}$$

The $(p + q) \times n$ matrix whose ith row is the gradient of the ith element of $c(x)$ is called the Jacobian matrix of $c(x)$ (or simply the Jacobian of the NLP problem) and is denoted by $J(x)$. You can also use $J_h(x)$ to denote the $p \times n$ Jacobian matrix of the equality constraints and use $J_g(x)$ to denote the $q \times n$ Jacobian matrix of the inequality constraints. It is easy to see that

Constrained Optimization

A function that plays a pivotal role in establishing conditions that characterize a local minimum of an NLP problem is the Lagrangian function $\mathcal{L}(x, y, z)$, which is defined as

$$\mathcal{L}(x, y, z) = f(x) - \sum_{i \in \mathcal{E}} y_i h_i(x) - \sum_{i \in \mathcal{I}} z_i g_i(x)$$

Note that the Lagrangian function can be seen as a linear combination of the objective and constraint functions. The coefficients of the constraints, $y_i, i \in \mathcal{E}$, and $z_i, i \in \mathcal{I}$, are called the Lagrange multipliers or dual variables. At a feasible point $\hat{x}$, an inequality constraint is called active if it is satisfied as an equality—that is, $g_i(\hat{x}) = 0$. The set of active constraints at a feasible point $\hat{x}$ is then defined as the union of the index set of the equality constraints, $\mathcal{E}$, and the indices of those inequality constraints that are active at $\hat{x}$; that is,

$$\mathcal{A}(\hat{x}) = \mathcal{E} \cup \{i \in \mathcal{I} : g_i(\hat{x}) = 0\}$$

An important condition that is assumed to hold in the majority of optimization algorithms is the so-called linear independence constraint qualification (LICQ). The LICQ states that at any feasible point $\hat{x}$, the gradients of all the active constraints are linearly independent. The main purpose of the LICQ is to ensure that the set of constraints is well-defined in a way that there are no redundant constraints or in a way that there are no constraints defined such that their gradients are always equal to zero.

The First-Order Necessary Optimality Conditions

If x^* is a local minimum of the NLP problem and the LICQ holds at x^*, then there are vectors of Lagrange multipliers y^* and z^*, with components $y_i^*, i \in \mathcal{E}$, and $z_i^*, i \in \mathcal{I}$, respectively, such that the following conditions are satisfied:

$$
\begin{aligned}
\nabla_x \mathcal{L}(x^*, y^*, z^*) &= 0 \\
h_i(x^*) &= 0, \quad i \in \mathcal{E} \\
g_i(x^*) &\geq 0, \quad i \in \mathcal{I} \\
z_i^* &\geq 0, \quad i \in \mathcal{I} \\
z_i^* g_i(x^*) &= 0, \quad i \in \mathcal{I}
\end{aligned}
$$

where $\nabla_x \mathcal{L}(x^*, y^*, z^*)$ is the gradient of the Lagrangian function with respect to x, defined as

$$
\nabla_x \mathcal{L}(x^*, y^*, z^*) = \nabla f(x) - \sum_{i \in \mathcal{E}} y_i \nabla h_i(x) - \sum_{i \in \mathcal{I}} z_i \nabla g_i(x)
$$

The preceding conditions are often called the *Karush-Kuhn-Tucker (KKT) conditions*. The last group of equations ($z_i g_i(x) = 0, i \in I$) is called the complementarity condition. Its main aim is to try to force the Lagrange multipliers, z_i^*, of the inactive inequalities (that is, those inequalities with $g_i(x^*) > 0$) to zero.

The KKT conditions describe the way the first derivatives of the objective and constraints are related at a local minimum x^*. However, they are not enough to fully characterize a local minimum. The second-order optimality conditions attempt to fulfill this aim by examining the curvature of the Hessian matrix of the Lagrangian function at a point that satisfies the KKT conditions.

The Second-Order Necessary Optimality Condition

Let x^* be a local minimum of the NLP problem, and let y^* and z^* be the corresponding Lagrange multipliers that satisfy the first-order optimality conditions. Then $d^{\mathrm{T}} \nabla_x^2 \mathcal{L}(x^*, y^*, z^*) d \geq 0$ for all nonzero vectors d that satisfy the following conditions:

1. $\nabla h_i^{\mathrm{T}}(x^*) d = 0, \forall i \in \mathcal{E}$

2. $\nabla g_i^{\mathrm{T}}(x^*) d = 0, \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z_i^* > 0$

3. $\nabla g_i^{\mathrm{T}}(x^*) d \geq 0, \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z_i^* = 0$

The second-order necessary optimality condition states that, at a local minimum, the curvature of the Lagrangian function along the directions that satisfy the preceding conditions must be nonnegative.

Interior Point Algorithm

Primal-dual interior point methods can be classified into two categories: feasible and infeasible. The first category requires that the starting point and all subsequent iterations of the algorithm strictly satisfy all the inequality constraints. The second category relaxes those requirements and allows the iterations to violate some or all of the inequality constraints during the course of the minimization procedure. The NLP solver implements an infeasible algorithm; this section concentrates on that type of algorithm.

To make the notation less cluttered and the fundamentals of interior point methods easier to understand, consider without loss of generality the following simpler NLP problem:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \geq 0, i \in \mathcal{I} = \{1, 2, \ldots, q\} \end{aligned}$$

Note that the equality and bound constraints have been omitted from the preceding problem. Initially, slack variables are added to the inequality constraints, giving rise to the problem

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) - s_i = 0, i \in \mathcal{I} \\ & s \geq 0 \end{aligned}$$

where $s = (s_1, \ldots, s_q)^{\mathrm{T}}$ is the vector of slack variables, which are required to be nonnegative. Next, all the nonnegativity constraints on the slack variables are eliminated by being incorporated into the objective function, by means of a logarithmic function. This gives rise to the equality-constrained NLP problem

$$\begin{aligned} \text{minimize} \quad & B(x, s) = f(x) - \mu \sum_{i \in \mathcal{I}} \ln(s_i) \\ \text{subject to} \quad & g_i(x) - s_i = 0, i \in \mathcal{I} \end{aligned}$$

where μ is a positive parameter. The nonnegativity constraints on the slack variables are implicitly enforced by the logarithmic functions, since the logarithmic function prohibits s from taking zero or negative values.

Next, the equality constraints can be absorbed by using a quadratic penalty function to obtain the following:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \|g(x) - s\|_2^2 - \mu \sum_{i \in \mathcal{I}} \ln(s_i)$$

The preceding unconstrained problem is often called the *penalty-barrier subproblem*. Depending on the size of the parameter μ, a local minimum of the barrier problem provides an approximation to the local minimum of the original NLP problem. The smaller the size of μ, the better the approximation becomes. Infeasible primal-dual interior point algorithms repeatedly solve the penalty-barrier problem for different values of μ that progressively go to zero, in order to get as close as possible to a local minimum of the original NLP problem.

An unconstrained minimizer of the penalty-barrier problem must satisfy the equations

$$\begin{aligned} \nabla f(x) - J(x)^{\mathrm{T}} z &= 0 \\ z - \mu S^{-1} e &= 0 \end{aligned}$$

where $z = -(g(x) - s)/\mu$, $J(x)$ is the Jacobian matrix of the vector function $g(x)$, S is the diagonal matrix whose elements are the elements of the vector s (that is, $S = \mathrm{diag}\{s_1, \ldots, s_q\}$), and e is a vector of all ones.

Multiplying the second equation by S and adding the definition of z as a third equation produces the following equivalent nonlinear system:

$$F^\mu(x, s, z) = \begin{pmatrix} \nabla f(x) - J(x)^{\mathrm{T}} z \\ Sz - e \\ g(x) - s + \mu z \end{pmatrix} = 0$$

Note that if $\mu = 0$, the preceding conditions represent the optimality conditions of the original optimization problem, after adding slack variables. One of the main aims of the algorithm is to gradually reduce the value of μ to zero, so that it converges to a local optimum of the original NLP problem. The rate by which μ approaches zero affects the overall efficiency of the algorithm. Algorithms that treat z as an additional variable are considered primal-dual, while those that enforce the definition of $z = -(g(x) - s)/\mu$ at each iteration are consider purely primal approaches.

At iteration k, the infeasible primal-dual interior point algorithm approximately solves the preceding system by using Newton's method. The Newton system is

$$\begin{bmatrix} H_{\mathcal{L}}(x^k, z^k) & 0 & -J(x^k)^{\mathrm{T}} \\ 0 & Z^k & S^k \\ J(x^k) & -I & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta s^k \\ \Delta z^k \end{bmatrix} = - \begin{bmatrix} \nabla_x f(x^k) - J(x^k)^{\mathrm{T}} z \\ -\mu e + S^k z^k \\ g(x^k) - s^k + \mu z^k \end{bmatrix}$$

where $H_{\mathcal{L}}$ is the Hessian matrix of the Lagrangian function $\mathcal{L}(x, z) = f(x) - z^{\mathrm{T}} g(x)$ of the original NLP problem; that is,

$$H_{\mathcal{L}}(x, z) = \nabla^2 f(x) - \sum_{i \in \mathcal{I}} z_i \nabla^2 g_i(x)$$

The solution $(\Delta x^k, \Delta s^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration (x^k, s^k, z^k) to the next,

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

where α is the step length along the Newton direction. The step length is determined through a line-search procedure that ensures sufficient decrease of a merit function based on the augmented Lagrangian function of the barrier problem. The role of the merit function and the line-search procedure is to ensure that the objective and the infeasibility reduce sufficiently at every iteration and that the iterations approach a local minimum of the original NLP problem.

Active-Set Method

Active-set methods differ from interior point methods in that no barrier term is used to ensure that the algorithm remains interior with respect to the inequality constraints. Instead, attempts are made to learn the true active set. For simplicity, use the same initial slack formulation used by the interior point method description,

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) - s_i = 0, i \in \mathcal{I} \\ & s \geq 0 \end{aligned}$$

where $s = (s_1, \ldots, s_q)^{\mathrm{T}}$ is the vector of slack variables, which are required to be nonnegative. Begin by absorbing the equality constraints as before into a penalty function, but keep the slack bound constraints explicitly:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \|g(x) - s\|_2^2$$

$$\text{subject to} \quad s \geq 0$$

where μ is a positive parameter. Given a solution pair $(x(\mu), s(\mu))$ for the preceding problem, you can define the active-set projection matrix P as follows:

$$P_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } s_i(\mu) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then $(x(\mu), s(\mu))$ is also a solution of the equality constraint subproblem:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \|g(x) - s\|_2^2$$

$$\text{subject to} \quad Ps = 0.$$

The minimizer of the preceding subproblem must be a stationary point of the Lagrangian function

$$\mathcal{L}^{\mu}(x, s, z) = f(x) + \frac{1}{2\mu} \|g(x) - s\|_2^2 - z^T Ps$$

which gives the optimality equations

$$\begin{aligned} \nabla_x \mathcal{L}^{\mu}(x, s, z) &= \nabla f(x) - J(x)^T y &= 0 \\ \nabla_s \mathcal{L}^{\mu}(x, s, z) &= y - P^T z &= 0 \\ &= Ps &= 0 \end{aligned}$$

where $y = -(g(x) - s)/\mu$. Using the second equation, you can simplify the preceding equations to get the following optimality conditions for the bound-constrained penalty subproblem:

$$\begin{aligned} \nabla f(x) - J(x)^T P^T z &= 0 \\ P(g(x) - s) + \mu z &= 0 \\ Ps &= 0 \end{aligned}$$

Using the third equation directly, you can reduce the system further to

$$\begin{aligned} \nabla f(x) - J(x)^T P^T z &= 0 \\ P g(x) + \mu z &= 0 \end{aligned}$$

At iteration k, the primal-dual active-set algorithm approximately solves the preceding system by using Newton's method. The Newton system is

$$\begin{bmatrix} H_{\mathcal{L}}(x^k, z^k) & -J_{\mathcal{A}}^{\mathrm{T}} \\ J_{\mathcal{A}} & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta z^k \end{bmatrix} = - \begin{bmatrix} \nabla_x f(x^k) - J_{\mathcal{A}}^{\mathrm{T}} z \\ Pg(x^k) + \mu z^k \end{bmatrix}$$

where $J_{\mathcal{A}} = PJ(x^k)$ and $H_{\mathcal{L}}$ denotes the Hessian of the Lagrangian function $f(x) - z^{\mathrm{T}} Pg(x)$. The solution $(\Delta x^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration (x^k, s^k, z^k) to the next,

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

where α is the step length along the Newton direction. The corresponding slack variable update s^{k+1} is defined as the solution to the following subproblem whose solution can be computed analytically:

$$\text{minimize} \quad \mathcal{M}(x^{k+1}, s) = f(x) + \frac{1}{2\mu}\|g(x^{k+1}) - s\|_2^2$$
$$\text{subject to} \quad s \geq 0$$

The step length α is then determined in a similar manner to the preceding interior point approach. At each iteration, the definition of the active-set projection matrix P is updated with respect to the new value of the constraint function $g(x^{k+1})$. For large-scale NLP, the computational bottleneck typically arises in seeking to solve the Newton system. Thus active-set methods can achieve substantial computational savings when the size of $J_\mathcal{A}$ is much smaller than $J(x)$; however, convergence can be slow if the active-set estimate changes combinatorially. Further, the active-set algorithm is often the superior algorithm when only bound constraints are present. In practice, both the interior point and active-set approach incorporate more sophisticated merit functions than those described in the preceding sections; however, their description is beyond the scope of this document. See Gill and Robinson (2010) for further reading.

Multistart

Frequently, nonlinear optimization problems contain many local minima because the objective or the constraints are nonconvex functions. The quality of different local minima is measured by the objective value achieved at those points. For example, if x_1^* and x_2^* are two distinct local minima and $f(x_1^*) \leq f(x_2^*)$, then x_1^* is said to be of better quality than x_2^*. The NLP solver provides a mechanism that can locate local minima of better quality by starting the local solver multiple times from different initial points. By doing so, the local solver can converge to different local minima. The local minimum with the lowest objective value is then reported back to the user.

The multistart feature consists of two phases. In the first phase, the entire feasible region is explored by generating sample points from a uniform distribution. The aim of this phase is to place at least one sample point in the region of attraction of every local minimum. Here the region of attraction of a local minimum is defined as the set of feasible points that, when used as starting points, enable a local solver to converge to that local minimum.

During the second phase, a subset of the sample points generated in the first phase is chosen by applying a clustering technique. The goal of the clustering technique is to group the initial sample points around the local minima and allow only a single local optimization to start from each cluster or group. The clustering technique aims to reduce computation time by sparing the work of unnecessarily starting multiple local optimizations within the region of attraction of the same local minimum.

The number of starting points is critical to the time spent by the solver to find a good local minimum. You can specify the maximum number of starting points by using the MSMAXSTARTS= option. If this option is not specified, the solver determines the minimum number of starting points that can provide reasonable evidence that a good local minimum will be found.

Many optimization problems contain variables with infinite upper or lower bounds. These variables can cause the sampling procedure to generate points that are not useful for locating different local minima. The efficiency of the sampling procedure can be increased by reducing the range of these variables by using the MSBNDRANGE= option. This option forces the sampling procedure to generate points that are in a smaller interval, thereby increasing the efficiency of the solver to converge to a local optimum.

The multistart feature is compatible with the PERFORMANCE statement in the OPTMODEL procedure. See Chapter 4, "Shared Concepts and Topics," for more information about the PERFORMANCE statement. The multistart feature currently supports only the DETERMINISTIC value for the PARALLELMODE= option in the PERFORMANCE statement. To ensure reproducible results, specify a nonzero value for the SEED= option.

Accessing the Starting Point That Leads to the Best Local Optimum

The starting point that leads to the best local optimum can be accessed by using the .msinit suffix in PROC OPTMODEL. In some cases, the knowledge of that starting point might be useful. For example, you can run the local solver again but this time providing as initial point the one that is stored in .msinit. This way the multistart explores a different part of the feasible region and might discover a local optimum of better quality than those found in previous runs. The use of the suffix .msinit is demonstrated in Example 8.5. For more information about suffixes in PROC OPTMODEL, see "Suffixes" on page 131 in Chapter 5, "The OPTMODEL Procedure."

Iteration Log for the Local Solver

The iteration log for the local solver provides detailed information about progress towards a locally optimal solution. This log appears when multistart mode is disabled.

The following information is displayed in the log:

Iter	indicates the iteration number.
Objective Value	indicates the objective function value.
Infeasibility	indicates the maximum value out of all constraint violations.
Optimality Error	indicates the relative optimality error (see the section "Solver Termination Criterion" on page 319).

Iteration Log for Multistart

When the MULTISTART option is enabled, the iteration log differs from that of the local solver. More specifically, when a value of 2 is specified for the MSLOGLEVEL= option, the following information is displayed in the log:

Start	indicates the index number of each local optimization run. The following indicators can appear beside this number to provide additional information about the run:

	*	indicates the local optimization started from a user-supplied point.
	r	indicates the local optimization converged to a previously found optimal solution.
	R	indicates the local optimization started from a user-supplied point and converged to a previously found optimal solution.

Best Objective	indicates the value of the objective function at the best local solution found so far.

Local Objective   indicates the value of the objective function obtained at the solution returned by the local solver.

Infeasibility   indicates the infeasibility error at the solution returned by the local solver.

Optimality Error   indicates the optimality error at the solution returned by the local solver.

Local Iters   indicates the number of iterations taken by the local solver.

Local Status   indicates the solution status of the local solver. Several different values can appear in this column:

OPTIMAL	indicates that the local solver found a locally optimal solution.
BESTFEASIBLE	indicates that the local solver returned the best feasible point found. See the SOLTYPE= option for more information.
INFEASIBLE	indicates that the local solver converged to a point that might be infeasible.
LOCALINFEAS	indicates that the local solver converged to a point of minimal local infeasibility.
UNBOUNDED	indicates that the local solver determined that the problem is unbounded.
ITERLIMIT	indicates that the local solver reached the maximum number of iterations and could not find a locally optimal solution.
TIMELIMIT	indicates that the local solver reached the maximum allowable time and could not find a locally optimal solution.
ABORTED	indicates that the local solver terminated due to a user interrupt.
FUNEVALERR	indicates that the local solver encountered a function evaluation error.
NUMERICERR	indicates that the local solver encountered a numerical error other than a function evaluation error.
INTERNALERR	indicates that the local solver encountered a solver system error.
OUTMEMORY	indicates that the local solver ran out of memory.
FAILED	indicates a general failure of the local solver in the absence of any other error.

Solver Termination Criterion

Because badly scaled problems can lead to slow convergence, the NLP solver dynamically rescales both the objective and constraint functions adaptively as needed. The optimality conditions are always stated with respect to the rescaled NLP. However, because typically you are most interested in the constraint violation of the original NLP, and not the internal scaled variant, you always work with respect to the true constraint violation. Thus, the solver terminates when both of the following conditions are true:

- The norm of the optimality conditions of the scaled NLP is less than the user-defined or default tolerance (OPTTOL= option).

- The norm of the constraint violation of the original NLP is less than the user-defined feasibility tolerance (FEASTOL= option).

More specifically, if

$$F(x, s, z) = (\nabla_x f(x) - J(x)^{\mathrm{T}} z, \quad Sz, \quad g(x) - s)^{\mathrm{T}}$$

is the vector of the optimality conditions of the rescaled NLP problem, then the solver terminates when

$$\| F(x, s, z) \| \leq \mathrm{OPTTOL}(1 + \|(x, s)\|)$$

and the maximum constraint violation is less than FEASTOL.

Solver Termination Messages

Upon termination, the solver produces the following messages in the log:

Optimal
  The solver has successfully found a local solution to the optimization problem.

Conditionally optimal solution found
  The solver is sufficiently close to a local solution, but it has difficulty in completely satisfying the user-defined optimality tolerance. This can happen when the line search finds very small steps that result in very slight progress of the algorithm. It can also happen when the prespecified tolerance is too strict for the optimization problem at hand.

Maximum number of iterations reached
  The solver could not find a local optimum in the prespecified number of iterations.

Maximum specified time reached
  The solver could not find a local optimum in the prespecified maximum real time for the optimization process.

Did not converge
  The solver could not satisfy the optimality conditions and failed to converge.

Problem might be unbounded
  The objective function takes arbitrarily large values, and the optimality conditions fail to be satisfied. This can happen when the problem is unconstrained or when the problem is constrained and the feasible region is not bounded.

Problem might be infeasible
  The solver cannot identify a point in the feasible region.

Problem is infeasible
  The solver detects that the problem is infeasible.

Out of memory
>   The problem is so large that the solver requires more memory to solve the problem.

Problem solved by the OPTMODEL presolver
>   The problem was solved by the OPTMODEL presolver.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. After the NLP solver is called, the various terms of the variable are interpreted as follows:

STATUS
>   indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	The use of syntax is incorrect.
DATA_ERROR	The input data are inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem in reading or writing of data has occurred.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, has occurred.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM
>   indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

IP	The interior point algorithm produced the solution data.
AS	The active-set algorithm produced the solution data.

>   When running algorithms concurrently (ALGORITHM=CONCURRENT), this term indicates which algorithm was the first to terminate.

SOLUTION_STATUS
>   indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
BEST_FEASIBLE	The solution returned is the best feasible solution. This solution status indicates that the algorithm has converged to a local optimum but a feasible (not locally optimal) solution with a better objective value has been found and hence is returned.

INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem might be unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is not supported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations has been reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE

indicates the objective value that is obtained by the solver at termination.

NUMSTARTS

indicates the number of starting points. This term appears only in multistart mode.

SAMPLE_POINTS

indicates the number of points that are evaluated in the sampling phase. This term appears only in multistart mode.

DISTINCT_OPTIMA

indicates the number of distinct local optima that the solver finds. This term appears only in multistart mode.

SEED

indicates the seed value that is used to initialize the solver. This term appears only in multistart mode.

INFEASIBILITY

indicates the level of infeasibility of the constraints at the solution.

OPTIMALITY_ERROR

indicates the norm of the optimality conditions at the solution. See the section "Solver Termination Criterion" on page 319 for details.

ITERATIONS

indicates the number of iterations required to solve the problem.

PRESOLVE_TIME

indicates the real time taken for preprocessing (seconds).

SOLUTION_TIME

indicates the real time taken by the solver to perform iterations for solving the problem (seconds).

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: NLP Solver

Example 8.1: Solving Highly Nonlinear Optimization Problems

This example demonstrates the use of the NLP solver to solve the following highly nonlinear optimization problem, which appears in Hock and Schittkowski (1981):

$$
\begin{aligned}
\text{minimize} \quad & f(x) = 0.4(x_1/x_7)^{0.67} + 0.4(x_2/x_8)^{0.67} + 10 - x_1 - x_2 \\
\text{subject to} \quad & 1 - 0.0588x_5x_7 - 0.1x_1 \geq 0 \\
& 1 - 0.0588x_6x_8 - 0.1x_1 - 0.1x_2 \geq 0 \\
& 1 - 4x_3/x_5 - 2/(x_3^{0.71}x_5) - 0.0588x_7/x_3^{1.3} \geq 0 \\
& 1 - 4x_4/x_6 - 2/(x_4^{0.71}x_6) - 0.0588x_8/x_4^{1.3} \geq 0 \\
& 0.1 \leq f(x) \leq 4.2 \\
& 0.1 \leq x_i \leq 10, i = 1, 2, \ldots, 8
\end{aligned}
$$

The initial point used is $x^0 = (6, 3, 0.4, 0.2, 6, 6, 1, 0.5)$. You can call the NLP solver within PROC OPTMODEL to solve the problem by writing the following SAS statements:

```
proc optmodel;
   var x{1..8} >= 0.1 <= 10;

   min f = 0.4*(x[1]/x[7])^0.67 + 0.4*(x[2]/x[8])^0.67 + 10 - x[1] - x[2];

   con c1: 1 - 0.0588*x[5]*x[7] - 0.1*x[1] >= 0;
   con c2: 1 - 0.0588*x[6]*x[8] - 0.1*x[1] - 0.1*x[2] >= 0;
   con c3: 1 - 4*x[3]/x[5] - 2/(x[3]^0.71*x[5]) - 0.0588*x[7]/x[3]^1.3 >= 0;
   con c4: 1 - 4*x[4]/x[6] - 2/(x[4]^0.71*x[6]) - 0.0588*x[8]/x[4]^1.3 >= 0;
   con c5: 0.1 <= f <= 4.2;

   /* starting point */
   x[1] = 6;
   x[2] = 3;
   x[3] = 0.4;
   x[4] = 0.2;
   x[5] = 6;
   x[6] = 6;
   x[7] = 1;
   x[8] = 0.5;

   solve with nlp / algorithm=activeset;
   print x;
quit;
```

The summaries and the solution are shown in Output 8.1.1.

Output 8.1.1 Summaries and the Optimal Solution

```
                     The OPTMODEL Procedure

                        Problem Summary

      Objective Sense            Minimization
      Objective Function                    f
      Objective Type               Nonlinear

      Number of Variables                   8
      Bounded Above                         0
      Bounded Below                         0
      Bounded Below and Above               8
      Free                                  0
      Fixed                                 0

      Number of Constraints                 5
      Linear LE  (<=)                       0
      Linear EQ  (=)                        0
      Linear GE  (>=)                       0
      Linear Range                          0
      Nonlinear LE (<=)                     0
      Nonlinear EQ (=)                      0
      Nonlinear GE (>=)                     4
      Nonlinear Range                       1


                   Performance Information

       Execution Mode        Single-Machine
       Number of Threads                   4


                     Solution Summary

        Solver                          NLP
        Algorithm                Active Set
        Objective Function                f
        Solution Status             Optimal
        Objective Value       3.9511634753

        Optimality Error        1.691012E-9
        Infeasibility          7.8982136E-9

        Iterations                       23
        Presolve Time                  0.00
        Solution Time                  0.08
```

Output 8.1.1 *continued*

	[1]	x
	1	6.46511
	2	2.23271
	3	0.66740
	4	0.59576
	5	5.93268
	6	5.52723
	7	1.01332
	8	0.40067

Example 8.2: Solving Unconstrained and Bound-Constrained Optimization Problems

Although the NLP techniques are suited for solving generally constrained nonlinear optimization problems, these techniques can also be used to solve unconstrained and bound-constrained problems efficiently. This example considers the relatively large nonlinear optimization problems

$$\text{minimize } f(x) = \sum_{i=1}^{n-1}(-4x_i + 3.0) + \sum_{i=1}^{n-1}(x_i^2 + x_n^2)^2$$

and

$$\text{minimize} \quad f(x) = \sum_{i=1}^{n-1}\cos(-.5x_{i+1} - x_i^2)$$
$$\text{subject to} \quad 1 \le x_i \le 2, \ i = 1, \dots, n$$

with $n = 100,000$. These problems are unconstrained and bound-constrained, respectively.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section "Memory Limit" on page 30 for more information.

To solve the first problem, you can write the following statements:

```
proc optmodel;
   number N=100000;
   var x{1..N} init 1.0;

   minimize f = sum {i in 1..N - 1} (-4 * x[i] + 3.0)  +
                sum {i in 1..N - 1} (x[i]^2 + x[N]^2)^2;

   solve with nlp;
quit;
```

The problem and solution summaries are shown in Output 8.2.1.

Output 8.2.1 Problem Summary and Solution Summary

```
                        The OPTMODEL Procedure

                           Problem Summary

              Objective Sense            Minimization
              Objective Function                    f
              Objective Type               Nonlinear

              Number of Variables             100000
              Bounded Above                        0
              Bounded Below                        0
              Bounded Below and Above              0
              Free                            100000
              Fixed                                0

              Number of Constraints                0


                       Performance Information

                Execution Mode       Single-Machine
                Number of Threads                 4


                         Solution Summary

                Solver                          NLP
                Algorithm            Interior Point
                Objective Function                f
                Solution Status             Optimal
                Objective Value                   0

                Optimality Error        1.007903E-14
                Infeasibility                     0

                Iterations                       16
                Presolve Time                  0.02
                Solution Time                 15.03
```

To solve the second problem, you can write the following statements (here the active-set method is specifically selected):

```
proc optmodel;
   number N=100000;
   var x{1..N} >= 1 <= 2;

   minimize f = sum {i in 1..N − 1} cos(−0.5*x[i+1] − x[i]^2);

   solve with nlp / algorithm=activeset;
quit;
```

The problem and solution summaries are shown in Output 8.2.2.

Output 8.2.2 Problem Summary and Solution Summary

```
                    The OPTMODEL Procedure

                       Problem Summary

        Objective Sense              Minimization
        Objective Function                      f
        Objective Type                  Nonlinear

        Number of Variables               100000
        Bounded Above                          0
        Bounded Below                          0
        Bounded Below and Above           100000
        Free                                   0
        Fixed                                  0

        Number of Constraints                  0


                   Performance Information

         Execution Mode       Single-Machine
         Number of Threads                  4


                     Solution Summary

          Solver                        NLP
          Algorithm               Active Set
          Objective Function               f
          Solution Status            Optimal
          Objective Value             -99999

          Optimality Error      1.437473E-12
          Infeasibility                     0

          Iterations                        8
          Presolve Time                  0.02
          Solution Time                 18.38
```

Example 8.3: Solving NLP Problems with Range Constraints

Some constraints have both lower and upper bounds (that is, $a \leq g(x) \leq b$). These constraints are called *range constraints*. The NLP solver can handle range constraints in an efficient way. Consider the following NLP problem, taken from Hock and Schittkowski (1981),

$$\text{minimize} \quad f(x) = 5.35(x_3)^2 + 0.83x_1x_5 + 37.29x_1 - 40792.141$$
$$\text{subject to} \quad 0 \leq a_1 + a_2x_2x_5 + a_3x_1x_4 - a_4x_3x_5 \leq 92$$
$$0 \leq a_5 + a_6x_2x_5 + a_7x_1x_2 + a_8x_3^2 - 90 \leq 20$$
$$0 \leq a_9 + a_{10}x_3x_5 + a_{11}x_1x_3 + a_{12}x_3x_4 - 20 \leq 5$$
$$78 \leq x_1 \leq 102$$
$$33 \leq x_2 \leq 45$$
$$27 \leq x_i \leq 45, \quad i = 3, 4, 5$$

where the values of the parameters $a_i, i = 1, 2, \ldots, 12$, are shown in Table 8.4.

Table 8.4 Data for Example 3

i	a_i	i	a_i	i	a_i
1	85.334407	5	80.51249	9	9.300961
2	0.0056858	6	0.0071317	10	0.0047026
3	0.0006262	7	0.0029955	11	0.0012547
4	0.0022053	8	0.0021813	12	0.0019085

The initial point used is $x^0 = (78, 33, 27, 27, 27)$. You can call the NLP solver within PROC OPTMODEL to solve this problem by writing the following statements:

```
proc optmodel;
   number l {1..5} = [78 33 27 27 27];
   number u {1..5} = [102 45 45 45 45];

   number a {1..12} =
      [85.334407 0.0056858 0.0006262 0.0022053
       80.51249 0.0071317 0.0029955 0.0021813
       9.300961 0.0047026 0.0012547 0.0019085];

   var x {j in 1..5} >= l[j] <= u[j];

   minimize f = 5.35*x[3]^2 + 0.83*x[1]*x[5] + 37.29*x[1]
                - 40792.141;

   con constr1:
      0 <= a[1] + a[2]*x[2]*x[5] + a[3]*x[1]*x[4] -
         a[4]*x[3]*x[5] <= 92;
   con constr2:
      0 <= a[5] + a[6]*x[2]*x[5] + a[7]*x[1]*x[2] +
         a[8]*x[3]^2 - 90 <= 20;
   con constr3:
      0 <= a[9] + a[10]*x[3]*x[5] + a[11]*x[1]*x[3] +
         a[12]*x[3]*x[4] - 20 <= 5;

   x[1] = 78;
   x[2] = 33;
   x[3] = 27;
   x[4] = 27;
   x[5] = 27;
```

```
    solve with nlp / algorithm=activeset;
    print x;
quit;
```

The summaries and solution are shown in Output 8.3.1.

Output 8.3.1 Summaries and the Optimal Solution

```
                        The OPTMODEL Procedure

                          Problem Summary

            Objective Sense              Minimization
            Objective Function                      f
            Objective Type                  Quadratic

            Number of Variables                     5
            Bounded Above                           0
            Bounded Below                           0
            Bounded Below and Above                 5
            Free                                    0
            Fixed                                   0

            Number of Constraints                   3
            Linear LE  (<=)                         0
            Linear EQ  (=)                          0
            Linear GE  (>=)                         0
            Linear Range                            0
            Nonlinear LE  (<=)                      0
            Nonlinear EQ  (=)                       0
            Nonlinear GE  (>=)                      0
            Nonlinear Range                         3


                      Performance Information

              Execution Mode       Single-Machine
              Number of Threads                 4


                          Solution Summary

              Solver                          NLP
              Algorithm                Active Set
              Objective Function                f
              Solution Status             Optimal
              Objective Value        -30689.16932

              Optimality Error       2.2010098E-8
              Infeasibility                     0

              Iterations                       20
              Presolve Time                  0.00
              Solution Time                  0.04
```

Output 8.3.1 *continued*

```
                          [1]          x

                           1        78.000
                           2        33.000
                           3        29.995
                           4        45.000
                           5        36.776
```

Example 8.4: Solving Large-Scale NLP Problems

The following example is a selected large-scale problem from the CUTEr test set (Gould, Orban, and Toint, Ph. L. 2003) that has 20,400 variables, 20,400 lower bounds, and 9,996 linear equality constraints. This problem was selected to provide an idea of the size of problem that the NLP solver is capable of solving. In general, the maximum size of nonlinear optimization problems that can be solved with the NLP solver is controlled less by the number of variables and more by the density of the first and second derivatives of the nonlinear objective and constraint functions.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section "Memory Limit" on page 30 for more information.

```
proc optmodel;
   num nx = 100;
   num ny = 100;

   var x {1..nx, 0..ny+1} >= 0;
   var y {0..nx+1, 1..ny} >= 0;

   min f = (
        sum {i in 1..nx-1, j in 1..ny-1} (x[i,j] - 1)^2
      + sum {i in 1..nx-1, j in 1..ny-1} (y[i,j] - 1)^2
      + sum {i in 1..nx-1} (x[i,ny] - 1)^2
      + sum {j in 1..ny-1} (y[nx,j] - 1)^2
      ) / 2;

   con con1 {i in 2..nx-1, j in 2..ny-1}:
      (x[i,j] - x[i-1,j]) + (y[i,j] - y[i,j-1]) = 1;
   con con2 {i in 2..nx-1}:
      x[i,0] + (x[i,1] - x[i-1,1]) + y[i,1] = 1;
   con con3 {i in 2..nx-1}:
      x[i,ny+1] + (x[i,ny] - x[i-1,ny]) - y[i,ny-1] = 1;
   con con4 {j in 2..ny-1}:
      y[0,j] + (y[1,j] - y[1,j-1]) + x[1,j] = 1;
   con con5 {j in 2..ny-1}:
      y[nx+1,j] + (y[nx,j] - y[nx,j-1]) - x[nx-1,j] = 1;

   for {i in 1..nx-1} x[i,ny].lb = 1;
   for {j in 1..ny-1} y[nx,j].lb = 1;
```

```
    solve with nlp;
  quit;
```

The problem and solution summaries are shown in Output 8.4.1.

Output 8.4.1 Problem Summary and Solution Summary

```
                      The OPTMODEL Procedure

                         Problem Summary

           Objective Sense            Minimization
           Objective Function                    f
           Objective Type               Quadratic

           Number of Variables              20400
           Bounded Above                        0
           Bounded Below                    20400
           Bounded Below and Above              0
           Free                                 0
           Fixed                                0

           Number of Constraints             9996
           Linear LE (<=)                       0
           Linear EQ (=)                     9996
           Linear GE (>=)                       0
           Linear Range                         0


                    Performance Information

            Execution Mode       Single-Machine
            Number of Threads                 4


                       Solution Summary

            Solver                          NLP
            Algorithm           Interior Point
            Objective Function                f
            Solution Status             Optimal
            Objective Value       6237012.1174

            Optimality Error      6.8106459E-7
            Infeasibility         6.8106459E-7

            Iterations                        6
            Presolve Time                  0.02
            Solution Time                 46.04
```

Example 8.5: Solving NLP Problems That Have Several Local Minima

Some NLP problems contain many local minima. By default, the NLP solver converges to a single local minimum. However, the NLP solver can search the feasible region for other local minima. After it completes the search, it returns the point where the objective function achieves its minimum value. (This point might not be a local minimum; see the SOLTYPE= option for more details.) Consider the following example, taken from Hock and Schittkowski (1981):

$$
\begin{aligned}
\text{minimize} \quad & f(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \\
\text{subject to} \quad & x_1 + x_2^2 + x_3^3 = 2 + 3\sqrt{2} \\
& x_2 + x_4 - x_3^2 = -2 + 2\sqrt{2} \\
& x_1 x_5 = 2 \\
& -5 \le x_i \le 5, i = 1, \ldots, 5
\end{aligned}
$$

The following statements call the NLP solver to search the feasible region for different local minima. The PERFORMANCE statement requests that the multistart algorithm use up to four threads. The SEED= option is specified for reproducibility, but it is not required in running the multistart algorithm.

```
proc optmodel;
   var x{i in 1..5} >= -5 <= 5 init -2;

   min f=(x[1] - 1)^2 + (x[1] - x[2])^2 + (x[2] - x[3])^3 +
         (x[3] - x[4])^4 + (x[4] - x[5])^4;

   con g1: x[1] + x[2]^2 + x[3]^3 =  2 + 3*sqrt(2);
   con g2: x[2] + x[4]    - x[3]^2 = -2 + 2*sqrt(2);
   con g3: x[1]*x[5] = 2;

   performance nthreads=4;
   solve with nlp/multistart seed=1234 msmaxstarts=10;
   print x.init x.msinit x;
quit;
```

The PRINT statement prints the initial point (x.init) that was specified in the INIT option of the VAR declaration; the starting point (x.msinit) that led to the best local solution; and finally the best local solution (x) that was found by the NLP solver in multistart mode. The SAS log is shown in Output 8.5.1.

Output 8.5.1 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3 nonlinear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in single-machine mode.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 1234 is used.
                   Best        Local  Optimality    Infeasi-  Local  Local
      Start     Objective   Objective       Error      bility  Iters  Status
          1    607.035801  607.035801  8.62585E-7  8.62585E-7      7  Optimal
          2    607.035512  607.035512  2.81991E-7  5.88394E-9      7  Optimal
          3    52.9025715  52.9025715  1.19566E-7  9.15988E-8      7  Optimal
          4    52.9025715  52.9025734  4.37492E-7  6.32167E-8     13  Optimal
          5 R  52.9025715  607.035793  5.31529E-7  4.33395E-7      8  Optimal
          6    52.9025715  64.8740012  6.86447E-7  4.08169E-7      6  Optimal
          7    27.871905   27.871905   7.19197E-7   1.4543E-7      5  Optimal
          8    27.871905   27.8719052        5E-9   1.3246E-9      7  Optimal
          9    0.02931102  0.02931102  5.51954E-7  5.42744E-7     10  Optimal
         10    0.02931081  0.02931081        5E-7  2.66484E-7     11  Optimal
NOTE: The Multistart algorithm generated 800 sample points.
NOTE: 9 distinct local optima were found.
NOTE: The best objective value found by local solver = 0.0293108081.
NOTE: The solution found by local solver with objective = 0.0293108081 was
      returned.
```

The first column in the log indicates the index of the current starting point. An additional indicator (*, r, or R) can appear after the index to provide more information about the optimization run that started from the current point. For more information, see the section "Iteration Log for Multistart" on page 318. The second column records the best objective that has been found so far. Columns 3 to 6 report the objective value, optimality error, infeasibility, and number of iterations that the local solver returned when it was started from the current starting point. Finally, the last column records the status of the local solver—namely, whether it was able to converge to a local optimum from the current starting point.

The summaries and solution are shown in Output 8.5.2. Note that the best local solution was found by starting the local solver from a starting point (x.msinit) that is different from the point specified in the VAR declaration (x.init).

Output 8.5.2 Summaries and the Optimal Solution

```
                        The OPTMODEL Procedure

                          Problem Summary

     Objective Sense               Minimization
     Objective Function                       f
     Objective Type                  Nonlinear

     Number of Variables                      5
     Bounded Above                            0
     Bounded Below                            0
     Bounded Below and Above                  5
     Free                                     0
     Fixed                                    0

     Number of Constraints                    3
     Linear LE  (<=)                          0
     Linear EQ  (=)                           0
     Linear GE  (>=)                          0
     Linear Range                             0
     Nonlinear LE  (<=)                       0
     Nonlinear EQ  (=)                        3
     Nonlinear GE  (>=)                       0
     Nonlinear Range                          0


                       Performance Information

        Execution Mode        Single-Machine
        Number of Threads                  4


                         Solution Summary

     Solver                        Multistart NLP
     Algorithm                     Interior Point
     Objective Function                         f
     Solution Status                      Optimal
     Objective Value                   0.0293108146

     Number of Starts                         10
     Number of Sample Points                 800
     Number of Distinct Optima                 9
     Random Seed Used                       1234
     Optimality Error               6.044933E-7
     Infeasibility                  3.8669805E-7

     Presolve Time                          0.00
     Solution Time                          3.67
```

Output 8.5.2 *continued*

```
         [1]     x.INIT     x.MSINIT        x

          1         -2      -0.23856     1.1167
          2         -2       2.93912     1.2205
          3         -2       1.08486     1.5377
          4         -2      -1.11094     1.9726
          5         -2       1.80540     1.7910
```

Alternatively, the following SAS statements show how you can add the NODES= option in the PERFOR-
MANCE statement to run this example in distributed mode.

NOTE: SAS High-Performance Optimization software must be installed before you can invoke the MULTI-
START option in distributed mode.

```
proc optmodel;
   var x{i in 1..5} >= -5 <= 5 init -2;

   min f=(x[1] - 1)^2 + (x[1] - x[2])^2 + (x[2] - x[3])^3 +
         (x[3] - x[4])^4 + (x[4] - x[5])^4;

   con g1: x[1] + x[2]^2 + x[3]^3 =  2 + 3*sqrt(2);
   con g2: x[2] + x[4]    - x[3]^2 = -2 + 2*sqrt(2);
   con g3: x[1]*x[5] = 2;

   performance nodes=4 nthreads=4;
   solve with nlp/multistart seed=1234 msmaxstarts=10;
   print x;
quit;
```

The SAS log is displayed in Output 8.5.3.

Output 8.5.3 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3 nonlinear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in the distributed computing
      environment with 4 worker nodes.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 1234 is used.
                    Best        Local  Optimality   Infeasi-  Local  Local
       Start    Objective    Objective       Error     bility  Iters  Status
           1   607.035871   607.035871  8.43836E-7  8.43836E-7      8  Optimal
           2   52.9025792   52.9025792  2.39856E-7  2.84012E-8      7  Optimal
           3   52.9025792   607.035521  4.17297E-7  4.17297E-7      8  Optimal
           4   52.9025792   52.9025794  6.69221E-8  3.26907E-9      8  Optimal
           5   52.9025792   64.8739968   2.9959E-7  1.19421E-7      8  Optimal
           6   52.9025792   52.9026071  3.70891E-7  3.70891E-7     11  Optimal
           7   0.02931083   0.02931083  2.53214E-7  3.54144E-9     10  Optimal
           8   0.02931083   52.9025015  9.03254E-7  9.03254E-7      5  Optimal
           9   0.02931083   52.9025916  7.08702E-7  7.08702E-7      6  Optimal
          10   0.02931083   52.9026134  4.08103E-7  4.08103E-7      8  Optimal
NOTE: The Multistart algorithm generated 1600 sample points.
NOTE: 10 distinct local optima were found.
NOTE: The best objective value found by local solver = 0.0293108314.
NOTE: The solution found by local solver with objective = 0.0293108314 was
      returned.
```

Output 8.5.4 shows the summaries and solution. Note that the "Performance Information" table shows that four computing nodes with four threads on each node are used in distributed mode.

Output 8.5.4 Summaries and the Optimal Solution

```
                    The OPTMODEL Procedure

                        Problem Summary

        Objective Sense            Minimization
        Objective Function                    f
        Objective Type               Nonlinear

        Number of Variables                  5
        Bounded Above                        0
        Bounded Below                        0
        Bounded Below and Above              5
        Free                                 0
        Fixed                                0

        Number of Constraints                3
        Linear LE  (<=)                      0
        Linear EQ  (=)                       0
        Linear GE  (>=)                      0
        Linear Range                         0
        Nonlinear LE  (<=)                   0
        Nonlinear EQ  (=)                    3
        Nonlinear GE  (>=)                   0
        Nonlinear Range                      0


                    Performance Information

    Host Node                     << your grid host >>
    Execution Mode                         Distributed
    Grid Mode                               Symmetric
    Number of Compute Nodes                         4
    Number of Threads per Node                      4


                      Solution Summary

        Solver                     Multistart NLP
        Algorithm                  Interior Point
        Objective Function                      f
        Solution Status                   Optimal
        Objective Value             0.0293108314

        Number of Starts                       10
        Number of Sample Points              1600
        Number of Distinct Optima              10
        Random Seed Used                     1234
        Optimality Error            2.5321436E-7
        Infeasibility               3.5414378E-9

        Presolve Time                        0.00
        Solution Time                        3.99
```

Output 8.5.4 *continued*

```
            [1]        x

            1       1.1167
            2       1.2204
            3       1.5378
            4       1.9727
            5       1.7911
```

References

Akrotirianakis, I. and Rustem, B. (2005), "Globally Convergent Interior-Point Algorithm for Nonlinear Programming," *Journal of Optimization Theory and Applications*, 125, 497–521.

Armand, P., Gilbert, J. C., and Jan-Jégou, S. (2002), "A BFGS-IP Algorithm for Solving Strongly Convex Optimization Problems with Feasibility Enforced by an Exact Penalty Approach," *Mathematical Programming*, 92, 393–424.

Erway, J., Gill, P. E., and Griffin, J. D. (2007), "Iterative Solution of Augmented Systems Arising in Interior Point Methods," *SIAM Journal on Optimization*, 18, 666–690.

Forsgren, A. and Gill, P. E. (1998), "Primal-Dual Interior Methods for Nonconvex Nonlinear Programming," *SIAM Journal on Optimization*, 8, 1132–1152.

Forsgren, A., Gill, P. E., and Wright, M. H. (2002), "Interior Methods for Nonlinear Optimization," *SIAM Review*, 44, 525–597.

Gill, P. E. and Robinson, D. P. (2010), "A Primal-Dual Augmented Lagrangian," *Computational Optimization and Applications*, 47, 1–25.

Gould, N. I. M., Orban, D., and Toint, Ph. L. (2003), "CUTEr and SifDec: A Constrained and Unconstrained Testing Environment, Revised," *ACM Transactions on Mathematical Software*, 29, 373–394.

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.

Nocedal, J. and Wright, S. J. (1999), *Numerical Optimization*, New York: Springer-Verlag.

Vanderbei, R. J. (1999), "LOQO: An Interior Point Code for Quadratic Programming," *Optimization Methods and Software*, 11, 451–484.

Wächter, A. and Biegler, L. T. (2006), "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming," *Mathematical Programming*, 106, 25–57.

Wright, S. J. (1997), *Primal-Dual Interior-Point Methods*, Philadelphia: SIAM Publications.

Yamashita, H. (1998), "A Globally Convergent Primal-Dual Interior Point Method for Constrained Optimization," *Optimization Methods and Software*, 10, 443–469.

Chapter 9
The Quadratic Programming Solver

Contents

Overview: QP Solver

The OPTMODEL procedure provides a framework for specifying and solving quadratic programs.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

$$
\begin{aligned}
\min \quad & \tfrac{1}{2}\, \mathbf{x}^{\mathrm{T}} \mathbf{Q} \mathbf{x} + \mathbf{c}^{\mathrm{T}} \mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \; \{\geq, =, \leq\} \; \mathbf{b} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{aligned}
$$

where

$\mathbf{Q}$	$\in$	$\mathbb{R}^{n \times n}$	is the quadratic (also known as Hessian) matrix
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the constraints matrix
$\mathbf{x}$	$\in$	$\mathbb{R}^{n}$	is the vector of decision variables
$\mathbf{c}$	$\in$	$\mathbb{R}^{n}$	is the vector of linear objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^{m}$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^{n}$	is the vector of lower bounds on the decision variables
$\mathbf{u}$	$\in$	$\mathbb{R}^{n}$	is the vector of upper bounds on the decision variables

The quadratic matrix $\mathbf{Q}$ is assumed to be symmetric; that is,

$$q_{ij} = q_{ji}, \quad \forall i, j = 1, \ldots, n$$

Indeed, it is easy to show that even if $\mathbf{Q} \neq \mathbf{Q}^{\mathrm{T}}$, then the simple modification

$$\tilde{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} + \mathbf{Q}^{\mathrm{T}})$$

produces an equivalent formulation $\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \mathbf{x}^{\mathrm{T}}\tilde{\mathbf{Q}}\mathbf{x}$; hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, $\mathbf{Q}$ is also required to be positive semidefinite for minimization type of models:

$$\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^n$$

$\mathbf{Q}$ is required to be negative semidefinite for maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

$$\mathbf{Q} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

or as a consequence of physical laws, and so on. See Figure 9.1 for examples of convex, concave, and nonconvex objective functions.

Figure 9.1 Examples of Convex, Concave, and Nonconvex Objective Functions
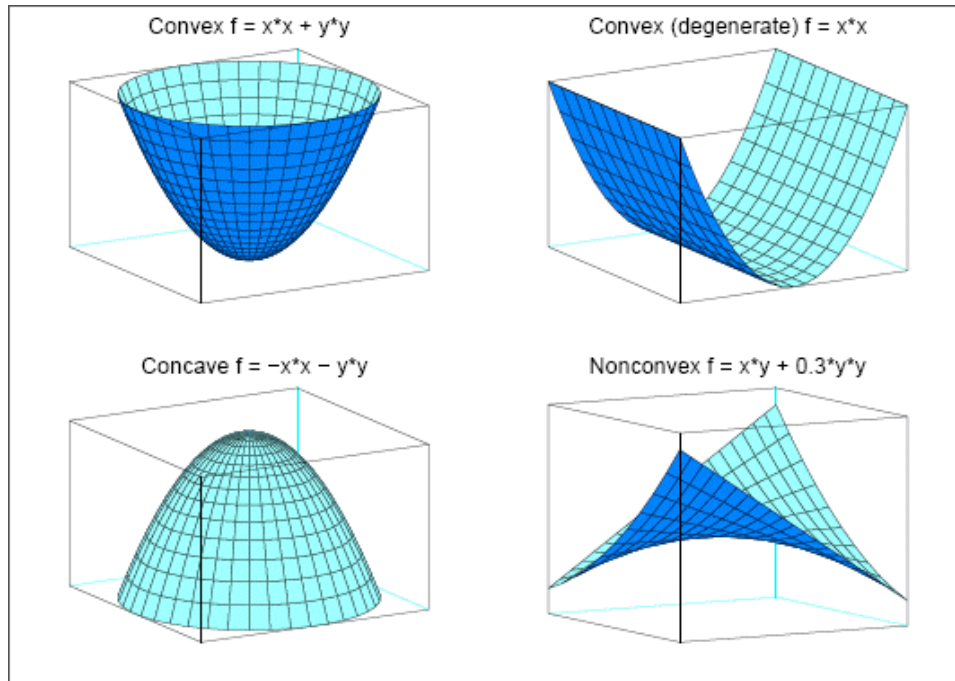
The order of constraints is insignificant. Some or all components of $\mathbf{l}$ or $\mathbf{u}$ (lower and upper bounds, respectively) can be omitted.

Getting Started: QP Solver

Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function $f(x_1, x_2)$ on the nonnegative quadrant, subject to two constraints:

$$
\begin{array}{rcccccccc}
\min & 2x_1 & + & 3x_2 & + & x_1^2 & + & 10x_2^2 & + & 2.5x_1x_2 \\
\text{subject to} & x_1 & - & x_2 & \leq & 1 \\
& x_1 & + & 2x_2 & \geq & 100 \\
& x_1 & & & \geq & 0 \\
& & & x_2 & \geq & 0
\end{array}
$$

To use the OPTMODEL procedure, it is not necessary to fit this problem into the general QP formulation mentioned in the section "Overview: QP Solver" on page 339 and to compute the corresponding parameters. However, since these parameters are closely related to the data set that is used by the OPTQP procedure and has a quadratic programming system (QPS) format, you can compute these parameters as follows. The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

$$
\mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 100 \end{bmatrix}, \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}
$$

Carefully construct the quadratic matrix $\mathbf{Q}$. Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

$$
\frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \equiv \frac{1}{2} \sum_{i,j=1}^{n} x_i\, q_{ij}\, x_j = \frac{1}{2} \sum_{i=1}^{n} q_{ii}\, x_i^2 + \sum_{i>j} x_i\, q_{ij}\, x_j
$$

The first expression

$$
\frac{1}{2} \sum_{i=1}^{n} q_{ii}\, x_i^2
$$

sums the main-diagonal elements. Thus, in this case you have

$$
q_{11} = 2, \quad q_{22} = 20
$$

Notice that the main-diagonal values are doubled in order to accommodate the 1/2 factor. Now the second term

$$
\sum_{i>j} x_i\, q_{ij}\, x_j
$$

sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal $(x_i\, x_j, \ i \neq j)$ term in the objective function is $2.5\, x_1\, x_2$, so you have

$$
q_{21} = 2.5
$$

Notice that you do not need to specify the upper triangular part of the quadratic matrix.

Finally, the matrix of constraints is as follows:

$$
\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}
$$

The following OPTMODEL program formulates the preceding problem in a manner that is very close to the mathematical specification of the given problem:

```
/* getting started */
proc optmodel;
   var x1 >= 0; /* declare nonnegative variable x1 */
   var x2 >= 0; /* declare nonnegative variable x2 */

   /* objective: quadratic function f(x1, x2) */
   minimize f =
       /* the linear objective function coefficients */
       2 * x1 + 3 * x2 +

       /* quadratic <x, Qx> */
       x1 * x1 + 2.5 * x1 * x2 + 10 * x2 * x2;

   /* subject to the following constraints */
   con r1: x1 - x2 <= 1;
   con r2: x1 + 2 * x2 >= 100;

   /* specify iterative interior point algorithm (QP)
    * in the SOLVE statement */
   solve with qp;

   /* print the optimal solution */
   print x1 x2;
   save qps qpsdata;
quit;
```

The "with qp" clause in the SOLVE statement invokes the QP solver to solve the problem. The output is shown in Figure 9.2.

Figure 9.2 Summaries and Optimal Solution

```
                         The OPTMODEL Procedure

                            Problem Summary

              Objective Sense              Minimization
              Objective Function                      f
              Objective Type                  Quadratic

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

              Number of Constraints                   2
              Linear LE (<=)                          1
              Linear EQ (=)                           0
              Linear GE (>=)                          1
              Linear Range                            0

              Constraint Coefficients                 4
```

Figure 9.2 *continued*

```
                    Performance Information

        Execution Mode         Single-Machine
        Number of Threads                   4


                      Solution Summary

        Solver                             QP
        Algorithm              Interior Point
        Objective Function                  f
        Solution Status               Optimal
        Objective Value                 15018

        Primal Infeasibility      2.110419E-16
        Dual Infeasibility        8.727374E-15
        Bound Infeasibility                  0
        Duality Gap               3.633377E-16
        Complementarity                      0

        Iterations                           6
        Presolve Time                     0.00
        Solution Time                     0.14


                  x1     x2

                  34     33
```

In this example, the SAVE QPS statement is used to save the QP problem in the QPS-format data set qpsdata, shown in Figure 9.3. The data set is consistent with the parameters of general quadratic programming previously computed. Also, the data set can be used as input to the OPTQP procedure.

Figure 9.3 QPS-Format Data Set

```
Obs     FIELD1      FIELD2      FIELD3      FIELD4      FIELD5      FIELD6

 1     NAME                    qpsdata        .                       .
 2     ROWS                                   .                       .
 3     N           f                          .                       .
 4     L           r1                         .                       .
 5     G           r2                         .                       .
 6     COLUMNS                                .                       .
 7                 x1          f            2.0         r1          1
 8                 x1          r2           1.0                     .
 9                 x2          f            3.0         r1         -1
10                 x2          r2           2.0                     .
11     RHS                                   .                       .
12                 .RHS.       r1           1.0                     .
13                 .RHS.       r2         100.0                     .
14     QSECTION                              .                       .
15                 x1          x1           2.0                     .
16                 x1          x2           2.5                     .
17                 x2          x2          20.0                     .
18     ENDATA                                .                       .
```

Syntax: QP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH QP < / *options* > **;**

Functional Summary

Table 9.1 summarizes the list of options available for the SOLVE WITH QP statement, classified by function.

Table 9.1 Options for the QP Solver

Description	Option
Control Options	
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the type of presolve	PRESOLVER=
Interior Point Algorithm Options	
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=
Specifies units of CPU time or real time	TIMETYPE=

QP Solver Options

This section describes the options recognized by the QP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the QP solver is explicitly specified using a WITH clause.

The QP solver does not provide an intermediate solution if the solver terminates before reaching optimality.

Control Options

LOGFREQ=k

PRINTFREQ=k

> specifies that the printing of the solution progress to the iteration log is to occur after every k iterations. The print frequency, k, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.
>
> The value $k = 0$ disables the printing of the progress of the solution. The default value of this option is 1.

MAXITER=k

> specifies the maximum number of iterations. The value k can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=t

> specifies an upper limit of t units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of t can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

PRESOLVER=number | string

PRESOL=number | string

> specifies one of the following presolve options:

number	string	Description
0	NONE	Disables presolver.
−1	AUTOMATIC	Applies presolver by using default setting.

> You can specify the PRESOLVER= value either by a character-valued option or by an integer. The default option is AUTOMATIC.

Interior Point Algorithm Options

STOP_DG=δ

> specifies the desired relative duality gap, $\delta \in [1E\text{--}9, 1E\text{--}4]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 346 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 346 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 346 for details.

TIMETYPE=*number* | *string*

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 9.3 describes the valid values of the TIMETYPE= option.

Table 9.3 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The Optimization Statistics table, an output of PROC OPTMODEL if option PRINTLEVEL=2 is specified in the PROC OPTMODEL statement, also includes the same time units for "Presolver Time" and "Solver Time." The other times (such as "Problem Generation Time") in the Optimization Statistics table are always CPU times.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement of the OPTMODEL procedure. See the section "PERFOR-MANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and NODES= options.

If you specify a value greater than 1 for either the NTHREADS= or NODES= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and NODES= options, the default value of the TIMETYPE= option is CPU.

Details: QP Solver

Interior Point Algorithm: Overview

The QP solver implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

$$
\begin{aligned}
\min \quad & \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\
& \mathbf{x} \geq \mathbf{0}
\end{aligned}
$$

The corresponding dual formulation is

$$
\begin{array}{rlll}
\max & -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} & + & \mathbf{b}^{\mathrm{T}}\mathbf{y} \\
\text{subject to} & -\mathbf{Q}\mathbf{x} & + & \mathbf{A}^{\mathrm{T}}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\
& & & & & \mathbf{y} & \geq & \mathbf{0} \\
& & & & & \mathbf{w} & \geq & \mathbf{0}
\end{array}
$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables:

$$
\begin{array}{rcll}
\mathbf{A}\mathbf{x} - \mathbf{s} & = & \mathbf{b} & \text{(primal feasibility)} \\
-\mathbf{Q}\mathbf{x} + \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & \text{(dual feasibility)} \\
\mathbf{W}\mathbf{X}\mathbf{e} & = & \mathbf{0} & \text{(complementarity)} \\
\mathbf{S}\mathbf{Y}\mathbf{e} & = & \mathbf{0} & \text{(complementarity)} \\
\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{s} & \geq & \mathbf{0}
\end{array}
$$

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding x, y, w, and s on the main diagonal and zero elsewhere, as in the following example:

$$
\mathbf{X} \equiv
\begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

$$
\begin{bmatrix}
\mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\
\mathbf{A}^{\mathrm{T}} & -\mathbf{Q} - \mathbf{X}^{-1}\mathbf{W}
\end{bmatrix}
\begin{bmatrix}
\Delta\mathbf{y} \\
\Delta\mathbf{x}
\end{bmatrix}
=
\begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The QP solver uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector v:

- relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{Qx} + \mathbf{c} - \mathbf{A}^\mathsf{T}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{x}^\mathsf{T}\mathbf{Qx} + \mathbf{c}^\mathsf{T}\mathbf{x} - \mathbf{b}^\mathsf{T}\mathbf{y}|}{|\frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{Qx} + \mathbf{c}^\mathsf{T}\mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Iteration Log

The following information is displayed in the iteration log:

Iter	indicates the iteration number.
Complement	indicates the (absolute) complementarity.
Duality Gap	indicates the (relative) duality gap.
Primal Infeas	indicates the (relative) primal infeasibility measure.
Bound Infeas	indicates the (relative) bound infeasibility measure.
Dual Infeas	indicates the (relative) dual infeasibility measure.

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the option MAXITER= or MAXTIME=. If the complementarity or the duality gap does not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure

causes the ODS table ProblemStatistics to be generated when the QP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 9.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 9.4 ODS Table ProblemStatistics

```
                        The OPTMODEL Procedure

                          Problem Statistics

         Number of Constraint Matrix Nonzeros           4
         Maximum Constraint Matrix Coefficient          2
         Minimum Constraint Matrix Coefficient          1
         Average Constraint Matrix Coefficient       1.25

         Number of Linear Objective Nonzeros            2
         Maximum Linear Objective Coefficient           3
         Minimum Linear Objective Coefficient           2
         Average Linear Objective Coefficient         2.5

         Number of Lower Triangular Hessian Nonzeros    1
         Number of Diagonal Hessian Nonzeros            2
         Maximum Hessian Coefficient                   20
         Minimum Hessian Coefficient                    2
         Average Hessian Coefficient                 6.75

         Number of RHS Nonzeros                         2
         Maximum RHS                                  100
         Minimum RHS                                    1
         Average RHS                                 50.5

         Maximum Number of Nonzeros per Column          2
         Minimum Number of Nonzeros per Column          2
         Average Number of Nonzeros per Column          2

         Maximum Number of Nonzeros per Row             2
         Minimum Number of Nonzeros per Row             2
         Average Number of Nonzeros per Row             2
```

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the QP solver is called are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, occurred.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take the following value:

IP	The interior point algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FUNCTION_CALL_LIMIT_REACHED	The solver reached its limit on function evaluations.
INTERRUPTED	The solver was interrupted externally.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY

indicates the (relative) infeasibility of the primal constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 346 for details.

DUAL_INFEASIBILITY

indicates the (relative) infeasibility of the dual constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 346 for details.

BOUND_INFEASIBILITY

indicates the (relative) violation by the solution of the lower or upper bounds (or both). See the section "Interior Point Algorithm: Overview" on page 346 for details.

DUALITY_GAP

indicates the (relative) duality gap. See the section "Interior Point Algorithm: Overview" on page 346 for details.

COMPLEMENTARITY

indicates the (absolute) complementarity at the solution. See the section "Interior Point Algorithm: Overview" on page 346 for details.

ITERATIONS

indicates the number of iterations required to solve the problem.

PRESOLVE_TIME

indicates the time taken for preprocessing (seconds).

SOLUTION_TIME

indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: QP Solver

This section presents examples that illustrate the use of the OPTMODEL procedure to solve quadratic programming problems. Example 9.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTMODEL. Example 9.2 and Example 9.3 show in detail how to model the portfolio optimization and selection problems.

Example 9.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $m > n$. Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$.

This problem is called a least squares problem for the following reason. Let $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ be defined as previously. Let $k_i(x)$ be the kth component of the vector $\mathbf{A}\mathbf{x} - \mathbf{b}$:

$$k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i, \ i = 1, 2, \ldots, m$$

By definition of the Euclidean norm, the objective function can be expressed as follows:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 = \sum_{i=1}^{m} k_i(x)^2$$

Therefore, the function you minimize is the sum of squares of m terms $k_i(x)$; hence the term least squares. The following example is an illustration of the *linear* least squares problem; that is, each of the terms k_i is a linear function of x.

Consider the following least squares problem defined by

$$\mathbf{A} = \begin{bmatrix} 4 & 0 \\ -1 & 1 \\ 3 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

This translates to the following set of linear equations:

$$4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1$$

The corresponding least squares problem is:

$$\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2$$

The preceding objective function can be expanded to:

$$\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2$$

In addition, you impose the following constraint so that the equation $3x_1 + 2x_2 = 1$ is satisfied within a tolerance of 0.1:

$$0.9 \le 3x_1 + 2x_2 \le 1.1$$

You can use the following SAS statements to solve the least squares problem:

```
/* example 1: linear least-squares problem */
proc optmodel;
   var x1; /* declare free (no explicit bounds) variable x1 */
   var x2; /* declare free (no explicit bounds) variable x2 */
   /* declare slack variable for ranged constraint */
   var w >= 0 <= 0.2;

   /* objective function: minimize is the sum of squares */
   minimize f = 26 * x1 * x1 + 5 * x2 * x2 + 10 * x1 * x2
       - 14 * x1 - 4 * x2 + 2;

   /* subject to the following constraint */
   con L: 3 * x1 + 2 * x2 - w = 0.9;

   solve with qp;

   /* print the optimal solution */
   print x1 x2;
quit;
```

The output is shown in Output 9.1.1.

Output 9.1.1 Summaries and Optimal Solution

```
                        The OPTMODEL Procedure


                          Problem Summary

        Objective Sense            Minimization
        Objective Function                    f
        Objective Type                Quadratic

        Number of Variables                   3
        Bounded Above                         0
        Bounded Below                         0
        Bounded Below and Above               1
        Free                                  2
        Fixed                                 0

        Number of Constraints                 1
        Linear LE (<=)                        0
        Linear EQ (=)                         1
        Linear GE (>=)                        0
        Linear Range                          0

        Constraint Coefficients               3


                     Performance Information

          Execution Mode        Single-Machine
          Number of Threads                   4


                        Solution Summary

        Solver                               QP
        Algorithm                 Interior Point
        Objective Function                    f
        Solution Status                 Optimal
        Objective Value             0.0095238095

        Primal Infeasibility        9.742207E-14
        Dual Infeasibility           1.305621E-9
        Bound Infeasibility         5.782412E-18
        Duality Gap                 4.9397126E-8
        Complementarity             1.5357015E-7

        Iterations                            3
        Presolve Time                      0.00
        Solution Time                      0.02


                    x1          x2

                0.2381      0.1619
```

Example 9.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio's total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $\mathcal{B}$ be the amount of capital you have, $\mathbf{R}$ be the random vector of asset returns over some period, and $\mathbf{r}$ be the expected value of $\mathbf{R}$. Let G be the minimum growth you hope to obtain, and $\mathcal{C}$ be the covariance matrix of $\mathbf{R}$. The objective function is $\mathrm{Var}\left(\sum_{i=1}^{n} x_i R_i\right)$, which can be equivalently denoted as $\mathbf{x}^T \mathcal{C} \mathbf{x}$.

Assume, for example, $n = 4$. Let $\mathcal{B} = 10{,}000$, $G = 1{,}000$, $\mathbf{r} = [0.05, -0.2, 0.15, 0.30]$, and

$$
\mathcal{C} = \begin{bmatrix}
0.08 & -0.05 & -0.05 & -0.05 \\
-0.05 & 0.16 & -0.02 & -0.02 \\
-0.05 & -0.02 & 0.35 & 0.06 \\
-0.05 & -0.02 & 0.06 & 0.35
\end{bmatrix}
$$

The QP formulation can be written as:

$$
\begin{aligned}
\min \quad & 0.08x_1^2 \;-\; 0.1x_1x_2 \;-\; 0.1x_1x_3 \;-\; 0.1x_1x_4 \;+ \\
& 0.16x_2^2 \;-\; 0.04x_2x_3 \;-\; 0.02x_2x_4 \;+\; 0.35x_3^2 \;+ \\
& 0.12x_3x_4 \;+\; 0.35x_4^2
\end{aligned}
$$

subject to

$$
\begin{array}{lrclcrcrclc}
\text{(budget)} & x_1 & + & x_2 & + & x_3 & + & x_4 & \leq & 10000 \\
\text{(growth)} & 0.05x_1 & - & 0.2x_2 & + & 0.15x_3 & + & 0.30x_4 & \geq & 1000 \\
& & & & & & & x_1, x_2, x_3, x_4 & \geq & 0
\end{array}
$$

Use the following SAS statements to solve the problem:

```
/* example 2: portfolio optimization */
proc optmodel;
   /* let x1, x2, x3, x4 be the amount invested in each asset */
   var x{1..4} >= 0;

   num coeff{1..4, 1..4} = [0.08 -.05 -.05 -.05
                            -.05 0.16 -.02 -.02
                            -.05 -.02 0.35 0.06
                            -.05 -.02 0.06 0.35];
   num r{1..4}=[0.05 -.20 0.15 0.30];

   /* minimize the variance of the portfolio's total return */
   minimize f = sum{i in 1..4, j in 1..4}coeff[i,j]*x[i]*x[j];

   /* subject to the following constraints */
   con BUDGET: sum{i in 1..4}x[i] <= 10000;
   con GROWTH: sum{i in 1..4}r[i]*x[i] >= 1000;
```

```
solve with qp;

/* print the optimal solution */
print x;
```

The summaries and the optimal solution are shown in Output 9.2.1.

Output 9.2.1 Portfolio Optimization

```
                      The OPTMODEL Procedure

                         Problem Summary

          Objective Sense              Minimization
          Objective Function                      f
          Objective Type                  Quadratic

          Number of Variables                     4
          Bounded Above                           0
          Bounded Below                           4
          Bounded Below and Above                 0
          Free                                    0
          Fixed                                   0

          Number of Constraints                   2
          Linear LE (<=)                          1
          Linear EQ (=)                           0
          Linear GE (>=)                          1
          Linear Range                            0

          Constraint Coefficients                 8


                    Performance Information

           Execution Mode        Single-Machine
           Number of Threads                   4


                       Solution Summary

          Solver                                 QP
          Algorithm                   Interior Point
          Objective Function                      f
          Solution Status                   Optimal
          Objective Value              2232313.4432

          Primal Infeasibility         2.9198351E-9
          Dual Infeasibility           1.9359348E-8
          Bound Infeasibility                     0
          Duality Gap                  1.3492254E-7
          Complementarity              1.0600935E-7

          Iterations                              7
          Presolve Time                        0.00
          Solution Time                        0.02
```

Output 9.2.1 *continued*

```
               [1]          x

                1       3452.9
                2          0.0
                3       1068.8
                4       2223.5
```

Thus, the minimum variance portfolio that earns an expected return of at least 10% is $x_1 = 3,452$, $x_2 = 0$, $x_3 = 1,068$, $x_4 = 2,223$. Asset 2 gets nothing because its expected return is -20% and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop the nonnegativity assumption?

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and "more than 100%" in high-mean assets.

To solve the portfolio optimization problem with the short-sale option, continue to submit the following SAS statements:

```
   /* example 2: portfolio optimization with short-sale option */
   /* dropping nonnegativity assumption */
   for {i in 1..4} x[i].lb=-x[i].ub;

   solve with qp;

   /* print the optimal solution */
   print x;
quit;
```

You can see in the optimal solution displayed in Output 9.2.2 that the decision variable x_2, denoting Asset 2, is equal to $-1,563.61$, which means short sale of that asset.

Output 9.2.2 Portfolio Optimization with Short-Sale Option

```
                    The OPTMODEL Procedure

                      Solution Summary

            Solver                             QP
            Algorithm               Interior Point
            Objective Function                  f
            Solution Status               Optimal
            Objective Value           1907122.2254

            Primal Infeasibility       6.4988973E-8
            Dual Infeasibility         7.3644141E-9
            Bound Infeasibility                   0
            Duality Gap                2.0104247E-8
            Complementarity            6.299141E-7

            Iterations                           5
            Presolve Time                     0.00
            Solution Time                     0.01


                    [1]             x

                     1         1684.35
                     2        -1563.61
                     3          682.51
                     4         1668.95
```

Example 9.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector $\mathbf{c} = [200, 300, 500]$, the amount of asset bought and sold is denoted by b_i and s_i, respectively, and the net investment in each asset is denoted by x_i and is defined by the following relation:

$$x_i - b_i + s_i = c_i, \quad i = 1, 2, 3$$

Suppose that you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix $\mathcal{C}$ be defined as

$$\mathcal{C} \;=\; \begin{bmatrix} 0.027489 & -0.00874 & -0.00015 \\ -0.00874 & 0.109449 & -0.00012 \\ -0.00015 & -0.00012 & 0.000766 \end{bmatrix}$$

Assume that you hope to obtain at least 12% growth. Let $\mathbf{r} = [1.109048, 1.169048, 1.074286]$ be the vector of expected return on the three assets, and let $\mathcal{B}=1000$ be the available funds. Mathematically, this problem

can be written in the following manner:

$$\min \quad 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2$$
$$-0.00024x_2x_3 + 0.000766x_3^2$$

subject to

(return)		$\sum_{i=1}^3 r_i x_i$	$\geq$	$1.12\mathcal{B}$
(budget)	$\sum_{i=1}^3 x_i + \sum_{i=1}^3 0.01(b_i + s_i)$		$=$	$\mathcal{B}$
(balance)		$x_i - b_i + s_i$	$=$	$c_i, \quad i = 1, 2, 3$
		x_i, b_i, s_i	$\geq$	$0, \quad i = 1, 2, 3$

The problem can be solved by the following SAS statements:

```
/* example 3: portfolio selection with transactions */
proc optmodel;
   /* let x1, x2, x3 be the amount invested in each asset */
   var x{1..3} >= 0;
   /* let b1, b2, b3 be the amount of asset bought */
   var b{1..3} >= 0;
   /* let s1, s2, s3 be the amount of asset sold */
   var s{1..3} >= 0;

   /* current holdings */
   num c{1..3}=[ 200 300 500];
   /* covariance matrix */
   num coeff{1..3, 1..3} = [0.027489  -.008740  -.000150
                           -.008740   0.109449  -.000120
                           -.000150  -.000120   0.000766];
   /* returns */
   num r{1..3}=[1.109048 1.169048 1.074286];

   /* minimize the variance of the portfolio's total return */
   minimize f = sum{i in 1..3, j in 1..3}coeff[i,j]*x[i]*x[j];

   /* subject to the following constraints */
   con BUDGET: sum{i in 1..3}(x[i]+.01*b[i]+.01*s[i]) <= 1000;
   con RETURN: sum{i in 1..3}r[i]*x[i] >= 1120;
   con BALANC{i in 1..3}: x[i]-b[i]+s[i]=c[i];

   solve with qp;

   /* print the optimal solution */
   print x;
quit;
```

The output is displayed in .

Output 9.3.1 Portfolio Selection with Transactions

```
                    The OPTMODEL Procedure

                       Problem Summary

     Objective Sense              Minimization
     Objective Function                      f
     Objective Type                  Quadratic

     Number of Variables                     9
     Bounded Above                           0
     Bounded Below                           9
     Bounded Below and Above                 0
     Free                                    0
     Fixed                                   0

     Number of Constraints                   5
     Linear LE (<=)                          1
     Linear EQ (=)                           3
     Linear GE (>=)                          1
     Linear Range                            0

     Constraint Coefficients                21


                    Performance Information

      Execution Mode        Single-Machine
      Number of Threads                  4


                      Solution Summary

     Solver                                 QP
     Algorithm                   Interior Point
     Objective Function                      f
     Solution Status                   Optimal
     Objective Value              19560.725753

     Primal Infeasibility         1.103614E-16
     Dual Infeasibility           2.559425E-14
     Bound Infeasibility                     0
     Duality Gap                  7.221601E-16
     Complementarity                         0

     Iterations                             11
     Presolve Time                        0.00
     Solution Time                        0.02


                   [1]           x

                   1        397.58
                   2        406.12
                   3        190.17
```

References

Freund, R. W. (1991), "On Polynomial Preconditioning and Asymptotic Convergence Factors for Indefinite Hermitian Matrices," *Linear Algebra and Its Applications*, 154–156, 259–288.

Freund, R. W. and Jarre, F. (1997), "A QMR-Based Interior Point Algorithm for Solving Linear Programs," *Mathematical Programming*, 76, 183–210.

Freund, R. W. and Nachtigal, N. M. (1996), "QMRPACK: A Package of QMR Algorithms," *ACM Transactions on Mathematical Software*, 22, 46–77.

Vanderbei, R. J. (1999), "LOQO: An Interior Point Code for Quadratic Programming," *Optimization Methods and Software*, 11, 451–484.

Wright, S. J. (1996), *Primal-Dual Interior Point Algorithms*, Philadelphia: SIAM Publications.

Chapter 10
The OPTLP Procedure

Contents

Overview: OPTLP Procedure

The OPTLP procedure provides four methods of solving linear programs (LPs). A linear program has the following formulation:

$$\begin{array}{rl} \min & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \ \{\geq, =, \leq\} \ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables

The following LP solvers are available in the OPTLP procedure:

- primal simplex solver

- dual simplex solver

- network simplex solver

- interior point solver

The primal and dual simplex solvers implement the two-phase simplex method. In phase I, the solver tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the solver enters phase II to solve the original LP. The network simplex solver extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point solver implements a primal-dual predictor-corrector interior point algorithm.

PROC OPTLP 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. For details about the MPS format see Chapter 15, "The MPS-Format SAS Data Set."

You can use the MPSOUT= option to convert typical PROC LP format data sets into MPS-format SAS data sets. The option is available in the LP, INTPOINT, and NETFLOW procedures. For details about this option, see Chapter 5, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), Chapter 4, "The INTPOINT Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), and Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

Getting Started: OPTLP Procedure

The following example illustrates how you can use the OPTLP procedure to solve linear programs. Suppose you want to solve the following problem:

$$
\begin{array}{rrcrcrcrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & & \\
\text{subject to} & & - & 2x_2 & - & 3x_3 & \geq & -5 & \text{(R1)} \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 & \text{(R2)} \\
& x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 & \text{(R3)} \\
& x_1, & & x_2, & & x_3 & \geq & 0 &
\end{array}
$$

The corresponding MPS-format SAS data set is as follows:

```
data example;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .       EXAMPLE   .      .      .
ROWS          .       .         .      .      .
N          COST       .         .      .      .
G          R1         .         .      .      .
L          R2         .         .      .      .
L          R3         .         .      .      .
COLUMNS       .       .         .      .      .
   .       X1       COST        2    R2      1
   .       X1       R3          1    .       .
   .       X2       COST       -3    R1     -2
   .       X2       R2          1    R3      2
   .       X3       COST       -4    R1     -3
   .       X3       R2          2    R3      3
RHS           .       .         .      .      .
   .       RHS      R1         -5    R2      4
   .       RHS      R3          7    .       .
ENDATA        .       .         .      .      .
;
```

You can also create this data set from an MPS-format flat file (examp.mps) by using the following SAS macro:

```
%mps2sasd(mpsfile = "examp.mps", outdata = example);
```

NOTE: The SAS macro %MPS2SASD is provided in SAS/OR software. See "Converting an MPS/QPS-Format File: %MPS2SASD" on page 608 for details.

You can use the following statement to call the OPTLP procedure:

```
title1 'The OPTLP Procedure';
proc optlp data = example
  objsense  = min
  presolver = automatic
  algorithm = primal
  primalout = expout
  dualout   = exdout;
run;
```

NOTE: The "N" designation for "COST" in the rows section of the data set example also specifies a minimization problem. See the section "ROWS Section" on page 601 for details.

The optimal primal and dual solutions are stored in the data sets expout and exdout, respectively, and are displayed in Figure 10.1.

```
title2 'Primal Solution';
proc print data=expout label;
run;

title2 'Dual Solution';
proc print data=exdout label;
run;
```

Figure 10.1 Primal and Dual Solution Output

```
                          The OPTLP Procedure
                            Primal Solution

                  Objective
                  Function      RHS      Variable     Variable     Objective
          Obs        ID          ID        Name         Type       Coefficient

           1        COST        RHS        X1            N              2
           2        COST        RHS        X2            N             -3
           3        COST        RHS        X3            N             -4


                  Lower        Upper      Variable     Variable     Reduced
          Obs     Bound        Bound        Value       Status       Cost

           1        0       1.7977E308      0.0          L           2.0
           2        0       1.7977E308      2.5          B           0.0
           3        0       1.7977E308      0.0          L           0.5


                          The OPTLP Procedure
                            Dual Solution

            Objective                                                   Constraint
            Function      RHS     Constraint    Constraint   Constraint    Lower
     Obs       ID          ID        Name          Type         RHS        Bound

      1       COST        RHS        R1            G            -5           .
      2       COST        RHS        R2            L             4           .
      3       COST        RHS        R3            L             7           .


           Constraint      Dual
             Upper       Variable    Constraint    Constraint
     Obs     Bound        Value        Status       Activity

      1         .          1.5          U            -5.0
      2         .          0.0          B             2.5
      3         .          0.0          B             5.0
```

For details about the type and status codes displayed for variables and constraints, see the section "Data Input and Output" on page 373.

Syntax: OPTLP Procedure

The following statements are available in the OPTLP procedure:

PROC OPTLP < *options* > **;**
    DECOMP < *options* > **;**
    DECOMP_MASTER < *options* > **;**
    DECOMP_SUBPROB < *options* > **;**
    PERFORMANCE < *performance-options* > **;**

Functional Summary

Table 10.1 summarizes the list of options available for the OPTLP procedure, classified by function.

Table 10.1 Options for the OPTLP Procedure

Description	Option
Data Set Options	
Specifies the input data set	DATA=
Specifies the dual input data set for warm start	DUALIN=
Specifies the dual solution output data set	DUALOUT=
Specifies whether the LP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal input data set for warm start	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL
Solver Options	
Enables or disables IIS detection	IIS=
Specifies the type of solver	ALGORITHM=
Specifies the type of solver called after network simplex	ALGORITHM2=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Enables or disables printing summary	PRINTLEVEL=
Specifies units of CPU time or real time	TIMETYPE=
Simplex Algorithm Options	
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=

Table 10.1 (continued)

Description	Option
Specifies the queue size for determining entering variable	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Interior Point Algorithm Options	
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=

PROC OPTLP Statement

> **PROC OPTLP** < *options* > ;

You can specify the following options in the PROC OPTLP statement.

Data Set Options

DATA=SAS-data-set

specifies the input data set corresponding to the LP model. If this option is not specified, PROC OPTLP will use the most recently created SAS data set. See Chapter 15, "The MPS-Format SAS Data Set," for more details about the input data set.

DUALIN=SAS-data-set

DIN=SAS-data-set

specifies the input data set corresponding to the dual solution that is required for warm starting the primal and dual simplex solvers. See the section "Data Input and Output" on page 373 for details.

DUALOUT=SAS-data-set

DOUT=SAS-data-set

specifies the output data set for the dual solution. This data set contains the dual solution information. See the section "Data Input and Output" on page 373 for details.

OBJSENSE=option

specifies whether the LP model is a minimization or a maximization problem. You specify OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set; see the section "ROWS Section" on page 601 for details. If for some reason the objective sense is specified differently in these two places, this option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTLP interprets and solves the linear program as a minimization problem.

PRIMALIN=*SAS-data-set*

PIN=*SAS-data-set*

> specifies the input data set corresponding to the primal solution that is required for warm starting the primal and dual simplex solvers. See the section "Data Input and Output" on page 373 for details.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*

> specifies the output data set for the primal solution. This data set contains the primal solution information. See the section "Data Input and Output" on page 373 for details.

SAVE_ONLY_IF_OPTIMAL

> specifies that the PRIMALOUT= and DUALOUT= data sets be saved only if the final solution obtained by the solver at termination is optimal. If the PRIMALOUT= and DUALOUT= options are specified, then by default (that is, omitting the SAVE_ONLY_IF_OPTIMAL option), PROC OPTLP always saves the solutions obtained at termination, regardless of the final status. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets can contain an intermediate solution, if one is available.

Solver Options

IIS=*number | string*

> specifies whether PROC OPTLP attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 10.2 describes the valid values of the IIS= option.

Table 10.2 Values for IIS= Option

number	string	Description
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

> If an IIS is found, information about infeasible constraints or variable bounds can be found in the DUALOUT= and PRIMALOUT= data sets. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 387 for details.

ALGORITHM=*option*

SOLVER=*option*

SOL=*option*

> specifies one of the following LP solvers:

Option	Description
PRIMAL (PS)	Uses primal simplex solver.
DUAL (DS)	Uses dual simplex solver.
NETWORK (NS)	Uses network simplex solver.
INTERIORPOINT (IP)	Uses interior point solver.
CONCURRENT (CON) (experimental)	Uses several different algorithms in parallel.

> The valid abbreviated value for each option is indicated in parentheses. By default, the dual simplex solver is used.

ALGORITHM2=*option*

SOLVER2=*option*

>   specifies one of the following LP solvers if ALGORITHM=NS:

Option	Description
PRIMAL (PS)	Uses primal simplex solver (after network simplex).
DUAL (DS)	Uses dual simplex solver (after network simplex).

>   The valid abbreviated value for each option is indicated in parentheses. By default, the OPTLP procedure decides which algorithm is best to use after calling the network simplex solver on the extracted network.

Presolve Options

PRESOLVER=*number | string*

PRESOL=*number | string*

>   specifies one of the following presolve options:

number	*string*	Description
0	NONE	Disables presolver.
−1	AUTOMATIC	Applies presolver by using default setting.
1	BASIC	Performs basic presolve like removing empty rows, columns, and fixed variables.
2	MODERATE	Performs basic presolve and apply other inexpensive presolve techniques.
3	AGGRESSIVE	Performs moderate presolve and apply other aggressive (but expensive) presolve techniques.

>   The default option is AUTOMATIC (−1). See the section "Presolve" on page 376 for details.

Control Options

FEASTOL=*ε*

>   specifies the feasibility tolerance $\epsilon \in$[1E–9, 1E–4] for determining the feasibility of a variable value. The default value is 1E–6.

LOGFREQ=*k*

PRINTFREQ=*k*

>   specifies that the printing of the solution progress to the iteration log is to occur after every k iterations. The print frequency, k, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

>   The value $k = 0$ disables the printing of the progress of the solution.

>   If the LOGFREQ= option is not specified, then PROC OPTLP displays the iteration log with a dynamic frequency according to the problem size if the primal or dual simplex solver is used, with frequency 10,000 if the network simplex solver is used, or with frequency 1 if the interior point solver is used.

LOGLEVEL=*number | string*

PRINTLEVEL2=*number | string*

    controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. Table 10.6 describes the valid values for this option.

Table 10.6 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turn off all solver-related messages in SAS log.
1	BASIC	Display a solver summary after stopping.
2	MODERATE	Print a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.
3	AGGRESSIVE	Print a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

    The default value is MODERATE.

MAXITER=*k*

    specifies the maximum number of iterations. The value k can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the solver called after network simplex (either primal or dual simplex).

MAXTIME=*t*

    specifies an upper limit of t seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of t can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*ϵ*

    specifies the optimality tolerance $\epsilon \in$ [1E–9, 1E–4] for declaring optimality. The default value is 1E–6.

PRINTLEVEL=0 | 1 | 2

    specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the ODS (Output Delivery System) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then these tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

    For details about the ODS tables created by PROC OPTLP, see the section "ODS Tables" on page 383.

TIMETYPE=*number | string*

    specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTLP_ macro variable in a PROC OPTLP call. Table 10.7 describes the valid values of the TIMETYPE= option.

Table 10.7 Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement. See the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and NODES= options.

If you specify a value greater than 1 for either the NTHREADS= or the NODES= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and NODES= options, the default value of the TIMETYPE= option is CPU.

Simplex Algorithm Options

BASIS=number | string
  specifies the following options for generating an initial basis:

number	string	Description
0	CRASH	Generate an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
1	SLACK	Generate an initial basis by using all slack variables.
2	WARMSTART	Start the primal and dual simplex solvers with a user-specified initial basis. The PRIMALIN= and DUALIN= data sets are required to specify an initial basis.

The default option for the primal simplex solver is CRASH (0). The default option for the dual simplex solver is SLACK(1). For network simplex, this option has no effect.

PRICETYPE=number | string
  specifies one of the following pricing strategies for the primal and dual simplex solvers:

number	string	Description
0	HYBRID	Use a hybrid of Devex and steepest-edge pricing strategies. Available for the primal simplex solver only.
1	PARTIAL	Use Dantzig's rule on a queue of decision variables. Optionally, you can specify QUEUESIZE=. Available for the primal simplex solver only.
2	FULL	Use Dantzig's rule on all decision variables.
3	DEVEX	Use Devex pricing strategy.
4	STEEPESTEDGE	Use steepest-edge pricing strategy.

The default pricing strategy for the primal simplex solver is HYBRID (0) and for the dual simplex solver is STEEPESTEDGE (4). For the network simplex solver, this option applies only to the solver

specified by the ALGORITHM2= option. See the section "Pricing Strategies for the Primal and Dual Simplex Solvers" on page 377 for details.

QUEUESIZE=k

specifies the queue size $k \in [1, n]$, where n is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

SCALE=*number | string*

specifies one of the following scaling options:

number	string	Description
0	NONE	Disable scaling.
–1	AUTOMATIC	Automatically apply scaling procedure if necessary.

The default option is AUTOMATIC (–1).

Interior Point Algorithm Options

CROSSOVER=*number | string*

specifies whether to convert the interior point solution to a basic simplex solution. If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution.

number	string	Description
0	OFF	Do not convert the interior point solution to a basic simplex solution.
1	ON	Convert the interior point solution to a basic simplex solution.

The default value of the CROSSOVER= option is OFF.

STOP_DG=δ

specifies the desired relative duality gap $\delta \in [1E–9, 1E–4]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 378 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation $\beta \in [1E–9, 1E–4]$. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 378 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation $\alpha \in [1E–9, 1E–4]$. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 378 for details.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTLP procedure:

DECOMP < *options* > **;**

DECOMP_MASTER < *options* > **;**

DECOMP_SUBPROB < *options* > **;**

For more information about these statements, see Chapter 13, "The Decomposition Algorithm."

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for single-machine mode and distributed mode, and requests detailed performance results of the OPTLP procedure.

With the PERFORMANCE statement, you can also control whether the OPTLP procedure executes in single-machine or distribute mode. The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics."

For the OPTLP procedure, the decomposition algorithm, interior point algorithm, and concurrent LP algorithm can be run in single-machine mode. Only the decomposition algorithm can be run in distributed mode. The decomposition algorithm and concurrent LP algorithm support both the deterministic and nondeterministic modes. The interior point algorithm only supports the deterministic mode.

NOTE: Distributed mode requires SAS High-Performance Optimization.s

Details: OPTLP Procedure

Data Input and Output

This subsection describes the PRIMALIN= and DUALIN= data sets required to warm start the primal and dual simplex solvers, and the PRIMALOUT= and DUALOUT= output data sets.

Definitions of Variables in the PRIMALIN= Data Set

The PRIMALIN= data set has two required variables defined as follows:

VAR
> specifies the name of the decision variable.

STATUS
> specifies the status of the decision variable. It can take one of the following values:

> B   basic variable

> L   nonbasic variable at its lower bound

> U   nonbasic variable at its upper bound

> F   free variable

> A   newly added variable in the modified LP model when using the BASIS=WARMSTART option

> **NOTE:** The PRIMALIN= data set is created from the PRIMALOUT= data set obtained from a previous "normal" run of PROC OPTLP—i.e., using only the DATA= data set as the input.

Definitions of Variables in the DUALIN= Data Set

The DUALIN= data set also has two required variables defined as follows:

ROW
> specifies the name of the constraint.

STATUS
> specifies the status of the slack variable for a given constraint. It can take one of the following values:

> B   basic variable

> L   nonbasic variable at its lower bound

> U   nonbasic variable at its upper bound

> F   free variable

> A   newly added variable in the modified LP model when using the BASIS=WARMSTART option

> **NOTE:** The DUALIN= data set is created from the DUALOUT= data set obtained from a previous "normal" run of PROC OPTLP—i.e., using only the DATA= data set as the input.

Definitions of Variables in the PRIMALOUT= Data Set

The PRIMALOUT= data set contains the primal solution to the LP model; each observation corresponds to a variable of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data set can contain an intermediate solution, if one is available. See Example 10.1 for an example of the PRIMALOUT= data set. The variables in the data set have the following names and meanings.

_OBJ_ID_

specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

NOTE: PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_

specifies the name of the variable that contains the right-hand-side value of each constraint.

VAR

specifies the name of the decision variable.

TYPE

specifies the type of the decision variable. _TYPE_ can take one of the following values:

N   nonnegative

D   bounded (with both lower and upper bound)

F   free

X   fixed

O   other (with either lower or upper bound)

OBJCOEF

specifies the coefficient of the decision variable in the objective function.

LBOUND

specifies the lower bound on the decision variable.

UBOUND

specifies the upper bound on the decision variable.

VALUE

specifies the value of the decision variable.

STATUS

specifies the status of the decision variable. _STATUS_ can take one of the following values:

B   basic variable

L   nonbasic variable at its lower bound

U   nonbasic variable at its upper bound

F   free variable

S   superbasic variable (a nonbasic variable with a value strictly between its bounds)

I   LP model infeasible (all decision variables have _STATUS_ equal to I)

For the interior point solver with IIS= OFF, _STATUS_ is blank.

The following values can appear only if IIS= ON. See the section "Irreducible Infeasible Set" on page 387 for details.

I_L   the lower bound of the variable is violated

I_U   the upper bound of the variable is violated

I_F   the fixed bound of the variable is violated

_R_COST_

specifies the reduced cost of the decision variable, which is the amount by which the objective function is increased per unit increase in the decision variable. The reduced cost associated with the ith variable is the ith entry of the following vector:

$$\left(\mathbf{c}^{\mathrm{T}} - \mathbf{c}_B^{\mathrm{T}} \mathbf{B}^{-1} \mathbf{A}\right)$$

where $\mathbf{B} \in \mathbb{R}^{m \times m}$ denotes the basis (matrix composed of *basic* columns of the constraints matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$), $\mathbf{c} \in \mathbb{R}^n$ is the vector of objective function coefficients, and $\mathbf{c}_B \in \mathbb{R}^m$ is the vector of objective coefficients of the variables in the basis.

Definitions of Variables in the DUALOUT= Data Set

The DUALOUT= data set contains the dual solution to the LP model; each observation corresponds to a constraint of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data set can contain an intermediate solution, if one is available. Information about the objective rows of the LP problems is not included. See Example 10.1 for an example of the DUALOUT= data set. The variables in the data set have the following names and meanings.

_OBJ_ID_

specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

NOTE: PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_

specifies the name of the variable that contains the right-hand-side value of each constraint.

ROW

specifies the name of the constraint.

TYPE

specifies the type of the constraint. _TYPE_ can take one of the following values:

L   "less than or equals" constraint

E   equality constraint

G   "greater than or equals" constraint

R   ranged constraint (both "less than or equals" and "greater than or equals")

RHS
> specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
> specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
> specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

VALUE
> specifies the value of the dual variable associated with the constraint.

STATUS
> specifies the status of the slack variable for the constraint. _STATUS_ can take one of the following values:

> B   basic variable

> L   nonbasic variable at its lower bound

> U   nonbasic variable at its upper bound

> F   free variable

> S   superbasic variable (a nonbasic variable with a value strictly between its bounds)

> I   LP model infeasible (all decision variables have _STATUS_ equal to I)

> The following values can appear only if option IIS= ON. See the section "Irreducible Infeasible Set" on page 387 for details.

> I_L   the "GE" ($\geq$) condition of the constraint is violated

> I_U   the "LE" ($\leq$) condition of the constraint is violated

> I_F   the "EQ" ($=$) condition of the constraint is violated

ACTIVITY
> specifies the left-hand-side value of a constraint. In other words, the value of _ACTIVITY_ for the ith constraint would be equal to $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}$, where $\mathbf{a}_i$ refers to the ith row of the constraints matrix and $\mathbf{x}$ denotes the vector of current decision variable values.

Presolve

Presolve in PROC OPTLP uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.

Pricing Strategies for the Primal and Dual Simplex Solvers

Several pricing strategies for the primal and dual simplex solvers are available. Pricing strategies determine which variable enters the basis at each simplex pivot. They can be controlled by specifying the PRICETYPE= option.

The primal simplex solver has the following five pricing strategies:

PARTIAL uses Dantzig's most violated reduced cost rule (Dantzig 1963). It scans a queue of decision variables and selects the variable with the most violated reduced cost as the entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.

FULL uses Dantzig's most violated reduced cost rule. It compares the reduced costs of all decision variables and selects the variable with the most violated reduced cost as the entering variable.

DEVEX implements the Devex pricing strategy developed by Harris (1973).

STEEPESTEDGE uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).

HYBRID uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex solver has only three pricing strategies available: FULL, DEVEX, and STEEPESTEDGE.

Warm Start for the Primal and Dual Simplex Solvers

You can warm start the primal and dual simplex solvers by specifying the option BASIS=WARMSTART. Additionally you need to specify the PRIMALIN= and DUALIN= data sets. The primal and dual simplex solvers start with the basis thus provided. If the given basis cannot form a valid basis, the solvers use the basis generated using their *crash* techniques.

After an LP model is solved using the primal and dual simplex solvers, the BASIS=WARMSTART option enables you to perform sensitivity analysis such as modifying the objective function, changing the right-hand sides of the constraints, adding or deleting constraints or decision variables, and combinations of these cases. A faster solution to such a modified LP model can be obtained by starting with the basis in the optimal solution to the original LP model. This can be done by using the BASIS=WARMSTART option, modifying the DATA= input data set, and specifying the PRIMALIN= and DUALIN= data sets. Example 10.4 and Example 10.5 illustrate how to reoptimize an LP problem with a modified objective function and a modified right-hand side by using this technique. Example 10.6 shows how to reoptimize an LP problem after adding a new constraint.

The network simplex solver ignores the option BASIS=WARMSTART.

CAUTION: Since the presolver uses the objective function and/or right-hand-side information, the basis provided by you might not be valid for the presolved model. It is therefore recommended that you turn the PRESOLVER= option off when using BASIS=WARMSTART.

The Network Simplex Algorithm

The network simplex solver in PROC OPTLP attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.

2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.

3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex solver on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where N denotes the nodes and A denotes the arcs of the graph. The decision variable x_{ij} denotes the amount of flow sent between node i and node j. The cost per unit of flow on the arcs is designated by c_{ij}, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at each node is designated as b_i, where $b_i > 0$ denotes a supply node and $b_i < 0$ denotes a demand node. The corresponding linear programming problem is as follows:

$$
\begin{array}{rlcll}
\min & \sum_{(i,j) \in A} c_{ij} x_{ij} & & & \\
\text{subject to} & \sum_{(i,j) \in A} x_{ij} - \sum_{(j,i) \in A} x_{ji} & = & b_i & \forall i \in N \\
& x_{ij} & \leq & u_{ij} & \forall (i, j) \in A \\
& x_{ij} & \geq & l_{ij} & \forall (i, j) \in A
\end{array}
$$

The network simplex algorithm used in PROC OPTLP is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph G is disconnected. In this case, the problem can be decomposed into its weakly connected components and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the solver to use after network simplex. However, you can override this selection with the ALGORITHM2= option.

The network simplex algorithm can be more efficient than the other solvers on problems with a large network substructure. You can view the size of the network structure in the log.

The Interior Point Algorithm

The interior point solver in PROC OPTLP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

$$
\begin{array}{rl}
\min & \mathbf{c}^{\mathrm{T}} \mathbf{x} \\
\text{subject to} & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\
& \mathbf{x} \geq \mathbf{0}
\end{array}
$$

The corresponding dual formulation is as follows:

$$
\begin{array}{rrcl}
\max & \mathbf{b}^{\mathrm{T}}\mathbf{y} \\
\text{subject to} & \mathbf{A}^{\mathrm{T}}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\
& & & \mathbf{y} & \geq & \mathbf{0} \\
& & & \mathbf{w} & \geq & \mathbf{0}
\end{array}
$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual formulation makes an important contribution to the certificate of optimality for the primal formulation. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$
\begin{array}{rcll}
\mathbf{A}\mathbf{x} - \mathbf{s} & = & \mathbf{b} & \text{(primal feasibility)} \\
\mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & \text{(dual feasibility)} \\
\mathbf{WXe} & = & \mathbf{0} & \text{(complementarity)} \\
\mathbf{SYe} & = & \mathbf{0} & \text{(complementarity)} \\
\mathbf{x, y, w, s} & \geq & \mathbf{0}
\end{array}
$$

where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding x, y, w, and s on the main diagonal and zero elsewhere, as in the following example:

$$
\mathbf{X} \equiv \begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

$$
\begin{bmatrix}
\mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\
\mathbf{A}^{\mathrm{T}} & -\mathbf{X}^{-1}\mathbf{W}
\end{bmatrix}
\begin{bmatrix}
\Delta\mathbf{y} \\
\Delta\mathbf{x}
\end{bmatrix}
=
\begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTLP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point solver is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector v:

- relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{A}\mathbf{x} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{c} - \mathbf{A}^{\mathsf{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{c}^{\mathsf{T}}\mathbf{x} - \mathbf{b}^{\mathsf{T}}\mathbf{y}|}{|\mathbf{c}^{\mathsf{T}}\mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Iteration Log for the Primal and Dual Simplex Solvers

The primal and dual simplex solvers implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When the LOGFREQ= option has a value of 1, the following information is printed in the iteration log:

Algorithm	indicates which simplex method is running by printing the letter P (primal) or D (dual).
Phase	indicates whether the solver is in phase I or phase II of the simplex method.
Iteration	indicates the iteration number.
Objective Value	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)". If the entering nonbasic variable has distinct, finite lower and upper bounds, then a "bound swap" can take place in the primal simplex method.
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)". The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value greater than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.

The behavior of objective values in the iteration log depends on both the current phase and the chosen solver. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is

found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method, phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality for the perturbed problem, PROC OPTLP solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

Iteration Log for the Network Simplex Solver

After finding the embedded network and formulating the appropriate relaxation, the network simplex solver uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log shows the progress of the simplex algorithm. The following information is displayed in the iteration log:

Iteration   indicates the iteration number.

PrimalObj   indicates the primal objective value of the current solution.

Primal Infeas   indicates the maximum primal infeasibility of the current solution.

Time   indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the LOGFREQ= option. The default value of the LOGFREQ= option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

Component   indicates the component number being processed.

Nodes   indicates the number of nodes in this component.

Arcs   indicates the number of arcs in this component.

Iterations   indicates the number of simplex iterations needed to solve this component.

Time   indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the LOGFREQ= option. In this case, the default value of the LOGFREQ= option is determined by the size of the network.

The LOGLEVEL= option adjusts the amount of detail shown. By default, LOGLEVEL= is set to MODER-ATE and reports as described previously. If set to NONE, no information is shown. If set to BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If set to AGGRESSIVE, in the case of one component, the log displays as described previously; in the case of multiple components, for each component, a separate simplex iteration log is displayed.

Iteration Log for the Interior Point Solver

The interior point solver implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter   indicates the iteration number.

Complement   indicates the (absolute) complementarity.

Duality Gap   indicates the (relative) duality gap.

Primal Infeas   indicates the (relative) primal infeasibility measure.

Bound Infeas   indicates the (relative) bound infeasibility measure.

Dual Infeas   indicates the (relative) dual infeasibility measure.

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the MAXITER= or MAXTIME= options. If the complementarity or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Iteration Log for the Crossover Algorithm

The crossover algorithm takes an optimal solution from the interior point solver and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When LOGFREQ=1, the following information is printed in the iteration log:

Phase   indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.

Iteration   indicates the iteration number.

Objective Value   indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.

Time   indicates the time elapsed (in seconds) since the beginning of the crossover algorithm.

Entering Variable   indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)."

Leaving Variable   indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)."

When you omit the LOGFREQ= option or specify a value greater than 1, only the phase, iteration, objective value, and time information is printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.

Concurrent LP (Experimental)

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The OPTLP procedure automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the procedure runs all four standard algorithms. When the first algorithm ends, the procedure returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of DETERMINISTIC for the PARALLELMODE= option in the PERFORMANCE statement, the algorithm for which the results are returned is not necessarily the one that finished first. The OPTLP procedure deterministically selects the algorithm for which the results are returned. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. See the section "Iteration Log for the Primal and Dual Simplex Solvers" on page 380 for more information about this iteration log. Upon termination, the procedure displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the OPTLP procedure displays the iteration logs for all algorithms that are run concurrently.

If you specify PRINTLEVEL=2 and ALGORITHM=CON, the OPTLP procedure produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

ODS Tables

PROC OPTLP creates three Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input LP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. For more information, see the section "Problem Statistics" on page 386. If you specify PRINTLEVEL=2 and ALGORITHM=CON, the ConcurrentSummary table is produced. This table contains solution status information for all algorithms that are run concurrently. For more information, see the section "Concurrent LP (Experimental)" on page 383.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 10.12 lists all the ODS tables that can be produced by the OPTLP procedure, along with the statement and option specifications required to produce each table.

Table 10.12   ODS Tables Produced by PROC OPTLP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input LP problem	PROC OPTLP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTLP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTLP	PRINTLEVEL=2

Table 10.12 (continued)

ODS Table Name	Description	Statement	Option
ConcurrentSummary	Summary of the solution status for all algorithms run concurrently	PROC OPTLP	PRINTLEVEL=2, ALGORITHM=CON
PerformanceInfo	List of performance options and their values	PROC OPTLP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical output of PROC OPTLP is shown in Figure 10.2.

Figure 10.2 Typical OPTLP Output

```
                       The OPTLP Procedure

                         Problem Summary

             Problem Name                     ADLITTLE
             Objective Sense              Minimization
             Objective Function                .Z....
             RHS                             ZZZZ0001

             Number of Variables                   97
             Bounded Above                          0
             Bounded Below                         97
             Bounded Above and Below                0
             Free                                   0
             Fixed                                  0

             Number of Constraints                 56
             LE  (<=)                              40
             EQ  (=)                               15
             GE  (>=)                               1
             Range                                  0

             Constraint Coefficients              383


                     Performance Information

              Execution Mode        Single-Machine
              Number of Threads                  4
```

Figure 10.2 *continued*

```
                       Solution Summary

              Solver                           LP
              Algorithm             Dual Simplex
              Objective Function          .Z....
              Solution Status            Optimal
              Objective Value       225494.96316

              Primal Infeasibility    2.273737E-13
              Dual Infeasibility      1.847411E-13
              Bound Infeasibility                0

              Iterations                        77
              Presolve Time                   0.00
              Solution Time                   0.06
```

You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTLP runs. The output data sets corresponding to the preceding output are shown in Figure 10.3, where you can also find (at the row following the heading of each data set in display) the variable names that are used in the table definition (template) of each table.

Figure 10.3 ODS Output Data Sets

```
                       Problem Summary

    Obs    Label1                    cValue1              nValue1

     1     Problem Name              ADLITTLE                   .
     2     Objective Sense           Minimization               .
     3     Objective Function        .Z....                     .
     4     RHS                       ZZZZ0001                   .
     5                                                          .
     6     Number of Variables       97                 97.000000
     7     Bounded Above             0                          0
     8     Bounded Below             97                 97.000000
     9     Bounded Above and Below   0                          0
    10     Free                      0                          0
    11     Fixed                     0                          0
    12                                                          .
    13     Number of Constraints     56                 56.000000
    14     LE  (<=)                  40                 40.000000
    15     EQ  (=)                   15                 15.000000
    16     GE  (>=)                  1                   1.000000
    17     Range                     0                          0
    18                                                          .
    19     Constraint Coefficients   383               383.000000
```

Figure 10.3 *continued*

```
                        Solution Summary

        Obs    Label1                    cValue1              nValue1

         1     Solver                    LP                         .
         2     Algorithm                 Dual Simplex               .
         3     Objective Function        .Z....                     .
         4     Solution Status           Optimal                    .
         5     Objective Value           225494.96316          225495
         6                                                          .
         7     Primal Infeasibility      2.273737E-13    2.273737E-13
         8     Dual Infeasibility        1.847411E-13    1.847411E-13
         9     Bound Infeasibility       0                          0
        10                                                          .
        11     Iterations                77                  77.000000
        12     Presolve Time                        0.00           0
        13     Solution Time                        0.06    0.062401
```

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTLP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 10.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 10.4 ODS Table ProblemStatistics

```
                        The OPTLP Procedure

                        Problem Statistics

        Number of Constraint Matrix Nonzeros            8
        Maximum Constraint Matrix Coefficient           3
        Minimum Constraint Matrix Coefficient           1
        Average Constraint Matrix Coefficient       1.875

        Number of Objective Nonzeros                    3
        Maximum Objective Coefficient                   4
        Minimum Objective Coefficient                   2
        Average Objective Coefficient                   3

        Number of RHS Nonzeros                          3
        Maximum RHS                                     7
        Minimum RHS                                     4
        Average RHS                           5.3333333333

        Maximum Number of Nonzeros per Column           3
        Minimum Number of Nonzeros per Column           2
        Average Number of Nonzeros per Column           2

        Maximum Number of Nonzeros per Row              3
        Minimum Number of Nonzeros per Row              2
        Average Number of Nonzeros per Row              2
```

Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help to isolate the structural infeasibility in an LP.

The presolver in the OPTLP procedure can detect infeasibility, but it only identifies the variable bound or constraint that triggers the infeasibility.

The IIS=ON option directs the OPTLP procedure to search for an IIS in a given LP. The presolver is not applied to the problem during the IIS search. If the OPTLP procedure detects an IIS, it first outputs the IIS to the data sets specified by the PRIMALOUT= and DUALOUT= options, and then it stops. The number of iterations that are reported in the macro variable and the ODS table is the total number of simplex iterations. This includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the _STATUS_ variables in the output data sets. (See the section "Data Input and Output" on page 373 for more information.) For constraints, a status of "I_L", "I_U", or "I_F" indicates, respectively, the "GE" ($\geq$), "LE" ($\leq$), or "EQ" ($=$) condition is violated. For range constraints, a status of "I_L" or "I_U" indicates, respectively, that the lower or upper bound of the constraint is violated. For variables, a status of "I_L", "I_U", or "I_F" indicates, respectively, the lower, upper, or fixed bound of the variable is violated. From this information, you can identify names of the constraints (variables) in the IIS as well as the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount removes the infeasibility; however, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to ∞ or $-\infty$. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can rerun the LP solver with IIS=ON specified after removing the infeasibility from an IIS. Repeat this process until the LP solver no longer detects an IIS. The resulting problem is feasible. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

Changing different constraints and bounds can require considerably different changes to the MPS-format SAS data set. For example, if you used the default lower bound of 0 for a variable but you want to relax the lower bound to $-\infty$, you might need to add a LB row to the BOUNDS section of the data set. For more information about changing variable and constraint bounds, see Chapter 15, "The MPS-Format SAS Data Set."

The IIS= option in PROC OPTLP uses two different methods to identify an IIS. Based on the result of the initial solve, the *sensitivity filter* removes several constraints and variable bounds at once while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS. Following that, the *deletion filter* removes each remaining constraint and variable bound one by one to check which of them are needed to get an infeasible system. This second phase is more time consuming, but it ensures that the IIS set returned by PROC OPTLP is indeed irreducible. The progress of the deletion filter is reported at regular intervals. Occasionally, the sensitivity filter might be called again during the deletion filter to improve performance.

See Example 10.7 for an example demonstrating the use of the IIS= option in locating and removing infeasibilities.

Macro Variable _OROPTLP_

The OPTLP procedure defines a macro variable named _OROPTLP_. This variable contains a character string that indicates the status of the OPTLP procedure upon termination. The various terms of the variable are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

PS	The primal simplex algorithm produced the solution data.
DS	The dual simplex algorithm produced the solution data.
NS	The network simplex algorithm produced the solution data.
IP	The interior point algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or preprocessing.
FEASIBLE	The problem is feasible.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY

    indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point solver, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY

    indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point solver, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY

    indicates, for the primal simplex and dual simplex solvers, the maximum (absolute) violation of the lower or upper bounds (or both) by the primal solution. For the interior point solver, this term indicates the relative violation of the lower or upper bounds (or both) by the primal solution.

DUALITY_GAP

    indicates the (relative) duality gap. This term appears only if the interior point algorithm is used.

COMPLEMENTARITY

    indicates the (absolute) complementarity. This term appears only if the interior point algorithm is used.

ITERATIONS

    indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2

    indicates the number of simplex iterations performed by the secondary solver. In network simplex, the secondary solver is selected automatically, unless a value has been specified for the ALGORITHM2= option. When crossover is enabled, the secondary solver is selected automatically. This term appears only if the network simplex solver is used or if crossover is enabled.

PRESOLVE_TIME

    indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME

    indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTLP_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: OPTLP Procedure

Example 10.1: Oil Refinery Problem

Consider an oil refinery scenario. 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 (a_l), Arabian heavy (a_h), and Brega (br). These crudes are distilled into light naphtha (na_l), intermediate naphtha (na_i), and heating oil (h_o). These in turn are blended into two types of jet fuel. Jet fuel j_1 is made up of 30% intermediate naphtha and 70% heating oil, and jet fuel j_2 is made up of 20% light naphtha and 80% heating oil. What amounts of the three crudes maximize the profit from producing jet fuel (j_1, j_2)? This problem can be formulated as the following linear program:

$$
\begin{array}{lrcrcrcl}
\max & -175\,a\_l - 165\,a\_h - 205\,br + 350\,j\_1 + 350\,j\_2 \\
\text{subject to} \\
(\text{napha\_l}) & 0.035\,a\_l & + & 0.03\,a\_h & + & 0.045\,br & = & na\_l \\
(\text{napha\_i}) & 0.1\,a\_l & + & 0.075\,a\_h & + & 0.135\,br & = & na\_i \\
(\text{htg\_oil}) & 0.39\,a\_l & + & 0.3\,a\_h & + & 0.43\,br & = & h\_o \\
(\text{blend1}) & & & 0.3\,j\_1 & & & \leq & na\_i \\
(\text{blend2}) & & & & & 0.2\,j\_2 & \leq & na\_l \\
(\text{blend3}) & & & 0.7\,j\_1 & + & 0.8\,j\_2 & \leq & h\_o \\
& a\_l & & & & & \leq & 110 \\
& & & a\_h & & & \leq & 165 \\
& & & & & br & \leq & 80 \\
& \multicolumn{5}{c}{a\_l,\ a\_h,\ br,\ na\_1,\ na\_i,\ h\_o,\ j\_1,\ j\_2} & \geq & 0
\end{array}
$$

The constraints "blend1" and "blend2" ensure that j_1 and j_2 are made with the specified amounts of na_i and na_l, respectively. The constraint "blend3" is actually the reduced form of the following constraints:

$$
\begin{array}{rcrcl}
h\_o1 & & & \geq & 0.7\,j\_1 \\
& & h\_o2 & \geq & 0.8\,j\_2 \\
h\_o1 & + & h\_o2 & \leq & h\_o
\end{array}
$$

where h_o1 and h_o2 are dummy variables.

You can use the following SAS code to create the input data set ex1:

```
data ex1;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME             .      EX1        .     .      .
ROWS             .      .          .     .      .
 N         profit       .          .     .      .
 E         napha_l      .          .     .      .
 E         napha_i      .          .     .      .
 E         htg_oil      .          .     .      .
 L         blend1       .          .     .      .
```

```
   L          blend2       .        .        .        .
   L          blend3       .        .        .        .
COLUMNS       .            .        .        .        .
   .          a_l          profit   -175     napha_l  .035
   .          a_l          napha_i  .100     htg_oil  .390
   .          a_h          profit   -165     napha_l  .030
   .          a_h          napha_i  .075     htg_oil  .300
   .          br           profit   -205     napha_l  .045
   .          br           napha_i  .135     htg_oil  .430
   .          na_l         napha_l  -1       blend2   -1
   .          na_i         napha_i  -1       blend1   -1
   .          h_o          htg_oil  -1       blend3   -1
   .          j_1          profit   350      blend1   .3
   .          j_1          blend3   .7       .        .
   .          j_2          profit   350      blend2   .2
   .          j_2          blend3   .8       .        .
BOUNDS        .            .        .        .        .
UP            .            a_l      110      .        .
UP            .            a_h      165      .        .
UP            .            br       80       .        .
ENDATA        .            .        .        .        .
;
```

You can use the following call to PROC OPTLP to solve the LP problem:

```
proc optlp data=ex1
   objsense  = max
   algorithm = primal
   primalout = ex1pout
   dualout   = ex1dout
   logfreq   = 1;
run;
%put &_OROPTLP_;
```

Note that the OBJSENSE=MAX option is used to indicate that the objective function is to be maximized.

The primal and dual solutions are displayed in Output 10.1.1.

Output 10.1.1 Example 1: Primal and Dual Solution Output

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient
1	profit		a_l	D	−175
2	profit		a_h	D	−165
3	profit		br	D	−205
4	profit		na_l	N	0
5	profit		na_i	N	0
6	profit		h_o	N	0
7	profit		j_1	N	350
8	profit		j_2	N	350

Obs	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	0	110	110.000	U	10.2083
2	0	165	0.000	L	−22.8125
3	0	80	80.000	U	2.8125
4	0	1.7977E308	7.450	B	0.0000
5	0	1.7977E308	21.800	B	0.0000
6	0	1.7977E308	77.300	B	0.0000
7	0	1.7977E308	72.667	B	0.0000
8	0	1.7977E308	33.042	B	0.0000

Dual Solution

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound
1	profit		napha_l	E	0	.
2	profit		napha_i	E	0	.
3	profit		htg_oil	E	0	.
4	profit		blend1	L	0	.
5	profit		blend2	L	0	.
6	profit		blend3	L	0	.

Obs	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	.	0.000	L	0.00000
2	.	−145.833	U	0.00000
3	.	−437.500	U	0.00000
4	.	145.833	L	0.00000
5	.	0.000	B	−0.84167
6	.	437.500	L	0.00000

The progress of the solution is printed to the log as follows.

Output 10.1.2 Log: Solution Progress

```
NOTE: The problem EX1 has 8 variables (0 free, 0 fixed).
NOTE: The problem has 6 constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 19 constraint coefficients.
WARNING: The objective sense has been changed to maximization.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 3 variables and 3 constraints.
NOTE: The LP presolver removed 6 constraint coefficients.
NOTE: The presolved problem has 5 variables, 3 constraints, and 13 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                        Objective              Entering      Leaving
      Phase Iteration     Value        Time    Variable      Variable
        P 1         1   0.000000E+00      0
        P 2         2   0.000000E+00      0        j_1        blend1 (S)
        P 2         3   1.405640E-01      0        j_2        blend3 (S)
        P 2         4   1.454487E-01      0        a_l        blend2 (S)
        P 2         5   2.379819E-01      0         br          a_l
        P 2         6   1.202394E+03      0     blend2 (S)       br
        P 2         7   1.348074E+03      0
        D 2         8   1.347917E+03      0
        D 2         9   1.347917E+03      0
NOTE: Optimal.
NOTE: Objective = 1347.9166667.
NOTE: The Primal Simplex solve time is 0.03 seconds.
NOTE: The data set WORK.EX1POUT has 8 observations and 10 variables.
NOTE: The data set WORK.EX1DOUT has 6 observations and 10 variables.
```

Note that the %put statement immediately after the OPTLP procedure prints value of the macro variable _OROPTLP_ to the log as follows.

Output 10.1.3 Log: Value of the Macro Variable _OROPTLP_

```
STATUS=OK    ALGORITHM=PS    SOLUTION_STATUS=OPTIMAL    OBJECTIVE=1347.9166667
PRIMAL_INFEASIBILITY=3.552714E-15    DUAL_INFEASIBILITY=0
BOUND_INFEASIBILITY=0    ITERATIONS=9    PRESOLVE_TIME=0.00    SOLUTION_TIME=0.03
```

The value briefly summarizes the status of the OPTLP procedure upon termination.

Example 10.2: Using the Interior Point Solver

You can also solve the oil refinery problem described in Example 10.1 by using the interior point solver. You can create the input data set from an external MPS-format flat file by using the SAS macro %MPS2SASD or SAS DATA step code, both of which are described in "Getting Started: OPTLP Procedure" on page 362. You can use the following SAS code to solve the problem:

```
proc optlp data=ex1
  objsense  = max
  algorithm = ip
  primalout = ex1ipout
  dualout   = ex1idout
  logfreq   = 1;
run;
```

The optimal solution is displayed in Output 10.2.1.

Output 10.2.1 Interior Point Solver: Primal Solution Output

```
                           Primal Solution

              Objective
              Function     RHS     Variable    Variable    Objective
       Obs       ID        ID        Name        Type      Coefficient

        1      profit                a_l          D           -175
        2      profit                a_h          D           -165
        3      profit                br           D           -205
        4      profit                na_l         N              0
        5      profit                na_i         N              0
        6      profit                h_o          N              0
        7      profit                j_1          N            350
        8      profit                j_2          N            350


              Lower            Upper     Variable    Variable     Reduced
       Obs    Bound            Bound       Value      Status        Cost

        1       0                110      110.000                 10.2083
        2       0                165        0.000                -22.8125
        3       0                 80       80.000                  2.8125
        4       0         1.7977E308        7.450                  0.0000
        5       0         1.7977E308       21.800                  0.0000
        6       0         1.7977E308       77.300                  0.0000
        7       0         1.7977E308       72.667                  0.0000
        8       0         1.7977E308       33.042                  0.0000
```

The iteration log is displayed in Output 10.2.2.

Output 10.2.2 Log: Solution Progress

```
NOTE: The problem EX1 has 8 variables (0 free, 0 fixed).
NOTE: The problem has 6 constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 19 constraint coefficients.
WARNING: The objective sense has been changed to maximization.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 3 variables and 3 constraints.
NOTE: The LP presolver removed 6 constraint coefficients.
NOTE: The presolved problem has 5 variables, 3 constraints, and 13 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                        Primal         Bound          Dual
      Iter   Complement  Duality Gap     Infeas         Infeas        Infeas
         0   4.06515E+01  2.05269E+00  1.13742E-14  0.00000E+00  2.70189E-01
         1   6.98840E+00  3.06315E+00  6.77873E-15  0.00000E+00  5.33713E-02
         2   5.99555E+00  3.57653E-01  2.93104E-14  0.00000E+00  4.88028E-02
         3   1.39762E+00  1.01631E-01  2.62511E-14  5.55800E-17  1.04677E-02
         4   1.03056E+00  6.83923E-02  4.42791E-14  1.10891E-16  5.86194E-03
         5   2.52016E-02  2.98995E-04  3.82469E-14  1.13953E-16  2.08475E-04
         6   2.52303E-04  3.02383E-06  3.45164E-14  2.49422E-16  2.08475E-06
         7   6.82704E-04  3.62810E-08  7.84605E-15  3.96766E-17  4.54026E-06
         8   6.82704E-06  3.62807E-10  2.49698E-15  2.54085E-16  4.54026E-08
         9   0.00000E+00  1.17992E-15  3.55271E-15  0.00000E+00  0.00000E+00
NOTE: Conditional optimal.
NOTE: Objective = 1347.9166667.
NOTE: The Interior Point solve time is 0.01 seconds.
NOTE: The data set WORK.EX1IPOUT has 8 observations and 10 variables.
NOTE: The data set WORK.EX1IDOUT has 6 observations and 10 variables.
```

Example 10.3: The Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 10.13.

Table 10.13 Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories	90	120	106	97	130	180

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can use the following SAS code to create the MPS-format input data set:

```
data ex3;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .          EX3      .      .          .
ROWS          .          .        .      .          .
 N        diet       .        .      .          .
 G        calories   .        .      .          .
 L        protein    .        .      .          .
 G        fat        .        .      .          .
 G        carbs      .        .      .          .
COLUMNS       .          .        .      .          .
 .        br         diet     2      calories   90
 .        br         protein  4      fat        1
 .        br         carbs    15     .          .
 .        mi         diet     3.5    calories   120
 .        mi         protein  8      fat        5
 .        mi         carbs    11.7   .          .
 .        ch         diet     8      calories   106
 .        ch         protein  7      fat        9
 .        ch         carbs    .4     .          .
 .        po         diet     1.5    calories   97
 .        po         protein  1.3    fat        .1
 .        po         carbs    22.6   .          .
 .        fi         diet     11     calories   130
 .        fi         protein  8      fat        7
 .        fi         carbs    0      .          .
 .        yo         diet     1      calories   180
 .        yo         protein  9.2    fat        1
 .        yo         carbs    17     .          .
RHS           .          .        .      .          .
 .            .          calories 300    protein    10
 .            .          fat      8      carbs      10
BOUNDS        .          .        .      .          .
UP            .          mi       1      .          .
LO            .          fi       .5     .          .
ENDATA        .          .        .      .          .
;
```

You can solve the diet problem by using PROC OPTLP as follows:

```
proc optlp data=ex3
   presolver = none
   algorithm = ps
   primalout = ex3pout
   dualout   = ex3dout
   logfreq   = 1;
run;
```

The solution summary and the optimal primal solution are displayed in Output 10.3.1.

Output 10.3.1 Diet Problem: Solution Summary and Optimal Primal Solution

```
                          Solution Summary

      Obs    Label1                   cValue1              nValue1

       1     Solver                   LP                        .
       2     Algorithm                Primal Simplex            .
       3     Objective Function       diet                      .
       4     Solution Status          Optimal                   .
       5     Objective Value          12.081337881        12.081338
       6                                                        .
       7     Primal Infeasibility     0                         0
       8     Dual Infeasibility       0                         0
       9     Bound Infeasibility      0                         0
      10                                                        .
      11     Iterations               9                  9.000000
      12     Presolve Time                    0.00              0
      13     Solution Time                    0.03       0.031200


                          Primal Solution

            Objective
            Function     RHS     Variable    Variable    Objective
      Obs      ID        ID        Name        Type      Coefficient

       1      diet                  br          N           2.0
       2      diet                  mi          D           3.5
       3      diet                  ch          N           8.0
       4      diet                  po          N           1.5
       5      diet                  fi          O          11.0
       6      diet                  yo          N           1.0


            Lower           Upper     Variable    Variable    Reduced
      Obs   Bound           Bound      Value      Status       Cost

       1    0.0       1.7977E308     0.00000        L        1.19066
       2    0.0                1     0.05360        B        0.00000
       3    0.0       1.7977E308     0.44950        B        0.00000
       4    0.0       1.7977E308     1.86517        B        0.00000
       5    0.5       1.7977E308     0.50000        L        5.15641
       6    0.0       1.7977E308     0.00000        L        1.10849
```

The cost of the optimal diet is 12.08 units.

Example 10.4: Reoptimizing after Modifying the Objective Function

Using the diet problem described in Example 10.3, this example illustrates how to reoptimize an LP problem after modifying the objective function.

Assume that the optimal solution of the diet problem is found and the optimal solutions are stored in the data sets ex3pout and ex3dout.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. The COLUMNS section in the input data set ex3 is updated (and the data set is saved as ex4) as follows:

```
COLUMNS     .         .        .       .          .
   ...
   .        ch        diet     10      calories   106
   ...
   .        fi        diet     7       calories   130
   ...
RHS         .         .        .       .          .
   ...

ENDATA
;
```

You can use the following DATA step to create the data set ex4:

```
data ex4;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME            .        EX4       .       .          .
ROWS            .        .         .       .          .
 N              diet     .         .       .          .
 G              calories .         .       .          .
 L              protein  .         .       .          .
 G              fat      .         .       .          .
 G              carbs    .         .       .          .
COLUMNS         .        .         .       .          .
 .              br       diet      2       calories   90
 .              br       protein   4       fat        1
 .              br       carbs     15      .           .
 .              mi       diet      3.5     calories   120
 .              mi       protein   8       fat        5
 .              mi       carbs     11.7    .           .
 .              ch       diet      10      calories   106
 .              ch       protein   7       fat        9
 .              ch       carbs     .4      .           .
 .              po       diet      1.5     calories   97
 .              po       protein   1.3     fat        .1
 .              po       carbs     22.6    .           .
 .              fi       diet      7       calories   130
 .              fi       protein   8       fat        7
 .              fi       carbs     0       .           .
```

```
    .           yo          diet      1       calories  180
    .           yo          protein   9.2     fat       1
    .           yo          carbs     17      .         .
    RHS         .           .         .       .         .
    .           .           calories  300     protein   10
    .           .           fat       8       carbs     10
    BOUNDS      .           .         .        .         .
    UP          .           mi        1       .         .
    LO          .           fi        .5      .         .
    ENDATA      .           .         .       .         .
    ;
```

You can use the BASIS=WARMSTART option (and the ex3pout and ex3dout data sets from Example 10.3) in the following call to PROC OPTLP to solve the modified problem:

```
proc optlp data=ex4
   presolver = none
   basis     = warmstart
   primalin  = ex3pout
   dualin    = ex3dout
   algorithm = primal
   primalout = ex4pout
   dualout   = ex4dout
   logfreq   = 1;
run;
```

The following iteration log indicates that it takes the primal simplex solver no extra iterations to solve the modified problem by using BASIS=WARMSTART, since the optimal solution to the LP problem in Example 10.3 remains optimal after the objective function is changed.

Output 10.4.1 Iteration Log

```
NOTE: The problem EX4 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                            Objective               Entering      Leaving
      Phase Iteration         Value          Time   Variable      Variable
        P 2          1     1.098034E+01          0
NOTE: Optimal.
NOTE: Objective = 10.980335514.
NOTE: The Primal Simplex solve time is 0.03 seconds.
NOTE: The data set WORK.EX4POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX4DOUT has 4 observations and 10 variables.
```

Note that the primal simplex solver is preferred because the primal solution to the original LP is still feasible for the modified problem in this case.

Example 10.5: Reoptimizing after Modifying the Right-Hand Side

You can also modify the right-hand side of your problem and use the BASIS=WARMSTART option to obtain an optimal solution more quickly. Since the dual solution to the original LP is still feasible for the modified problem in this case, the dual simplex solver is preferred. This case is illustrated by using the same diet problem as in Example 10.3. Assume that you now need a diet that supplies at least 150 calories. The RHS section in the input data set ex3 is updated (and the data set is saved as ex5) as follows:

```
    ...
 RHS            .        .        .        .        .
 .         .         calories 150    protein   10
 .         .         fat       8     carbs     10
 BOUNDS    .         .        .        .        .
    ...
```

You can use the following DATA step to create the data set ex5:

```
data ex5;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME           .        EX5      .        .        .
ROWS           .        .        .        .        .
 N         diet         .        .        .        .
 G         calories     .        .        .        .
 L         protein      .        .        .        .
 G         fat          .        .        .        .
 G         carbs        .        .        .        .
COLUMNS        .        .        .        .        .
 .         br       diet     2        calories 90
 .         br       protein  4        fat      1
 .         br       carbs    15       .        .
 .         mi       diet     3.5      calories 120
 .         mi       protein  8        fat      5
 .         mi       carbs    11.7     .        .
 .         ch       diet     8        calories 106
 .         ch       protein  7        fat      9
 .         ch       carbs    .4       .        .
 .         po       diet     1.5      calories 97
 .         po       protein  1.3      fat      .1
 .         po       carbs    22.6     .        .
 .         fi       diet     11       calories 130
 .         fi       protein  8        fat      7
 .         fi       carbs    0        .        .
 .         yo       diet     1        calories 180
 .         yo       protein  9.2      fat      1
 .         yo       carbs    17       .        .
RHS            .        .        .        .        .
 .         .        calories 150      protein  10
 .         .        fat      8        carbs    10
BOUNDS         .        .        .        .        .
 UP        .        mi       1        .        .
 LO        .        fi       .5       .        .
```

```
    ENDATA        .         .         .        .            .
    ;
```

You can use the BASIS=WARMSTART option in the following call to PROC OPTLP to solve the modified problem:

```
proc optlp data=ex5
   presolver = none
   basis      = warmstart
   primalin  = ex3pout
   dualin     = ex3dout
   algorithm = dual
   primalout = ex5pout
   dualout    = ex5dout
   logfreq   = 1;
run;
```

Note that the dual simplex solver is preferred because the dual solution to the last solved LP is still feasible for the modified problem in this case.

The following iteration log indicates that it takes the dual simplex solver just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.

Output 10.5.1 Iteration Log

```
NOTE: The problem EX5 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                           Objective              Entering       Leaving
      Phase Iteration         Value        Time    Variable      Variable
        D 2           1     8.813205E+00      0   calories (S)    carbs (S)
        D 2           2     9.174413E+00      0
        D 2           3     9.174413E+00      0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.03 seconds.
NOTE: The data set WORK.EX5POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX5DOUT has 4 observations and 10 variables.
```

Compare this with the following call to PROC OPTLP:

```
proc optlp data=ex5
   presolver = none
   algorithm = dual
   logfreq   = 1;
run;
```

This call to PROC OPTLP solves the modified problem "from scratch" (without using the BASIS=WARMSTART option) and produces the following iteration log.

Output 10.5.2 Iteration Log

```
NOTE: The problem EX5 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective              Entering     Leaving
      Phase Iteration       Value       Time     Variable     Variable
        D 1           1    0.000000E+00     0
        D 2           2    5.500000E+00     0         mi          fat (S)
        D 2           3    8.650000E+00     0         ch      protein (S)
        D 2           4    8.925676E+00     0         po        carbs (S)
        D 2           5    9.174413E+00     0
        D 2           6    9.174413E+00     0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.05 seconds.
```

It is clear that using the BASIS=WARMSTART option saves computation time. For larger or more complex examples, the benefits of using this option are more pronounced.

Example 10.6: Reoptimizing after Adding a New Constraint

Assume that after solving the diet problem in Example 10.3 you need to add a new constraint on sodium intake of no more than 550 mg/day for adults. The updated nutrition data are given in Table 10.14.

Table 10.14 Updated Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories, Cal	90	120	106	97	130	180
sodium, mg	148	122	337	186	56	132

The input data set ex3 is updated (and the data set is saved as ex6) as follows:

```
/* added a new constraint to the diet problem */
data ex6;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME            .        EX6        .     .         .
ROWS            .        .          .     .         .
 N          diet         .          .     .         .
 G          calories     .          .     .         .
 L          protein      .          .     .         .
 G          fat          .          .     .         .
```

```
 G          carbs      .       .       .       .
 L          sodium     .       .       .       .
COLUMNS      .          .       .       .       .
 .          br         diet    2       calories  90
 .          br         protein 4       fat       1
 .          br         carbs   15      sodium    148
 .          mi         diet    3.5     calories  120
 .          mi         protein 8       fat       5
 .          mi         carbs   11.7    sodium    122
 .          ch         diet    8       calories  106
 .          ch         protein 7       fat       9
 .          ch         carbs   .4      sodium    337
 .          po         diet    1.5     calories  97
 .          po         protein 1.3     fat       .1
 .          po         carbs   22.6    sodium    186
 .          fi         diet    11      calories  130
 .          fi         protein 8       fat       7
 .          fi         carbs   0       sodium    56
 .          yo         diet    1       calories  180
 .          yo         protein 9.2     fat       1
 .          yo         carbs   17      sodium    132
RHS          .          .       .       .       .
 .          .          calories 300    protein   10
 .          .          fat     8       carbs     10
 .          .          sodium  550     .         .
BOUNDS       .          .       .       .       .
UP           .          mi      1       .         .
LO           .          fi      .5      .         .
ENDATA       .          .       .       .       .
 ;
```

For the modified problem you can warm start the primal and dual simplex solvers to get a solution faster. The dual simplex solver is preferred because a dual feasible solution can be readily constructed from the optimal solution to the diet optimization problem.

Since there is a new constraint in the modified problem, you can use the following SAS code to create a new DUALIN= data set ex6din with this information:

```
data ex6newcon;
   _ROW_='sodium  '; _STATUS_='A';
   output;
run;

/* create a new DUALIN= data set to include the new constraint */
data ex6din;
   set ex3dout ex6newcon;
run;
```

Note that this step is optional. In this example, you can still use the data set ex3dout as the DUALIN= data set to solve the modified LP problem by using the BASIS=WARMSTART option. PROC OPTLP validates the PRIMALIN= and DUALIN= data sets against the input model. Any new variable (or constraint) in the model is added to the PRIMALIN= (or DUALIN=) data set, and its status is assigned to be 'A'. The primal and dual simplex solvers decide its corresponding status internally. Any variable in the PRIMALIN= and DUALIN= data sets but not in the input model is removed.

The _ROW_ and _STATUS_ columns of the DUALIN= data set ex6din are shown in Output 10.6.1.

Output 10.6.1 DUALIN= Data Set with a Newly Added Constraint

Obs	_ROW_	_STATUS_
1	calories	U
2	protein	L
3	fat	U
4	carbs	B
5	sodium	A

The dual simplex solver is called to solve the modified diet optimization problem more quickly with the following SAS code:

```
proc optlp data=ex6
   objsense=min
   presolver=none
   algorithm=ds
   primalout=ex6pout
   dualout=ex6dout
   scale=none
   logfreq=1
   basis=warmstart
   primalin=ex3pout
   dualin=ex6din;
run;
```

The optimal primal and dual solutions of the modified problem are displayed in Output 10.6.2.

Output 10.6.2 Primal and Dual Solution Output

```
                           Primal Solution

            Objective
            Function     RHS      Variable     Variable     Objective
    Obs        ID         ID        Name         Type      Coefficient

     1        diet                   br           N            2.0
     2        diet                   mi           D            3.5
     3        diet                   ch           N            8.0
     4        diet                   po           N            1.5
     5        diet                   fi           O           11.0
     6        diet                   yo           N            1.0


            Lower              Upper      Variable     Variable     Reduced
    Obs     Bound              Bound       Value        Status       Cost

     1       0.0        1.7977E308        0.00000         L         1.19066
     2       0.0                 1        0.05360         B         0.00000
     3       0.0        1.7977E308        0.44950         B         0.00000
     4       0.0        1.7977E308        1.86517         B         0.00000
     5       0.5        1.7977E308        0.50000         L         5.15641
     6       0.0        1.7977E308        0.00000         L         1.10849


                            Dual Solution

          Objective                                                      Constraint
          Function     RHS     Constraint    Constraint    Constraint      Lower
    Obs      ID         ID        Name          Type          RHS          Bound

     1      diet               calories          G            300            .
     2      diet               protein           L             10            .
     3      diet               fat               G              8            .
     4      diet               carbs             G             10            .
     5      diet               sodium            L            550            .

          Constraint       Dual
            Upper        Variable     Constraint    Constraint
    Obs     Bound         Value        Status        Activity

     1         .          0.02179        U           300.000
     2         .         -0.55360        L            10.000
     3         .          1.06286        U             8.000
     4         .          0.00000        B            42.960
     5         .          0.00000        B           532.941
```

The iteration log shown in Output 10.6.3 indicates that it takes the dual simplex solver no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 10.6.3 Iteration Log

```
NOTE: The problem EX6 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                         Objective              Entering     Leaving
      Phase Iteration      Value        Time    Variable     Variable
       D 2          1    1.208134E+01      0
NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.05 seconds.
NOTE: The data set WORK.EX6POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX6DOUT has 5 observations and 10 variables.
```

Both this example and Example 10.4 illustrate the situation in which the optimal solution does not change after some perturbation of the parameters of the LP problem. The simplex solver starts from an optimal solution and quickly verifies the optimality. Usually the optimal solution of the slightly perturbed problem can be obtained after performing relatively small number of iterations if starting with the optimal solution of the original problem. In such cases you can expect a dramatic reduction of computation time, for instance, if you want to solve a large LP problem and a slightly perturbed version of this problem by using the BASIS=WARMSTART option rather than solving both problems from scratch.

Example 10.7: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

$$
\begin{array}{lrcrcrcrll}
\min & x_1 & + & x_2 & + & x_3 & & & & \text{(cost)} \\
\text{subject to} & x_1 & + & x_2 & & & & \geq & 10 & \text{(con1)} \\
& x_1 & & & + & x_3 & & \leq & 4 & \text{(con2)} \\
4 \leq & & & x_2 & + & x_3 & & \leq & 5 & \text{(con3)} \\
& x_1, & & x_2 & & & & \geq & 0 & \\
& 0 & \leq & & & x_3 & & \leq & 3 &
\end{array}
$$

The corresponding MPS-format SAS data set is as follows:

```
/* infeasible */
data exiis;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
NAME       .        .        .        .        .
ROWS       .        .        .        .        .
 N        cost      .        .        .        .
 G        con1      .        .        .        .
 L        con2      .        .        .        .
 G        con3      .        .        .        .
```

```
COLUMNS    .           .           .           .           .
    .      x1          cost        1           con1        1
    .      x1          con2        1           .           .
    .      x2          cost        1           con1        1
    .      x2          con3        1           .           .
    .      x3          cost        1           con2        1
    .      x3          con3        1           .           .
RHS        .           .           .           .           .
    .      rhs         con1        10          con2        4
    .      rhs         con3        4           .           .
RANGES     .           .           .           .           .
    .      r1          con3        1           .           .
BOUNDS     .           .           .           .           .
UP         b1          x3          3           .           .
ENDATA     .           .           .           .           .
;
```

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem.

$$
\begin{array}{rrrrrll}
x_1 & + & x_2 & & & \geq & 10 & (\text{con1}) \\
x_1 & & & + & x_3 & \leq & 4 & (\text{con2}) \\
& & x_2 & + & x_3 & \leq & 5 & (\text{con3}) \\
& & & & x_3 & \geq & 0 &
\end{array}
$$

You can use the IIS=ON option to detect this IIS by using the following statements:

```
proc optlp data=exiis
   iis=on
   primalout=iis_vars
   dualout=iis_cons
   logfreq=1;
run;
```

The OPTLP procedure outputs the detected IIS to the data sets specified by the PRIMALOUT= and DU-ALOUT= options, then stops. The notes shown in Output 10.7.1 are printed to the log.

Output 10.7.1 The IIS= Option: Log

```
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The IIS option is enabled.
                         Objective                Entering      Leaving
      Phase Iteration      Value          Time     Variable     Variable
       P 1          1   1.400000E+01        0         x2         con3 (S)
       P 1          2   5.000000E+00        0         x1         con2 (S)
       P 1          3   1.000000E+00        0
       P 1          4   1.000000E+00        0
NOTE: The IIS option found the problem to be infeasible.
NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.
      Processed      Removed      Time
             0            0         0
             1            0         0
             2            0         0
             3            0         0
NOTE: Processing variable bounds.
      Processed      Removed      Time
             0            0         0
             1            0         0
             2            0         0
             3            0         0
NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS option found the problem to be infeasible.
NOTE: The IIS option found an irreducible infeasible set with 1 variables and 3
      constraints.
NOTE: The IIS solve time is 0.03 seconds.
NOTE: The data set WORK.IIS_VARS has 3 observations and 10 variables.
NOTE: The data set WORK.IIS_CONS has 3 observations and 10 variables.
```

The data sets iis_cons and iis_vars are shown in Output 10.7.2.

Output 10.7.2 Identify Rows and Columns in the IIS

Constraints in the IIS

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound
1	cost	rhs	con1	G	10	.
2	cost	rhs	con2	L	4	.
3	cost	rhs	con3	R	.	4

Obs	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	.	.	I_L	.
2	.	.	I_U	.
3	5	.	I_U	.

Variables in the IIS

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient
1	cost	rhs	x1	N	1
2	cost	rhs	x2	N	1
3	cost	rhs	x3	D	1

Obs	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	0	1.7977E308	.		.
2	0	1.7977E308	.		.
3	0	3	.	I_L	.

The constraint $x_2 + x_3 \le 5$, which is an element of the IIS, is created by the RANGES section. The original constraint is con3, a "$\ge$" constraint with an RHS value of 4. If you choose to remove the constraint $x_2 + x_3 \le 5$, you can accomplish this by removing con3 from the RANGES section in the MPS-format SAS data set exiis. Since con3 is the only observation in the section, the identifier observation can also be removed. The modified LP problem is specified in the following SAS statements:

```
  /* dropping con3, feasible */
data exiisf;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
NAME       .       .       .       .       .
ROWS       .       .       .       .       .
 N       cost      .       .       .       .
 G       con1      .       .       .       .
 L       con2      .       .       .       .
```

```
  G       con3      .        .        .        .
COLUMNS   .        .        .        .        .
  .       x1       cost     1        con1     1
  .       x1       con2     1        .        .
  .       x2       cost     1        con1     1
  .       x2       con3     1        .        .
  .       x3       cost     1        con2     1
  .       x3       con3     1        .        .
RHS       .        .        .        .        .
  .       rhs      con1     10       con2     4
  .       rhs      con3     4        .        .
BOUNDS    .        .        .        .        .
UP        b1       x3       3        .        .
ENDATA    .        .        .        .        .
;
```

Since one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets. You can confirm this by submitting the following SAS statements:

```
proc optlp data=exiisf
   pout=po
   iis=on;
run;
```

The notes shown in Output 10.7.3 are printed to the log.

Output 10.7.3 The IIS= Option: Log

```
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The IIS option is enabled.
                        Objective
      Phase Iteration     Value           Time
       P 1          1    1.400000E+01         0
       P 1          3    0.000000E+00         0
NOTE: The IIS option found the problem to be feasible.
NOTE: The IIS solve time is 0.05 seconds.
NOTE: The data set WORK.EXSS has 8 observations and 3 variables.
NOTE: The data set WORK.PO has 3 observations and 10 variables.
```

The solution summary and the primal solution are displayed in Output 10.7.4.

Output 10.7.4 Infeasibility Removed

```
                       Solution Summary

    Obs    Label1                   cValue1              nValue1

     1     Solver                LP                          .
     2     Algorithm             Primal Simplex              .
     3     Objective Function    cost                        .
     4     Solution Status       Feasible                    .
     5                                                       .
     6     Iterations            3                    3.000000
     7     Presolve Time                    0.00              0
     8     Solution Time                    0.05       0.046800


                       Primal Solution

           Objective
           Function    RHS     Variable    Variable    Objective
    Obs       ID        ID       Name         Type     Coefficient

     1      cost       rhs       x1           N             1
     2      cost       rhs       x2           N             1
     3      cost       rhs       x3           D             1


           Lower         Upper     Variable    Variable    Reduced
    Obs    Bound         Bound       Value      Status       Cost

     1       0       1.7977E308       .                        .
     2       0       1.7977E308       .                        .
     3       0                3       .                        .
```

Example 10.8: Using the Network Simplex Solver

This example demonstrates how to use the network simplex solver to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 10.5, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 10.5 Minimum Cost Network Flow Problem: Data
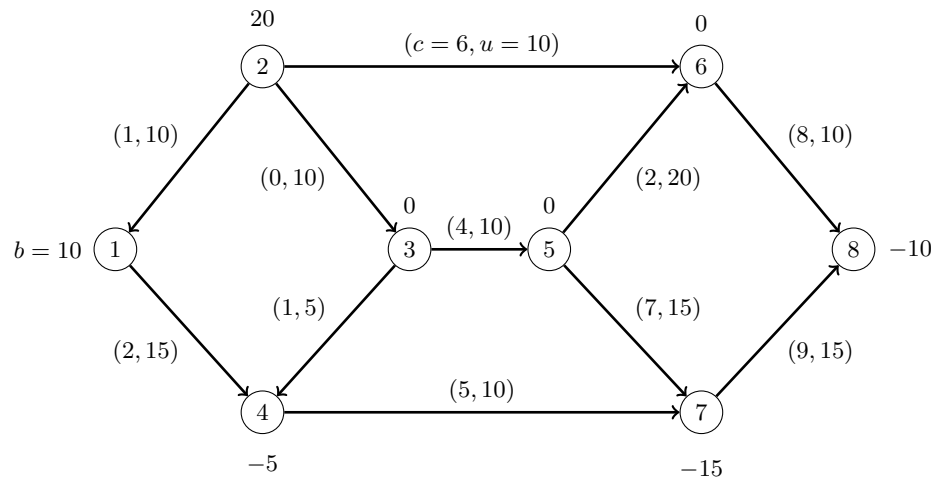
You can use the following SAS statements to create the input data set ex8:

```
data ex8;
   input field1 $8. field2 $13. @25 field3 $13. field4  @53 field5 $13. field6;
   datalines;
NAME      .                     .            .                     .
ROWS      .                     .            .                     .
N         obj                   .            .                     .
E         balance['1']          .            .                     .
E         balance['2']          .            .                     .
E         balance['3']          .            .                     .
E         balance['4']          .            .                     .
E         balance['5']          .            .                     .
E         balance['6']          .            .                     .
E         balance['7']          .            .                     .
E         balance['8']          .            .                     .
COLUMNS   .                     .            .             .
   .      x['1','4']            obj          2             balance['1']          1
   .      x['1','4']            balance['4']  -1            .                     .
   .      x['2','1']            obj          1             balance['1']         -1
   .      x['2','1']            balance['2']  1            .                     .
   .      x['2','3']            balance['2']  1             balance['3']         -1
   .      x['2','6']            obj          6             balance['2']          1
   .      x['2','6']            balance['6']  -1            .                     .
   .      x['3','4']            obj          1             balance['3']          1
   .      x['3','4']            balance['4']  -1            .                     .
   .      x['3','5']            obj          4             balance['3']          1
   .      x['3','5']            balance['5']  -1            .                     .
   .      x['4','7']            obj          5             balance['4']          1
   .      x['4','7']            balance['7']  -1            .                     .
   .      x['5','6']            obj          2             balance['5']          1
   .      x['5','6']            balance['6']  -1            .                     .
   .      x['5','7']            obj          7             balance['5']          1
   .      x['5','7']            balance['7']  -1            .                     .
   .      x['6','8']            obj          8             balance['6']          1
   .      x['6','8']            balance['8']  -1            .                     .
```

```
    .       x['7','8']       obj                     9    balance['7']         1
    .       x['7','8']       balance['8']           -1    .                    .
RHS         .                .                       .    .                    .
    .       .RHS.            balance['1']           10    .                    .
    .       .RHS.            balance['2']           20    .                    .
    .       .RHS.            balance['4']           -5    .                    .
    .       .RHS.            balance['7']          -15    .                    .
    .       .RHS.            balance['8']          -10    .                    .
BOUNDS      .                .                       .    .                    .
UP          .BOUNDS.         x['1','4']             15    .                    .
UP          .BOUNDS.         x['2','1']             10    .                    .
UP          .BOUNDS.         x['2','3']             10    .                    .
UP          .BOUNDS.         x['2','6']             10    .                    .
UP          .BOUNDS.         x['3','4']              5    .                    .
UP          .BOUNDS.         x['3','5']             10    .                    .
UP          .BOUNDS.         x['4','7']             10    .                    .
UP          .BOUNDS.         x['5','6']             20    .                    .
UP          .BOUNDS.         x['5','7']             15    .                    .
UP          .BOUNDS.         x['6','8']             10    .                    .
UP          .BOUNDS.         x['7','8']             15    .                    .
ENDATA      .                .                       .    .                    .
;
```

You can use the following call to PROC OPTLP to find the minimum-cost flow:

```
proc optlp
  presolver  = none
  printlevel = 2
  logfreq    = 1
  data       = ex8
  primalout  = ex8out
  algorithm  = ns;
run;
```

The optimal solution is displayed in Output 10.8.1.

Output 10.8.1 Network Simplex Solver: Primal Solution Output

```
                         Primal Solution

              Objective
              Function     RHS       Variable     Variable    Objective
     Obs         ID        ID          Name         Type      Coefficient

      1         obj       .RHS.     x['1','4']        D            2
      2         obj       .RHS.     x['2','1']        D            1
      3         obj       .RHS.     x['2','3']        D            0
      4         obj       .RHS.     x['2','6']        D            6
      5         obj       .RHS.     x['3','4']        D            1
      6         obj       .RHS.     x['3','5']        D            4
      7         obj       .RHS.     x['4','7']        D            5
      8         obj       .RHS.     x['5','6']        D            2
      9         obj       .RHS.     x['5','7']        D            7
     10         obj       .RHS.     x['6','8']        D            8
     11         obj       .RHS.     x['7','8']        D            9


              Lower     Upper     Variable    Variable    Reduced
     Obs      Bound     Bound      Value       Status      Cost

      1         0        15         10           B           0
      2         0        10          0           L           2
      3         0        10         10           B           0
      4         0        10         10           B           0
      5         0         5          5           B           0
      6         0        10          5           B           0
      7         0        10         10           U          -5
      8         0        20          0           L           0
      9         0        15          5           B           0
     10         0        10         10           B           0
     11         0        15          0           L           6
```

The optimal solution is represented graphically in Figure 10.6.

Figure 10.6 Minimum Cost Network Flow Problem: Optimal Solution
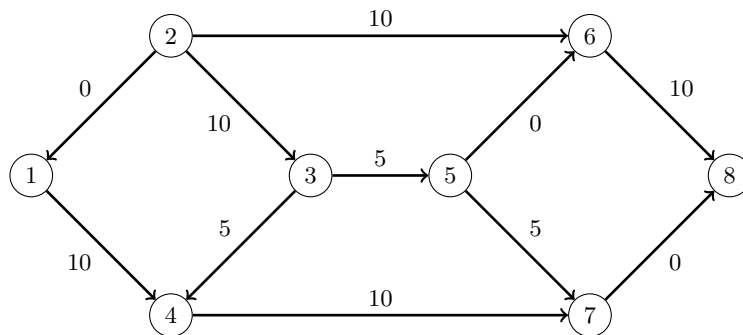
The iteration log is displayed in Output 10.8.2.

Output 10.8.2 Log: Solution Progress

```
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 8 constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 22 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 8 rows (100.00%), 11 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                        Primal         Primal           Dual
        Iteration      Objective    Infeasibility   Infeasibility      Time
                1              0     20.0000000       89.0000000        0.00
                2              0     20.0000000       89.0000000        0.00
                3      5.0000000     15.0000000       84.0000000        0.00
                4      5.0000000     15.0000000       83.0000000        0.00
                5     75.0000000     15.0000000       83.0000000        0.00
                6     75.0000000     15.0000000       79.0000000        0.00
                7    130.0000000     10.0000000       76.0000000        0.00
                8    270.0000000              0                0        0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
NOTE: The data set WORK.EX8OUT has 11 observations and 10 variables.
```

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993), *Network Flows: Theory, Algorithms, and Applications*, Englewood Cliffs, NJ: Prentice-Hall.

Andersen, E. D. and Andersen, K. D. (1995), "Presolving in Linear Programming," *Mathematical Programming*, 71, 221–245.

Chinneck, J. W. (2008), *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, volume 118 of *International Series in Operations Research and Management Sciences*, New York: Springer.

Dantzig, G. B. (1963), *Linear Programming and Extensions*, Princeton, NJ: Princeton University Press.

Forrest, J. J. and Goldfarb, D. (1992), "Steepest-Edge Simplex Algorithms for Linear Programming," *Mathematical Programming*, 5, 1–28.

Gondzio, J. (1997), "Presolve Analysis of Linear Programs prior to Applying an Interior Point Method," *INFORMS Journal on Computing*, 9, 73–91.

Harris, P. M. J. (1973), "Pivot Selection Methods in the Devex LP Code," *Mathematical Programming*, 57, 341–374.

Maros, I. (2003), *Computational Techniques of the Simplex Method*, Boston: Kluwer Academic.

Chapter 11
The OPTMILP Procedure

Contents

Overview: OPTMILP Procedure

The OPTMILP procedure is a solver for general mixed integer linear programs (MILPs).

A standard mixed integer linear program has the formulation

$$
\begin{array}{rll}
\min & \mathbf{c}^T \mathbf{x} & \\
\text{subject to} & \mathbf{A}\mathbf{x} \ \{\geq, =, \leq\} \ \mathbf{b} & \text{(MILP)} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} & \\
& \mathbf{x}_i \in \mathbb{Z} \quad \forall i \in \mathcal{S} &
\end{array}
$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of structural variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of technological coefficients
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables
$\mathcal{S}$			is a nonempty subset of the set $\{1 \ldots, n\}$ of indices

The OPTMILP procedure implements a linear-programming-based branch-and-bound algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The OPTMILP procedure also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The OPTMILP procedure requires a mixed integer linear program to be specified using a SAS data set that adheres to the mathematical programming system (MPS) format, a widely accepted format in the optimization community. Chapter 15 discusses the MPS format in detail. It is also possible to input an incumbent solution in MPS format; see the section "Warm Start" on page 435 for details.

You can use the MPSOUT= option to convert data sets that are formatted for the LP procedure into MPS-format SAS data sets. The option is available in the LP, INTPOINT, and NETFLOW procedures. For details about this option, see Chapter 5, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), Chapter 4, "The INTPOINT Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), and Chapter 6, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

The OPTMILP procedure provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned.

The OPTMILP procedure outputs an optimal solution or the best feasible solution found, if any, in SAS data sets. This enables you to generate solution reports and perform additional analyses by using SAS software.

Getting Started: OPTMILP Procedure

The following example illustrates the use of the OPTMILP procedure to solve mixed integer linear programs. For more examples, see the section "Examples: OPTMILP Procedure" on page 448. Suppose you want to solve the following problem:

$$
\begin{array}{rrrrrrrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & \\
\text{s.t.} & & - & 2x_2 & - & 3x_3 & \geq & -5 \quad \text{(R1)} \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 \quad \text{(R2)} \\
& x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 \quad \text{(R3)} \\
& x_1, & & x_2, & & x_3 & \geq & 0 \\
& x_1, & & x_2, & & x_3 & \in \mathbb{Z} &
\end{array}
$$

The corresponding MPS-format SAS data set follows:

```
data ex_mip;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME            .         EX_MIP        .    .         .
ROWS            .         .             .    .         .
N               COST      .             .    .         .
G               R1        .             .    .         .
L               R2        .             .    .         .
L               R3        .             .    .         .
COLUMNS         .         .             .    .         .
.               MARK00  'MARKER'        .   'INTORG'   .
.               X1        COST          2    R2        1
.               X1        R3            1    .         .
.               X2        COST         -3    R1       -2
.               X2        R2            1    R3        2
.               X3        COST         -4    R1       -3
.               X3        R2            2    R3        3
.               MARK01  'MARKER'        .   'INTEND'   .
RHS             .         .             .    .         .
.               RHS       R1           -5    R2        4
.               RHS       R3            7    .         .
ENDATA          .         .             .    .         .
;
```

You can also create this SAS data set from an MPS-format flat file (ex_mip.mps) by using the following SAS macro:

```
%mps2sasd(mpsfile = "ex_mip.mps", outdata = ex_mip);
```

This problem can be solved by using the following statement to call the OPTMILP procedure:

```
proc optmilp data = ex_mip
   objsense    = min
   primalout   = primal_out
   dualout     = dual_out
   presolver   = automatic
   heuristics  = automatic;
run;
```

The DATA= option names the MPS-format SAS data set that contains the problem data. The OBJSENSE= option specifies whether to maximize or minimize the objective function. The PRIMALOUT= option names the SAS data set to contain the optimal solution or the best feasible solution found by the solver. The DUALOUT= option names the SAS data set to contain the constraint activities. The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal integer solution and its corresponding constraint activities, stored in the data sets primal_out and dual_out, respectively, are displayed in Figure 11.1 and Figure 11.2.

Figure 11.1 Optimal Solution

```
                        The OPTMILP Procedure
                        Primal Integer Solution

        Objective
        Function   RHS  Variable  Variable   Objective   Lower  Upper  Variable
  Obs      ID      ID     Name      Type     Coefficient  Bound  Bound   Value

   1      COST    RHS     X1         B            2         0      1       0
   2      COST    RHS     X2         B           -3         0      1       1
   3      COST    RHS     X3         B           -4         0      1       1
```

Figure 11.2 Constraint Activities

```
                        The OPTMILP Procedure
                        Constraint Information

           Objective
           Function   RHS     Constraint    Constraint    Constraint
  Obs         ID      ID         Name          Type          RHS

   1         COST    RHS          R1            G             -5
   2         COST    RHS          R2            L              4
   3         COST    RHS          R3            L              7

           Constraint   Constraint
             Lower        Upper      Constraint
  Obs        Bound        Bound       Activity

   1           .            .           -5
   2           .            .            3
   3           .            .            5
```

The solution summary stored in the macro variable _OROPTMILP_ can be viewed by issuing the following statement:

```
%put &_OROPTMILP_;
```

This produces the output shown in Figure 11.3.

Figure 11.3 Macro Output

```
STATUS=OK   ALGORITHM=BAC   SOLUTION_STATUS=OPTIMAL   OBJECTIVE=-7
RELATIVE_GAP=0   ABSOLUTE_GAP=0   PRIMAL_INFEASIBILITY=0
BOUND_INFEASIBILITY=0   INTEGER_INFEASIBILITY=0   BEST_BOUND=.   NODES=0
ITERATIONS=0   PRESOLVE_TIME=0.00   SOLUTION_TIME=0.00
```

See the section "Data Input and Output" on page 433 for details about the type and status codes displayed for variables and constraints.

Syntax: OPTMILP Procedure

The following statements are available in the OPTMILP procedure:

PROC OPTMILP < *options* > ;
 DECOMP < *options* > ;
 DECOMP_MASTER < *options* > ;
 DECOMP_MASTER_IP < *options* > ;
 DECOMP_SUBPROB < *options* > ;
 PERFORMANCE < *performance-options* > ;
 TUNER < *tuner-options* > ;

Functional Summary

Table 11.1 summarizes the options available for the OPTMILP procedure, classified by function.

Table 11.1 Options for the OPTMILP Procedure

Description	Option
Data Set Options	
Specifies the input data set	DATA=
Specifies the constraint activities output data set	DUALOUT=
Specifies whether the MILP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution input data set (warm start)	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=

Table 11.1 (continued)

Description	Option
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the node log	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=
Toggles ODS output	PRINTLEVEL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Enables use of variable priorities	PRIORITY=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=

Description	Option
Specifies the zero-half cut level	CUTZEROHALF=

PROC OPTMILP Statement

>**PROC OPTMILP** *< options >* **;**

You can specify the following options in the PROC OPTMILP statement.

Data Set Options

DATA=*SAS-data-set*

> specifies the input data set that corresponds to the MILP model. If this option is not specified, PROC OPTMILP uses the most recently created SAS data set. See Chapter 15, "The MPS-Format SAS Data Set," for more details about the input data set.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

> specifies the output data set to contain the constraint activities.

OBJSENSE=MIN | **MAX**

> specifies whether the MILP model is a minimization or a maximization problem. You can use OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set. This option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTMILP interprets and solves the MILP as a minimization problem.

PRIMALIN=*SAS-data-set*

> enables you to input a warm start solution in a SAS data set. PROC OPTMILP validates both the data set and the solution stored in the data set. If the data set is not valid, then the PRIMALIN= data are ignored. If the solution stored in a valid PRIMALIN= data set is a feasible integer solution, then it provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution stored in a valid PRIMALIN= data set is infeasible, contains missing values, or contains fractional values for integer variables, PROC OPTMILP tries to repair the solution with a number of specialized repair heuristics. See the section "Warm Start" on page 435 for details.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*

> specifies the output data set for the primal solution. This data set contains the primal solution information. See the section "Data Input and Output" on page 433 for details.

Presolve Option

PRESOLVER=*number* | *string*

specifies a presolve *string* or its corresponding value *number*, as listed in Table 11.2.

Table 11.2   Values for PRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default value is AUTOMATIC.

Control Options

ABSOBJGAP=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the objective of the best remaining node becomes smaller than the value of *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=*number*

cuts off any nodes in a minimization (maximization) problem with an objective value above (below) *number*. The value of *number* can be any number; the default value is the positive (negative) number that has the largest absolute value that can be represented in your operating environment.

EMPHASIS=*number* | *string*

specifies a search emphasis *string* or its corresponding value *number* as listed in Table 11.3.

Table 11.3   Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

The default value is BALANCE.

FEASTOL=*number*

specifies the tolerance used to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default value is 1E–6.

If PROC OPTMILP fails to find a feasible solution within this tolerance but does find a solution with a slightly larger violation, then the procedure ends with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 445).

INTTOL=*number*

> specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 0.0 and 1.0; the default value is 1E–5. PROC OPTMILP attempts to find an optimal solution with integer infeasibility less than *number*. If you assign a value smaller than 1E–10 to *number* and the best solution found by PROC OPTMILP has integer infeasibility between *number* and 1E–10, then PROC OPTMILP ends with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 445).

LOGFREQ=*number*

PRINTFREQ=*number*

> specifies how often information is printed in the node log. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=*number | string*

PRINTLEVEL2=*number | string*

> controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each node. Table 11.4 describes the valid values for this option.

Table 11.4   Values for LOGLEVEL= Option

number	string	Description
0	NONE	Turns off all solver-related messages in the SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval dictated by the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval dictated by the LOGFREQ= option

> The default value is MODERATE.

MAXNODES=*number*

> specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXSOLS=*number*

> specifies a stopping criterion. If *number* solutions have been found, then the procedure stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXTIME=*t*

> specifies an upper limit of *t* seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*number*

> specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default is 1E–6.

PRINTLEVEL=0 | 1 | 2

> specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

> For details about the ODS tables created by PROC OPTMILP, see the section "ODS Tables" on page 441.

PROBE=*number* | *string*

> specifies a probing *string* or its corresponding value *number*, as listed in Table 11.5:

Table 11.5 Values for PROBE= Option

number	*string*	**Description**
−1	AUTOMATIC	Uses the probing strategy determined by PROC OPTMILP
0	NONE	Disables probing
1	MODERATE	Uses the probing moderately
2	AGGRESSIVE	Uses the probing aggressively

> The default value is AUTOMATIC. See the section "Presolve and Probing" on page 438 for more information.

RELOBJGAP=*number*

> specifies a stopping criterion based on the best integer objective (BestInteger) and the objective of the best remaining node (BestBound). The relative objective gap is equal to

$$| \text{ BestInteger} - \text{BestBound } | / (1\text{E}{-}10 + | \text{ BestBound } |)$$

> When this value becomes smaller than the specified gap size *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

SCALE=*number* | *string*

> indicates whether to scale the problem matrix. SCALE= can take either of the values AUTOMATIC (−1) and NONE (0). SCALE=AUTOMATIC scales the matrix as determined by PROC OPTMILP; SCALE=NONE disables scaling. The default value is AUTOMATIC.

TARGET=_number_

 specifies a stopping criterion for minimization (maximization) problems. If the best integer objective
 is better than or equal to _number_, the procedure stops. The value of _number_ can be any number; the
 default value is the negative (positive) number that has the largest absolute value representable in your
 operating environment.

TIMETYPE=_number | string_

 specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTMILP_
 macro variable in a PROC OPTMILP call. Table 11.6 describes the valid values of the TIMETYPE=
 option.

Table 11.6 Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

 The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES=
 options in the PERFORMANCE statement. See the section "PERFORMANCE Statement" on page 28
 in Chapter 4, "Shared Concepts and Topics." for more information about the NTHREADS= and
 NODES= options.

 If you specify a value greater than 1 for either the NTHREADS= or the NODES= option, the default
 value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and
 NODES= options, the default value of the TIMETYPE= option is CPU.

Heuristics Option

HEURISTICS=_number | string_

 controls the level of primal heuristics applied by PROC OPTMILP. This level determines how frequently
 primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum
 number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might
 be disabled by the solver at less aggressive levels. The values of _string_ and the corresponding values of
 number are listed in Table 11.7.

Table 11.7 Values for HEURISTICS= Option

number	string	Description
–1	AUTOMATIC	Applies the default level of heuristics, similar to MODERATE
0	NONE	Disables all primal heuristics
1	BASIC	Applies basic primal heuristics at low frequency
2	MODERATE	Applies most primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all primal heuristics at high frequency

Setting HEURISTICS=NONE does not disable the heuristics that repair an infeasible input solution that is
specified in a PRIMALIN= data set.

The default value of the HEURISTICS= option is AUTOMATIC. For details about primal heuristics, see the section "Primal Heuristics" on page 440.

Search Options

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search performed by PROC OPTMILP. Conflict search is used to find clauses resulting from infeasible subproblems that arise in the search tree. The values of *string* and the corresponding values of *number* are listed in Table 11.8.

Table 11.8 Values for CONFLICTSEARCH= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs conflict search based on a strategy determined by PROC OPTMILP
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

The default value is AUTOMATIC.

NODESEL=*number* | *string*

specifies the node selection strategy *string* or its corresponding value *number*, as listed in Table 11.9.

Table 11.9 Values for NODESEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node with the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node with the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

The default value is AUTOMATIC. For details about node selection, see the section "Node Selection" on page 437.

PRIORITY=0 | 1

indicates whether to use specified branching priorities for integer variables. PRIORITY=0 ignores variable priorities; PRIORITY=1 uses priorities when they exist. The default value is 1. See the section "Branching Priorities" on page 438 for details.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations performed for each variable in the candidate list when using the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

STRONGLEN=_number_ | **AUTOMATIC**

    specifies the number of candidates used when performing the strong branching variable selection strategy. The value of _number_ can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

VARSEL=_number_ | _string_

    specifies the rule for selecting the branching variable. The values of _string_ and the corresponding values of _number_ are listed in Table 11.10.

Table 11.10   Values for VARSEL= Option

number	_string_	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable with maximum infeasibility
1	MININFEAS	Chooses the variable with minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses strong branching variable selection strategy

The default value is AUTOMATIC. For details about variable selection, see the section "Variable Selection" on page 437.

Cut Options

Table 11.11 describes the _string_ and _number_ values for the cut options in PROC OPTMILP.

Table 11.11   Values for Individual Cut Options

number	_string_	**Description**
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by PROC OPTMILP
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

You can specify the CUTSTRATEGY= option to set the overall aggressiveness of the cut generation in PROC OPTMILP. Alternatively, you can use the ALLCUTS= option to set all cut types to the same level. You can override the ALLCUTS= value by using the options that correspond to particular cut types. For example, if you want PROC OPTMILP to generate only Gomory cuts, specify ALLCUTS=NONE and CUTGOMORY=AUTOMATIC. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set ALLCUTS=AGGRESSIVE and CUTLAP=NONE.

ALLCUTS=_number_ | _string_

    provides a shorthand way of setting all the cuts-related options in one setting. In other words, ALLCUTS=_number_ is equivalent to setting each of the individual cuts parameters to the same value _number_. Thus, ALLCUTS=–1 has the effect of setting CUTCLIQUE=–1, CUTFLOWCOVER=–1, CUTFLOWPATH=–1, ..., CUTMIR=–1, and CUTZEROHALF=–1. Table 11.11 lists the values that can be assigned to _string_ and _number_. In addition, you can override levels for individual cuts with the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIM-

PLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, and CUTZEROHALF= options. If the ALLCUTS= option is not specified, all the cuts-related options are either set to their individually specified values (if the corresponding option is specified) or to their default values (if that option is not specified).

CUTCLIQUE=*number* | *string*

specifies the level of clique cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTCLIQUE= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWCOVER=*number* | *string*

specifies the level of flow cover cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTFLOWCOVER= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWPATH=*number* | *string*

specifies the level of flow path cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTFLOWPATH= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGOMORY=*number* | *string*

specifies the level of Gomory cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTGOMORY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGUB=*number* | *string*

specifies the level of generalized upper bound (GUB) cover cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTGUB= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTIMPLIED=*number* | *string*

specifies the level of implied bound cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTIMPLIED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTKNAPSACK=*number* | *string*

specifies the level of knapsack cover cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTKNAPSACK= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTLAP=*number* | *string*

specifies the level of lift-and-project (LAP) cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTLAP= option overrides the ALLCUTS= option. The default value is NONE.

CUTMILIFTED=*number* | *string*

specifies the level of mixed lifted 0-1 cuts that are generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *option* and *num*. The CUTMILIFTED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMIR=*number | string*

> specifies the level of mixed integer rounding (MIR) cuts generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*. The CUTMIR= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTSFACTOR=*number*

> specifies a row multiplier factor for cuts. The number of cuts that are added is limited to *number* times the original number of rows. The value of *number* can be any nonnegative number less than or equal to 100; the default value is automatically calculated by PROC OPTMILP.

CUTSTRATEGY=*number | string*

CUTS=*number | string*

> specifies the overall aggressiveness of the cut generation in the solver. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is basic, moderate, or aggressive compared to the default value.

CUTZEROHALF=*number | string*

> specifies the level of zero-half cuts that are generated by PROC OPTMILP. Table 11.11 lists the values that can be assigned to *string* and *number*.The CUTZEROHALF= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTMILP procedure:

> **DECOMP** < *options* > **;**
>
> **DECOMP_MASTER** < *options* > **;**
>
> **DECOMP_MASTER_IP** < *options* > **;**
>
> **DECOMP_SUBPROB** < *options* > **;**

For more information about these statements, see Chapter 13, "The Decomposition Algorithm."

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for single-machine mode and distributed mode, and requests detailed performance results of the OPTMILP procedure.

With the PERFORMANCE statement, you can also control whether the OPTMILP procedure executes in single-machine or distribute mode. Only the decomposition algorithm and the option tuner can be run in those modes.

The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics."

NOTE: Distributed mode requires SAS High-Performance Optimization.

TUNER Statement

TUNER < *performance-options* > **;**

The TUNER statement invokes the OPTMILP option tuner. The option tuner is a tool that enables you to explore alternative (and potentially better) option configurations for your optimization problems. For more information about this feature, see Chapter 14, "The OPTMILP Option Tuner."

Details: OPTMILP Procedure

Data Input and Output

This subsection describes the PRIMALIN= data set required to warm start PROC OPTMILP, in addition to the PRIMALOUT= and DUALOUT= data sets.

Definitions of Variables in the PRIMALIN= Data Set

The PRIMALIN= data set has two required variables defined as follows:

VAR
> specifies the variable (column) names of the problem. The values should match the column names in the DATA= data set for the current problem.

VALUE
> specifies the solution value for each variable in the problem.

NOTE: If PROC OPTMILP produces a feasible solution, the primal output data set from that run can be used as the PRIMALIN= data set for a subsequent run, provided that the variable names are the same. If this input solution is not feasible for the subsequent run, the solver automatically tries to repair it. See the section "Warm Start" on page 435 for more details.

Definitions of Variables in the PRIMALOUT= Data Set

PROC OPTMILP stores the current best integer feasible solution of the problem in the data set specified by the PRIMALOUT= option. The variables in this data set are defined as follows:

_OBJ_ID_
> specifies the identifier of the objective function.

_RHS_ID_
> specifies the identifier of the right-hand side.

VAR
> specifies the variable (column) names.

TYPE
> specifies the variable type. _TYPE_ can take one of the following values:

> C   continuous variable

> I   general integer variable

> B   binary variable (0 or 1)

OBJCOEF
>   specifies the coefficient of the variable in the objective function.

LBOUND
>   specifies the lower bound on the variable.

UBOUND
>   specifies the upper bound on the variable.

VALUE
>   specifies the value of the variable in the current solution.

Definitions of the DUALOUT= Data Set Variables

The DUALOUT= data set contains the constraint activities that correspond to the primal solution in the PRIMALOUT= data set. Information about additional objective rows of the MILP problem is not included. The variables in this data set are defined as follows:

_OBJ_ID_
>   specifies the identifier of the objective function from the input data set.

_RHS_ID_
>   specifies the identifier of the right-hand side from the input data set.

ROW
>   specifies the constraint (row) name.

TYPE
>   specifies the constraint type. _TYPE_ can take one of the following values:

>   L   "less than or equal" constraint

>   E   equality constraint

>   G   "greater than or equal" constraint

>   R   ranged constraint (both "less than or equal" and "greater than or equal")

RHS
>   specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
>   specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
>   specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

ACTIVITY
>   specifies the activity of a constraint for a given primal solution. In other words, the value of _ACTIVITY_ for the ith constraint is equal to $a_i^T x$, where a_i refers to the ith row of the constraint matrix and x denotes the vector of the current primal solution.

Warm Start

PROC OPTMILP enables you to input a warm start solution by using the PRIMALIN= option. PROC OPTMILP checks that the decision variables named in _VAR_ are the same as those in the MPS-format SAS data set. If they are not the same, PROC OPTMILP issues a warning and ignores the input solution. PROC OPTMILP also checks whether the solution is infeasible, contains missing values, or contains fractional values for integer variables. If this is the case, PROC OPTMILP attempts to repair the solution with a number of specialized repair heuristics. The success of the attempt largely depends both on the specific model and on the proximity between the input solution and an integer feasible solution. An infeasible input solution can be considered a hint for PROC OPTMILP that might or might not help to solve the problem.

An integer feasible or repaired input solution provides an incumbent solution in addition to an upper (min) or lower (max) bound for the branch-and-bound algorithm. PROC OPTMILP uses the input solution to reduce the search space and to guide the search process. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how PROC OPTMILP enhances the basic algorithm by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP^0) associated with the root is identical to the original problem, which is called (MILP), given in the section "Overview: OPTMILP Procedure" on page 417.

The linear programming relaxation (LP^0) of (MILP^0) can be written as

$$
\begin{aligned}
\min \quad & \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} \ \{\geq, =, \leq\} \ \mathbf{b} \\
& \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{aligned}
$$

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider $\bar{x}^0$, the optimal solution to (LP^0), which is usually obtained by using the dual simplex algorithm. If $\bar{x}_i^0$ is an integer for all $i \in \mathcal{S}$, then $\bar{x}^0$ is an optimal solution to (MILP). Suppose that for some $i \in \mathcal{S}$, $\bar{x}_i^0$ is nonintegral. In that case the algorithm defines two new subproblems (MILP^1) and (MILP^2), descendants of the parent subproblem (MILP^0). The subproblem (MILP^1) is identical to (MILP^0) except for the additional constraint

$$x_i \leq \lfloor \bar{x}_i^0 \rfloor$$

and the subproblem (MILP^2) is identical to (MILP^0) except for the additional constraint

$$x_i \geq \lceil \bar{x}_i^0 \rceil$$

The notation $\lfloor y \rfloor$ represents the largest integer that is less than or equal to y, and the notation $\lceil y \rceil$ represents the smallest integer that is greater than or equal to y. The two preceding constraints can be handled by modifying the bounds of the variable x_i rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have $\bar{x}^0$ as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable x_i is called the *branching variable*.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is *integer feasible* (that is, x_i is an integer for all $i \in \mathcal{S}$), then its objective value provides an *upper bound* for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section "Controlling the Branch-and-Bound Algorithm" on page 436 for details.

Upper bounds from integer feasible solutions can be used to *fathom* or *cut off* active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if z is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to z can be discarded.

It is important to realize that mixed integer linear programs are non-deterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case $2^{10} = 1,024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case $2^{20} = 1,048,576$ nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has 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.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the *branch-and-cut algorithm*. Preprocessing can reduce problem size and improve problem solvability. PROC OPTMILP employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). PROC OPTMILP implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let (LP^k) be the linear programming relaxation of subproblem (MILP^k). Also, let

$$f_i(k) = \bar{x}_i^k - \lfloor \bar{x}_i^k \rfloor$$

where $\bar{x}^k$ is the optimal solution to the relaxation problem (LP^k) solved at node k.

Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

AUTOMATIC   allows PROC OPTMILP to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND   chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE   chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH   chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\bar{x}^k$, a relaxed optimal solution of (MILP^k), is used to define two active subproblems. In the following list, "INTTOL" refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section "Control Options" on page 424.

AUTOMATIC   enables PROC OPTMILP to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS   chooses as the branching variable the variable x_i such that i maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

MININFEAS   chooses as the branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

PSEUDO   chooses as the branching variable the variable x_i such that i maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG   chooses as the branching variable the variable x_i such that i maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMILP by attaching branching priorities to the integer variables in your model.

You can set branching priorities for use by PROC OPTMILP in two ways. You can specify the branching priorities directly in the input MPS-format data set; see the section "BRANCH Section (Optional)" on page 607 for details. If you are constructing a model in PROC OPTMODEL, you can set branching priorities for integer variables by using the .priority suffix. More information about this suffix is available in the section "Integer Variable Suffixes" on page 134 in Chapter 5. For an example in which branching priorities are used, see Example 7.3.

Presolve and Probing

PROC OPTMILP includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a *polyhedron*. Every polyhedron in n-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \le b, l \le x \le u\}$. After you add the restriction that

some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \; \forall i \in \mathcal{S}\}$, no longer forms a polyhedron.

The *convex hull* of a set X is the minimal convex set that contains X. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(\mathcal{F})$, of $\mathcal{F}$. If you can find $\text{conv}(\mathcal{F})$ and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section "Branch-and-Bound Algorithm" on page 435, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, $\text{conv}(\mathcal{F}) \subseteq \mathcal{Q}$.

Cutting plane techniques are used to tighten the linear relaxation to better approximate $\text{conv}(\mathcal{F})$. Assume you are given a solution $\bar{x}$ to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality $(\pi x \leq \pi^0)$, is some half-space with the following characteristics:

- The half-space contains $\text{conv}(\mathcal{F})$; that is, every integer feasible solution is feasible for the cut $(\pi x \leq \pi^0, \forall x \in \mathcal{F})$.

- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut $(\pi \bar{x} > \pi^0)$.

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are *generic cuts* and *structured cuts*. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of PROC OPTMILP. Table 11.12 lists the various types of cutting planes that are built into PROC OPTMILP. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

Table 11.12 Cutting Planes in PROC OPTMILP

Generic Cutting Planes	Structured Cutting Planes
Gomory mixed integer	Cliques
Lift-and-project	Flow cover
Mixed integer rounding	Flow path
Mixed lifted 0-1	Generalized upper bound cover
Zero-half	Implied bound
	Knapsack cover

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, and CUTMIR= options. The valid levels for these options are given in Table 11.11.

The cut level determines the internal strategy used by PROC OPTMILP for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in PROC OPTMILP, can take a great deal of CPU time. Typically the additional tightening of the relaxation helps to speed up the overall process as it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of PROC OPTMILP have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

Primal Heuristics

Primal heuristics, an important component of PROC OPTMILP, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)

- locating a reasonably good feasible solution when that is sufficient (sometimes a good feasible solution is the best the solver can produce within certain time or resource limits)

- providing upper bounds for some bound-tightening techniques

The OPTMILP procedure implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations allowed in iterative heuristics. The valid values for this option are listed in Table 11.7.

Node Log

The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node   indicates the sequence number of the current node in the search tree.

Active   indicates the current number of active nodes in the branch-and-bound tree.

Sols   indicates the number of feasible solutions found so far.

BestInteger   indicates the best upper bound (assuming minimization) found so far.

BestBound   indicates the best lower bound (assuming minimization) found so far.

Gap   indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.

Time   indicates the elapsed real time.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the node log. By default a new entry is included in the log at the first node, at the last node, and at 100-node intervals. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 11.4 shows a sample node log.

Figure 11.4 Sample Node Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 10 variables, 2 constraints, and 20 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger      BestBound      Gap    Time
            0        1       3     85.0000000    178.0000000   52.25%      0
            0        1       3     85.0000000     88.0955497    3.51%      0
            0        1       3     85.0000000     88.0626822    3.48%      0
            0        1       3     85.0000000     87.9666563    3.37%      0
            0        1       3     85.0000000     87.9661593    3.37%      0
            0        1       3     85.0000000     87.8181818    3.21%      0
NOTE: The MILP solver added 2 cuts with 13 cut coefficients at the root.
            3        0       5     87.0000000     87.0000000    0.00%      0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

ODS Tables

PROC OPTMILP creates three Output Delivery System (ODS) tables by default. The first table, Problem-Summary, is a summary of the input MILP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. See the section "Problem Statistics" on page 445 for more information.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 11.13 lists all the ODS tables that can be produced by the OPTMILP procedure, along with the statement and option specifications required to produce each table.

Table 11.13 ODS Tables Produced by PROC OPTMILP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input MILP problem	PROC OPTMILP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTMILP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTMILP	PRINTLEVEL=2
PerformanceInfo	List of performance options and their values	PROC OPTMILP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical ProblemSummary table is shown in Figure 11.5.

Figure 11.5 Example PROC OPTMILP Output: Problem Summary

```
                    The OPTMILP Procedure

                      Problem Summary

       Problem Name                        EX_MIP
       Objective Sense                Minimization
       Objective Function                    COST
       RHS                                    RHS

       Number of Variables                      3
       Bounded Above                            0
       Bounded Below                            0
       Bounded Above and Below                  3
       Free                                     0
       Fixed                                    0
       Binary                                   3
       Integer                                  0

       Number of Constraints                    3
       LE  (<=)                                 2
       EQ  (=)                                  0
       GE  (>=)                                 1
       Range                                    0

       Constraint Coefficients                  8
```

A typical SolutionSummary table is shown in Figure 11.6.

Figure 11.6 Example PROC OPTMILP Output: Solution Summary

```
                    The OPTMILP Procedure

                      Solution Summary

          Solver                          MILP
          Algorithm              Branch and Cut
          Objective Function              COST
          Solution Status              Optimal
          Objective Value                   -7

          Relative Gap                       0
          Absolute Gap                       0
          Primal Infeasibility               0
          Bound Infeasibility                0
          Integer Infeasibility              0

          Best Bound                         .
          Nodes                              0
          Iterations                         0
          Presolve Time                   0.00
          Solution Time                   0.00
```

You can create output data sets from these tables by using the ODS OUTPUT statement. The output data sets from the preceding example are displayed in Figure 11.7 and Figure 11.8, where you can also find variable names for the tables used in the ODS template of the OPTMILP procedure.

Figure 11.7 ODS Output Data Set: Problem Summary

```
                        Problem Summary

    Obs    Label1                     cValue1              nValue1

     1     Problem Name               EX_MIP                     .
     2     Objective Sense            Minimization               .
     3     Objective Function         COST                       .
     4     RHS                        RHS                        .
     5                                                           .
     6     Number of Variables        3                   3.000000
     7     Bounded Above              0                          0
     8     Bounded Below              0                          0
     9     Bounded Above and Below    3                   3.000000
    10     Free                       0                          0
    11     Fixed                      0                          0
    12     Binary                     3                   3.000000
    13     Integer                    0                          0
    14                                                           .
    15     Number of Constraints      3                   3.000000
    16     LE  (<=)                   2                   2.000000
    17     EQ  (=)                    0                          0
    18     GE  (>=)                   1                   1.000000
    19     Range                      0                          0
    20                                                           .
    21     Constraint Coefficients    8                   8.000000
```

Figure 11.8 ODS Output Data Set: Solution Summary

```
                        Solution Summary

    Obs    Label1                      cValue1              nValue1

     1     Solver                     MILP                       .
     2     Algorithm                  Branch and Cut             .
     3     Objective Function         COST                       .
     4     Solution Status            Optimal                    .
     5     Objective Value            -7                  -7.000000
     6                                                           .
     7     Relative Gap               0                          0
     8     Absolute Gap               0                          0
     9     Primal Infeasibility       0                          0
    10     Bound Infeasibility        0                          0
    11     Integer Infeasibility      0                          0
    12                                                           .
    13     Best Bound                 .                          .
    14     Nodes                      0                          0
    15     Iterations                 0                          0
    16     Presolve Time                       0.02       0.015600
    17     Solution Time                       0.02       0.015600
```

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMILP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 11.9 demonstrates the contents of the ODS table ProblemStatistics.

Figure 11.9 ODS Table ProblemStatistics

```
                          ProblemStatistics

  Obs    Label1                                  cValue1            nValue1

   1     Number of Constraint Matrix Nonzeros    8                  8.000000
   2     Maximum Constraint Matrix Coefficient   3                  3.000000
   3     Minimum Constraint Matrix Coefficient   1                  1.000000
   4     Average Constraint Matrix Coefficient   1.875              1.875000
   5                                                                       .
   6     Number of Objective Nonzeros            3                  3.000000
   7     Maximum Objective Coefficient           4                  4.000000
   8     Minimum Objective Coefficient           2                  2.000000
   9     Average Objective Coefficient           3                  3.000000
  10                                                                       .
  11     Number of RHS Nonzeros                  3                  3.000000
  12     Maximum RHS                             7                  7.000000
  13     Minimum RHS                             4                  4.000000
  14     Average RHS                             5.3333333333       5.333333
  15                                                                       .
  16     Maximum Number of Nonzeros per Column   3                  3.000000
  17     Minimum Number of Nonzeros per Column   2                  2.000000
  18     Average Number of Nonzeros per Column   2                  2.000000
  19                                                                       .
  20     Maximum Number of Nonzeros per Row      3                  3.000000
  21     Minimum Number of Nonzeros per Row      2                  2.000000
  22     Average Number of Nonzeros per Row      2                  2.000000
```

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMILP_

The OPTMILP procedure defines a macro variable named _OROPTMILP_. This variable contains a character string that indicates the status of the OPTMILP procedure upon termination. The various terms of the variable are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data was inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC	The branch-and-cut algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.
TARGET	The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
SOLUTION_LIM	The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified by the MAXTIME= option and found a solution.

TIME_LIM_NOSOL	The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

RELATIVE_GAP

specifies the relative gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the OPTMILP procedure. The relative gap is equal to

$$| \text{BestInteger} - \text{BestBound} | \, / \, (1\text{E--}10 + | \text{BestBound} |)$$

ABSOLUTE_GAP

specifies the absolute gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the OPTMILP procedure. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY

indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY

indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY

indicates the maximum (absolute) violation of the integrality of integer variables returned by the OPTMILP procedure.

BEST_BOUND

specifies the best LP objective value of all unprocessed nodes on the branch-and-bound tree at the end of execution. A missing value indicates that the OPTMILP procedure has processed either all or none of the nodes on the branch-and-bound tree.

NODES

specifies the number of nodes enumerated by the OPTMILP procedure when using the branch-and-bound algorithm.

ITERATIONS

indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME

indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME

indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTMILP Procedure

This section contains examples that illustrate the options and syntax of PROC OPTMILP. Example 11.1 demonstrates a model contained in an MPS-format SAS data set and finds an optimal solution by using PROC OPTMILP. Example 11.2 illustrates the use of standard MPS files in PROC OPTMILP. Example 11.3 demonstrates how to warm start PROC OPTMILP. More detailed examples of mixed integer linear programs, along with example SAS code, are given in Chapter 7.

Example 11.1: Simple Integer Linear Program

This example illustrates a model in an MPS-format SAS data set. This data set is passed to PROC OPTMILP, and a solution is found.

Consider a scenario where you have a container with a set of limiting attributes (volume V and weight W) and a set I of items that you want to pack. Each item type i has a certain value p_i, a volume v_i, and a weight w_i. You must choose at most four items of each type so that the total value is maximized and all the chosen items fit into the container. Let x_i be the number of items of type i to be included in the container. This model can be formulated as the following integer linear program:

$$
\begin{aligned}
\max \quad & \sum_{i \in I} p_i x_i \\
\text{s.t.} \quad & \sum_{i \in I} v_i x_i \leq V && \text{(volume\_con)} \\
& \sum_{i \in I} w_i x_i \leq W && \text{(weight\_con)} \\
& x_i \leq 4 && \forall i \in I \\
& x_i \in \mathbb{Z}^+ && \forall i \in I
\end{aligned}
$$

Constraint (volume_con) enforces the volume capacity limit, while constraint (weight_con) enforces the weight capacity limit. An instance of this problem can be saved in an MPS-format SAS data set by using the following code:

```
data ex1data;
    input field1 $ field2 $ field3 $ field4 field5 $ field6;
    datalines;
```

NAME	.	ex1data		.	.	.
ROWS	.	.		.	.	.
MAX	z	.		.	.	.
L	volume_con	.		.	.	.
L	weight_con	.		.	.	.
COLUMNS	.	.		.	.	.
.	.MRK0	'MARKER'		.	'INTORG'	.
.	x[1]	z	1	volume_con	10
.	x[1]	weight_con	12	.	.
.	x[2]	z	2	volume_con	300
.	x[2]	weight_con	15	.	.
.	x[3]	z	3	volume_con	250
.	x[3]	weight_con	72	.	.
.	x[4]	z	4	volume_con	610
.	x[4]	weight_con	100	.	.
.	x[5]	z	5	volume_con	500
.	x[5]	weight_con	223	.	.
.	x[6]	z	6	volume_con	120
.	x[6]	weight_con	16	.	.
.	x[7]	z	7	volume_con	45
.	x[7]	weight_con	73	.	.
.	x[8]	z	8	volume_con	100
.	x[8]	weight_con	12	.	.
.	x[9]	z	9	volume_con	200
.	x[9]	weight_con	200	.	.
.	x[10]	z	10	volume_con	61
.	x[10]	weight_con	110	.	.
.	.MRK1	'MARKER'	.	'INTEND'	.
RHS	.	.	.	.	.
.	.RHS.	volume_con	1000	.	.
.	.RHS.	weight_con	500	.	.
BOUNDS	.	.	.	.	.
UP	.BOUNDS.	x[1]	4	.	.
UP	.BOUNDS.	x[2]	4	.	.
UP	.BOUNDS.	x[3]	4	.	.
UP	.BOUNDS.	x[4]	4	.	.
UP	.BOUNDS.	x[5]	4	.	.
UP	.BOUNDS.	x[6]	4	.	.
UP	.BOUNDS.	x[7]	4	.	.
UP	.BOUNDS.	x[8]	4	.	.
UP	.BOUNDS.	x[9]	4	.	.
UP	.BOUNDS.	x[10]	4	.	.
ENDATA	.	.	.	.	.

```
;
```

In the COLUMNS section of this data set, the name of the objective is z, and the objective coefficients p_i appear in field4. The coefficients v_i of (volume_con) appear in field6. The coefficients w_i of (weight_con) appear in field4. In the RHS section, the bounds V and W appear in field4.

This problem can be solved by using the following statements to call the OPTMILP procedure:

```
proc optmodel;
    num nItems          = 10;
    num volume_capacity = 1000;
    num weight_capacity = 500;
    set<num> Items = {1..nItems};
    num value{Items}  = [1,2,3,4,5,6,7,8,9,10];
    num volume{Items} = [10, 300, 250, 610, 500, 120, 45, 100, 200, 61 ];
    num weight{Items} = [12,  15,  72, 100, 223,  16, 73,  12, 200, 110];
    var x{Items} integer >= 0 <= 4;
    max z = sum{i in Items} value[i] * x[i];
    con volume_con: sum{i in Items} volume[i] * x[i] <= volume_capacity;
    con weight_con: sum{i in Items} weight[i] * x[i] <= weight_capacity;
    save mps ex1data;
  quit;
run;
proc optmilp data=ex1data primalout=ex1soln;
run;
```

The progress of the solver is shown in Output 11.1.1.

Output 11.1.1 Simple Integer Linear Program PROC OPTMILP Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 10 variables, 2 constraints, and 20 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger      BestBound      Gap    Time
            0        1       3     85.0000000    178.0000000   52.25%       0
            0        1       3     85.0000000     88.0955497    3.51%       0
            0        1       3     85.0000000     88.0626822    3.48%       0
            0        1       3     85.0000000     87.9666563    3.37%       0
            0        1       3     85.0000000     87.9661593    3.37%       0
            0        1       3     85.0000000     87.8181818    3.21%       0
NOTE: The MILP solver added 2 cuts with 13 cut coefficients at the root.
            3        0       5     87.0000000     87.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

The data set ex1soln is shown in Output 11.1.2.

Output 11.1.2 Simple Integer Linear Program Solution

```
                          Example 1 Solution Data

 Objective
 Function    RHS     Variable   Variable   Objective    Lower   Upper   Variable
    ID       ID        Name       Type     Coefficient  Bound   Bound    Value

    z       .RHS.     x[1]         I            1          0       4        0
    z       .RHS.     x[2]         I            2          0       4        0
    z       .RHS.     x[3]         I            3          0       4        0
    z       .RHS.     x[4]         I            4          0       4        0
    z       .RHS.     x[5]         I            5          0       4        0
    z       .RHS.     x[6]         I            6          0       4        3
    z       .RHS.     x[7]         I            7          0       4        1
    z       .RHS.     x[8]         I            8          0       4        4
    z       .RHS.     x[9]         I            9          0       4        0
    z       .RHS.     x[10]        I           10          0       4        3
```

The optimal solution is $x_6 = 3, x_7 = 1, x_8 = 4$, and $x_{10} = 3$, with a total value of 87. From this solution, you can compute the total volume used, which is 988 ($\leq V = 1000$); the total weight used is 499 ($\leq W = 500$). The problem summary and solution summary are shown in Output 11.1.3.

```
proc optmodel;
    num nItems         = 10;
    num volume_capacity = 1000;
    num weight_capacity = 500;
    set<num> Items = {1..nItems};
    num value{Items}  = [1,2,3,4,5,6,7,8,9,10];
    num volume{Items} = [10, 300, 250, 610, 500, 120, 45, 100, 200, 61 ];
    num weight{Items} = [12,  15,  72, 100, 223,  16, 73,  12, 200, 110];
    var x{Items} integer >= 0 <= 4;
    max z = sum{i in Items} value[i] * x[i];
    con volume_con: sum{i in Items} volume[i] * x[i] <= volume_capacity;
    con weight_con: sum{i in Items} weight[i] * x[i] <= weight_capacity;
    save mps ex1data;
  quit;
run;
proc optmilp data=ex1data primalout=ex1soln;
title ' ';
run;
```

Output 11.1.3 Simple Integer Linear Program Summary

```
                    The OPTMILP Procedure

                      Problem Summary

       Problem Name                      ex1data
       Objective Sense              Maximization
       Objective Function                      z
       RHS                                 .RHS.

       Number of Variables                    10
       Bounded Above                           0
       Bounded Below                           0
       Bounded Above and Below                10
       Free                                    0
       Fixed                                   0
       Binary                                  0
       Integer                                10

       Number of Constraints                   2
       LE  (<=)                                 2
       EQ  (=)                                  0
       GE  (>=)                                 0
       Range                                   0

       Constraint Coefficients                20


                  Performance Information

         Execution Mode        Single-Machine
         Number of Threads                   4


                     Solution Summary

       Solver                               MILP
       Algorithm                  Branch and Cut
       Objective Function                      z
       Solution Status                   Optimal
       Objective Value                        87

       Relative Gap                            0
       Absolute Gap                            0
       Primal Infeasibility                    0
       Bound Infeasibility                     0
       Integer Infeasibility                   0

       Best Bound                             87
       Nodes                                   4
       Iterations                             31
       Presolve Time                        0.00
       Solution Time                        0.00
```

Example 11.2: MIPLIB Benchmark Instance

The following example illustrates the conversion of a standard MPS-format file into an MPS-format SAS data set. The problem is re-solved several times, each time by using a different control option. For such a small example, it is necessary to disable cuts and heuristics in order to see the computational savings gained by using other options. For larger or more complex examples, the benefits of using the various control options are more pronounced.

The standard set of MILP benchmark cases is called MIPLIB (Bixby et al. 1998, Achterberg, Koch, and Martin 2003) and can be found at `http://miplib.zib.de/`. The following statement uses the %MPS2SASD macro to convert an example from MIPLIB to a SAS data set:

```
%mps2sasd(mpsfile="bell3a.mps", outdata=mpsdata);
```

The problem can then be solved using PROC OPTMILP on the data set created by the conversion:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=10000;
run;
```

The resulting log is shown in Output 11.2.1.

Output 11.2.1 MIPLIB PROC OPTMILP Log

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 33 variables and 37 constraints.
NOTE: The MILP presolver removed 92 constraint coefficients.
NOTE: The MILP presolver modified 3 constraint coefficients.
NOTE: The presolved problem has 100 variables, 86 constraints, and 255
      constraint coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger      BestBound      Gap     Time
            0        1       0              .         866240        .        0
            0        1       0              .         866240        .        0
          801       49       1         916564         874287     4.84%       0
          881      108       2         916327         874287     4.81%       0
          947      159       3         915158         874287     4.67%       0
          979      165       4         898096         874287     2.72%       0
         1002      123       5         887234         874287     1.48%       0
         1088      122       6         883066         874287     1.00%       0
         1997      394       7         880717         874502     0.71%       0
         6188     1845       8         878430         875484     0.34%       1
        10000     2191      12         878430         875996     0.28%       2
        20000     1198      12         878430         876908     0.17%       4
        23231        2      12         878430         878365     0.01%       5
NOTE: Optimal within relative gap.
NOTE: Objective = 878430.316.
```

Suppose you do not have a bound for the solution. If there is an objective value that, even if it is not optimal, satisfies your requirements, then you can save time by using the TARGET= option. The following PROC OPTMILP call solves the problem with a target value of 880,000:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=5000
             target=880000;
run;
```

The relevant results from this run are displayed in Output 11.2.2. In this case, there is a decrease in CPU time, but the objective value has increased.

Output 11.2.2 MIPLIB PROC OPTMILP Log with TARGET= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 33 variables and 37 constraints.
NOTE: The MILP presolver removed 92 constraint coefficients.
NOTE: The MILP presolver modified 3 constraint coefficients.
NOTE: The presolved problem has 100 variables, 86 constraints, and 255
      constraint coefficients.
NOTE: The MILP solver is called.
          Node   Active    Sols    BestInteger       BestBound      Gap     Time
             0        1       0              .          866240        .        0
             0        1       0              .          866240        .        0
           801       49       1         916564          874287    4.84%        0
           881      108       2         916327          874287    4.81%        0
           947      159       3         915158          874287    4.67%        0
           979      165       4         898096          874287    2.72%        0
          1002      123       5         887234          874287    1.48%        0
          1088      122       6         883066          874287    1.00%        0
          1997      394       7         880717          874502    0.71%        0
          5000     1430       7         880717          875311    0.62%        1
          6188     1938       8         878430          875484    0.34%        1
NOTE: Target reached.
NOTE: Objective of the best integer solution found = 878430.316.
```

When the objective value of a solution is within a certain relative gap of the optimal objective value, the procedure stops. The acceptable relative gap can be changed using the RELOBJGAP= option, as demonstrated in the following example:

```
proc optmilp data=mpsdata allcuts=none heuristics=none relobjgap=0.01;
run;
```

The relevant results from this run are displayed in Output 11.2.3. In this case, since the specified RELOBJ-GAP= value is larger than the default value, the number of nodes and the CPU time have decreased from their values in the original run. Note that these savings are exchanged for an increase in the objective value of the solution.

Output 11.2.3 MIPLIB PROC OPTMILP Log with RELOBJGAP= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 33 variables and 37 constraints.
NOTE: The MILP presolver removed 92 constraint coefficients.
NOTE: The MILP presolver modified 3 constraint coefficients.
NOTE: The presolved problem has 100 variables, 86 constraints, and 255
      constraint coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols   BestInteger      BestBound      Gap     Time
            0        1       0             .         866240        .        0
            0        1       0             .         866240        .        0
          100       85       0             .         873180        .        0
          200      166       0             .         873577        .        0
          300      243       0             .         873730        .        0
          400      317       0             .         873867        .        0
          500      379       0             .         874141        .        0
          600      460       0             .         874247        .        0
          700      544       0             .         874262        .        0
          800       50       0             .         874287        .        0
          801       49       1        916564         874287     4.84%       0
          881      108       2        916327         874287     4.81%       0
          900      125       2        916327         874287     4.81%       0
          947      159       3        915158         874287     4.67%       0
          979      165       4        898096         874287     2.72%       0
         1000      185       4        898096         874287     2.72%       0
         1002      123       5        887234         874287     1.48%       0
         1088      122       6        883066         874287     1.00%       0
         1100      129       6        883066         874287     1.00%       0
         1200      170       6        883066         874287     1.00%       0
         1300      202       6        883066         874287     1.00%       0
         1400      244       6        883066         874287     1.00%       0
         1500      275       6        883066         874287     1.00%       0
         1600      303       6        883066         874287     1.00%       0
         1700      331       6        883066         874314     1.00%       0
         1754      353       6        883066         874325     1.00%       0
NOTE: Optimal within relative gap.
NOTE: Objective = 883066.108.
```

The MAXTIME= option enables you to accept the best solution produced by PROC OPTMILP in a specified amount of time. The following example illustrates the use of the MAXTIME= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxtime=0.1;
run;
```

The relevant results from this run are displayed in Output 11.2.4. Once again, a reduction in solution time is traded for an increase in objective value.

Output 11.2.4 MIPLIB PROC OPTMILP Log with MAXTIME= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 33 variables and 37 constraints.
NOTE: The MILP presolver removed 92 constraint coefficients.
NOTE: The MILP presolver modified 3 constraint coefficients.
NOTE: The presolved problem has 100 variables, 86 constraints, and 255
      constraint coefficients.
NOTE: The MILP solver is called.
        Node   Active    Sols    BestInteger       BestBound      Gap     Time
           0        1       0              .          866240        .        0
           0        1       0              .          866240        .        0
           1        2       0              .          866240        .        0
NOTE: CPU time limit reached.
NOTE: No integer solutions found.
```

The MAXNODES= option enables you to limit the number of nodes generated by PROC OPTMILP. The following example illustrates the use of the MAXNODES= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxnodes=500;
run;
```

The relevant results from this run are displayed in Output 11.2.5. PROC OPTMILP displays the best objective value of all the solutions produced.

Output 11.2.5 MIPLIB PROC OPTMILP Log with MAXNODES= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 33 variables and 37 constraints.
NOTE: The MILP presolver removed 92 constraint coefficients.
NOTE: The MILP presolver modified 3 constraint coefficients.
NOTE: The presolved problem has 100 variables, 86 constraints, and 255
      constraint coefficients.
NOTE: The MILP solver is called.
         Node  Active   Sols    BestInteger      BestBound      Gap     Time
            0       1      0              .         866240        .        0
            0       1      0              .         866240        .        0
          100      85      0              .         873180        .        0
          200     166      0              .         873577        .        0
          300     243      0              .         873730        .        0
          400     317      0              .         873867        .        0
          499     378      0              .         874141        .        0
NOTE: Node limit reached.
NOTE: No integer solutions found.
```

Example 11.3: Facility Location

This advanced example demonstrates how to warm start PROC OPTMILP by using the PRIMALIN= option. The model is constructed in PROC OPTMODEL and saved in an MPS-format SAS data set for use in PROC OPTMILP. This problem can also be solved from within PROC OPTMODEL; see Chapter 7 for details.

Consider the classical facility location problem. Given a set L of customer locations and a set F of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand d_i must be satisfied, and each facility has a demand capacity limit C. The total cost is the sum of the distances c_{ij} between facility j and its assigned customer i, plus a fixed charge f_j for building a facility at site j. Let $y_j = 1$ represent choosing site j to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer i to facility j, and 0 otherwise. This model can be formulated as the following integer linear program:

$$
\begin{aligned}
\min \quad & \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j \\
\text{s.t.} \quad & \sum_{j \in F} x_{ij} && = && 1 && \forall i \in L && \text{(assign\_def)} \\
& x_{ij} && \leq && y_j && \forall i \in L, j \in F && \text{(link)} \\
& \sum_{i \in L} d_i x_{ij} && \leq && C y_j && \forall j \in F && \text{(capacity)} \\
& x_{ij} \in \{0, 1\} && && && \forall i \in L, j \in F \\
& y_j \in \{0, 1\} && && && \forall j \in F
\end{aligned}
$$

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$\min \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij}$$

First, construct a random instance of this problem by using the following DATA steps:

```
%let NumCustomers  = 50;
%let NumSites      = 10;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
      output;
   end;
run;
```

The following PROC OPTMODEL statements generate the model and define both variants of the cost function:

```
proc optmodel;
   set <str> CUSTOMERS;
   set <str> SITES init {};
```

```
    /* x and y coordinates of CUSTOMERS and SITES */
    num x {CUSTOMERS union SITES};
    num y {CUSTOMERS union SITES};
    num demand {CUSTOMERS};
    num fixed_charge {SITES};

    /* distance from customer i to site j */
    num dist {i in CUSTOMERS, j in SITES}
        = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

    read data cdata into CUSTOMERS=[name] x y demand;
    read data sdata into SITES=[name] x y fixed_charge;

    var Assign {CUSTOMERS, SITES} binary;
    var Build {SITES} binary;
    /* each customer assigned to exactly one site */
    con assign_def {i in CUSTOMERS}:
        sum {j in SITES} Assign[i,j] = 1;

    /* if customer i assigned to site j, then facility must be */
    /* built at j                                             */
    con link {i in CUSTOMERS, j in SITES}:
        Assign[i,j] <= Build[j];

    /* each site can handle at most &SiteCapacity demand */
    con capacity {j in SITES}:
        sum {i in CUSTOMERS} demand[i] * Assign[i,j]
        <= &SiteCapacity * Build[j];

    min CostNoFixedCharge
        = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];

    save mps nofcdata;
    min CostFixedCharge
        = CostNoFixedCharge
          + sum {j in SITES} fixed_charge[j] * Build[j];

    save mps fcdata;

quit;
```

First solve the problem for the model with no fixed charge by using the following statements. The first PROC SQL call populates the macro variables varcostNo. This macro variable displays the objective value when the results are plotted. The second PROC SQL call generates a data set that is used to plot the results. The information printed in the log by PROC OPTMILP is displayed in Output 11.3.1.

```
proc optmilp data=nofcdata primalout=nofcout;
run;

proc sql noprint;
    select put(sum(_objcoef_ * _value_),6.1) into :varcostNo
    from nofcout;
quit;
```

```
proc sql;
   create table CostNoFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      nofcout(where=(substr(_var_,1,6)='Assign' and
                     round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 11.3.1 PROC OPTMILP Log for Facility Location with No Fixed Charges

```
NOTE: The problem nofcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000
      constraint coefficients.
NOTE: The MILP solver is called.
         Node  Active   Sols    BestInteger      BestBound      Gap    Time
            0       1      2     972.1737321              0    972.2       0
            0       1      2     972.1737321    961.2403449    1.14%       0
            0       1      2     972.1737321    966.4826332    0.59%       0
            0       1      3     966.4832160    966.4826332    0.00%       0
NOTE: The MILP solver added 2 cuts with 85 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 966.48321599.
NOTE: The data set WORK.NOFCOUT has 510 observations and 8 variables.
```

Next, solve the fixed-charge model by using the following statements. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for PROC OPTMILP. The PRIMALIN= option provides an incumbent solution ("warm start"). The two PROC SQL calls perform the same functions as in the case with no fixed charges. The results from this approach are shown in Output 11.3.2.

```
proc optmilp data=fcdata primalin=nofcout;
run;

proc sql noprint;
   select put(sum(_objcoef_ * _value_), 6.1) into :varcost
   from fcout(where=(substr(_var_,1,6)='Assign'));
   select put(sum(_objcoef_ * _value_), 5.1) into :fixcost
   from fcout(where=(substr(_var_,1,5)='Build'));
   select put(sum(_objcoef_ * _value_), 6.1) into :totalcost
   from fcout;
quit;
```

```
proc sql;
   create table CostFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      fcout(where=(substr(_var_,1,6)='Assign' and
                  round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 11.3.2 PROC OPTMILP Log for Facility Location with Fixed Charges, Using Warm Start

```
NOTE: The problem fcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010
      constraint coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger      BestBound       Gap    Time
            0        1       3  16070.0150023              0     16070       0
            0        1       3  16070.0150023  9946.2514269    61.57%       0
            0        1       3  16070.0150023  9975.8221868    61.09%       0
            0        1       3  16070.0150023  9996.7828163    60.75%       0
            0        1       3  16070.0150023 10005.3277737    60.61%       0
            0        1       3  16070.0150023 10022.6122282    60.34%       0
            0        1       3  16070.0150023 10026.8805471    60.27%       0
            0        1       6  10956.8966327 10031.9134743     9.22%       0
            0        1       7  10952.5054868 10035.6069807     9.14%       0
            0        1       7  10952.5054868 10039.0722957     9.10%       0
            0        1       7  10952.5054868 10040.8138109     9.08%       0
            0        1       7  10952.5054868 10041.6071322     9.07%       0
            0        1       7  10952.5054868 10041.9344067     9.07%       0
            0        1       7  10952.5054868 10041.9344067     9.07%       1
NOTE: The MILP solver added 24 cuts with 533 cut coefficients at the root.
          105       22       8  10949.9022613 10942.9069918     0.06%       3
          141       31       9  10949.9022613 10946.8973957     0.03%       3
          159       26      10  10948.4603465 10947.8766776     0.01%       3
NOTE: Optimal within relative gap.
NOTE: Objective = 10948.460347.
NOTE: The data set WORK.FCOUT has 510 observations and 8 variables.
```

The following two SAS programs produce a plot of the solutions for both variants of the model, using data sets produced by PROC SQL from the PRIMALOUT= data sets produced by PROC OPTMILP.

NOTE: Execution of this code requires SAS/GRAPH software.

```
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";
data csdata;
   set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
%annomac;
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
set CostNoFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the first program appears in Output 11.3.3.

Output 11.3.3 Solution Plot for Facility Location with No Fixed Charges


```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the second program appears in Output 11.3.4.

Output 11.3.4 Solution Plot for Facility Location with Fixed Charges
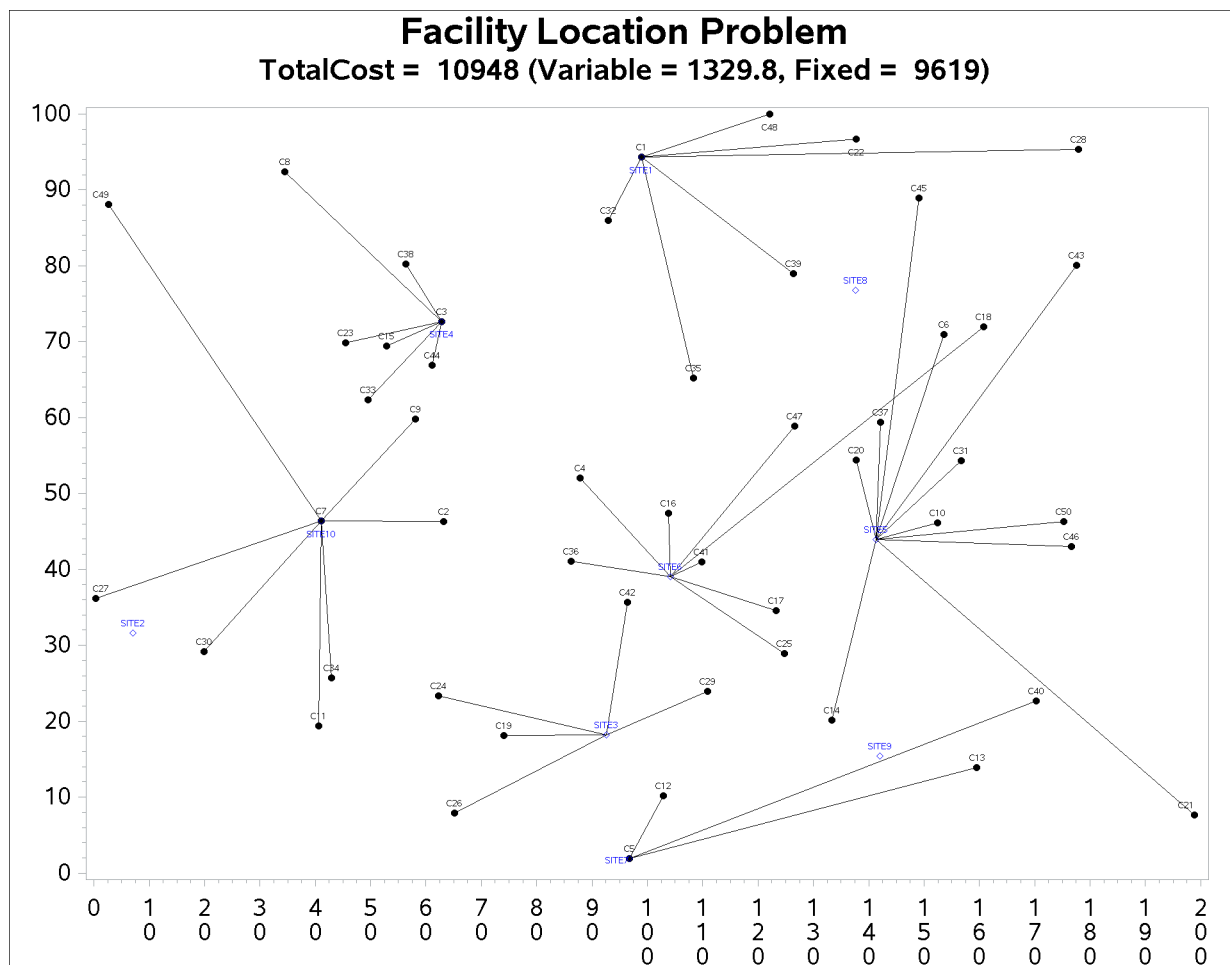
The economic tradeoff for the fixed-charge model forces you to build fewer sites and push more demand to each site.

Example 11.4: Scheduling

This example is intended for users who prefer to use the SAS DATA step, PROC SQL, and similar programming methods to prepare data for input to SAS/OR optimization procedures. SAS/OR users who prefer to use the algebraic modeling capabilities of PROC OPTMODEL to specify optimization models should consult Example 7.1 in Chapter 7, "The Mixed Integer Linear Programming Solver," for a discussion of the same business problem in a PROC OPTMODEL context.

Scheduling is an application area where techniques in model generation can be valuable. Problems that involve 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 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

$$
\begin{aligned}
\text{max} \quad & \sum_{ijk} p_{ijk} x_{ijk} \\
\text{subject to} \quad & \sum_i x_{ijk} = 1 && \text{for all } j \text{ and } k \\
& x_{i4k} + x_{i5k} \leq 1 && \text{for all } i \text{ and } k \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \leq 2 && \text{for all } i \text{ and } k, \text{ and } \ell = 1, \ldots, 6 \\
& \sum_{jk} x_{ijk} \leq h_i && \text{for all } i \\
& x_{ijk} = 0 \text{ or } 1 && \text{for all } i \text{ and } k \text{ such that } p_{ijk} > 0, \\
& && \quad \text{otherwise } x_{ijk} = 0
\end{aligned}
$$

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, which is constructed in the following steps:

1. The objective function is built using the data saved in the RAW data set.

2. The constraints that ensure that no one works during a time slot during which they are unavailable are built.

3. The constraints that require a person to be working in each time slot are built.

4. The constraints that allow each person time for lunch are added.

5. The constraints that restrict people to only two consecutive hours are added.

6. The constraints that limit the time that any one person works in a week are added.

7. The constraints that allow a person to be assigned only to a time slot for which he is available are added.

The statements to build each of these constraints follow 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;
end;


/* build the rest of the model */

/* cannot work during unavailable slots */
do k=1 to 5;
   if workweek{k}=0 then do;
      _row_='off'||put(i,1.)||put(slot,1.)||put(k,1.);
      _type_='eq';
      _col_='_RHS_';
      _coef_=0;
    output;
      _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);
      _coef_=1;
      _type_=' ';
    output;
   end;
end;


if eof then do;
   _coef_=.;
   _col_=' ';

   /* 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;
    end;
run;
```

Next, this SAS data set is converted to an MPS-format SAS data set by establishing the structure of the MPS format and through very minor conversions of the data.

```
    /* the following code transforms the above sparse data set */
    /* into an MPS-format data set                             */

    /* generate the header of the MPS-format data set */
    data mps0;
        format field1 field2 field3 $10.;
        format field4 10.;
        format field5 $10.;
```

```
      format field6 10.;
      field1 = 'NAME';
      field2 = '          ';
      field3 = 'PROBLEM';
      field4 = .;
      field5 = '          ';
      field6 = .;
      output;
      field1 = 'ROWS';
      field3 = '';
      output;
      field1 = 'MAX';
      field2 = 'object';
      field3 = '';
      output;
   run;

   /* generate rows */
   proc sql;
      create table mps1 as
         select _type_ as field1, _row_ as field2 from model
            where _row_ eq 'object' and _type_ ne '' union
         select 'E' as field1, _row_ as field2 from model
            where _type_ eq 'eq' union
         select 'L' as field1, _row_ as field2 from model
            where _type_ eq 'le' union
         select 'G' as field1, _row_ as field2 from model
            where _type_ eq 'ge';
   quit;

   /* indicate start of columns section and declare type of all */
   /* variables as integer                                      */
   data mps2;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'COLUMNS';
      field2 = '          ';
      field3 = '          ';
      field4 = .;
      field5 = '          ';
      field6 = .;
      output;
      field1 = '          ';
      field2 = '.MARK0';
      field3 = "'MARKER'";
      field4 = .;
      field5 = "'INTORG'";
      field6 = .;
      output;
   run;
```

```
   /* generate columns */
   data mps3;
      set model;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      keep field1-field6;
      field1 = '         ';
      field2 = _col_;
      field3 = _row_;
      field4 = _coef_;
      field5 = '         ';
      field6 = .;
      if field2 ne '_RHS_'  then do;
         output;
      end;
   run;

   /* sort columns by variable names */
   proc sort data=mps3;
      by field2;
   run;

   /* indicate the end of the columns section */
   data mps4;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = '          ';
      field2 = '.MARK1';
      field3 = "'MARKER'";
      field4 = .;
      field5 = "'INTEND'";
      field6 = .;
      output;
   run;

   /* indicate the start of the RHS section */
   data mps5;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'RHS';
   run;

   /* generate RHS entries */
   data mps6;
      set model;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
```

```
      format field6 10.;
      keep field1-field6;
      field1 = '        ';
      field2 = _col_;
      field3 = _row_;
      field4 = _coef_;
      field5 = '        ';
      field6 = .;
      if field2 eq '_RHS_'  then do;
         output;
      end;
   run;

   /* denote the end of the MPS-format data set */
   data mps7;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'ENDATA';
   run;

   /* merge all sections of the MPS-format data set */
   data mps;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      set mps0 mps1 mps2 mps3 mps4 mps5 mps6 mps7;
   run;
```

The model is solved using the OPTMILP procedure. The option PRIMALOUT=SOLUTION causes PROC OPTMILP to save the primal solution in the data set named SOLUTION.

```
   /* solve the binary program */
   proc optmilp data=mps
      printlevel=0 loglevel=0
      primalout=solution maxtime=1000;
   run;
```

The following DATA step takes the solution data set SOLUTION and generates a report data set named REPORT. It restores the original interpretation (person, shift, day) of the variable names xijk so that a more meaningful report can be written. Then PROC TABULATE is used to display a schedule that shows 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 11.4.1 from PROC TABULATE summarizes the schedule. Notice that the constraint that requires a person to be assigned to each possible time slot on each day is satisfied.

Output 11.4.1 A Scheduling Problem

```
                        Reported Solution

      -------------------------------------------------------------
      |                | mon   | tue   | wed   | thu   | fri   |
      |----------------+-------+-------+-------+-------+-------|
      |slot   |name    |       |       |       |       |       |
      |-------+--------|       |       |       |       |       |
      |1      |bill    |       | xxx   | xxx   |       | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |marc    |       |       |       | xxx   |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |mike    | xxx   |       |       |       |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |2      |bill    |       |       |       | xxx   | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |bob     | xxx   |       |       |       |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |marc    |       | xxx   | xxx   |       |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |3      |bill    |       |       | xxx   | xxx   |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |bob     | xxx   | xxx   |       |       |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |marc    |       |       |       |       | xxx   |
      |-------+--------+-------+-------+-------+-------+-------|
      |4      |bob     |       |       | xxx   | xxx   | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |mike    | xxx   | xxx   |       |       |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |5      |bill    |       |       |       |       | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |bob     |       | xxx   |       |       |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |marc    | xxx   |       | xxx   | xxx   |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |6      |bill    | xxx   |       |       |       |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |bob     |       |       | xxx   |       | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |mike    |       | xxx   |       | xxx   |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |7      |bob     | xxx   | xxx   |       | xxx   | xxx   |
      |       |--------+-------+-------+-------+-------+-------|
      |       |mike    |       |       | xxx   |       |       |
      |-------+--------+-------+-------+-------+-------+-------|
      |8      |bob     | xxx   |       | xxx   |       |       |
      |       |--------+-------+-------+-------+-------+-------|
      |       |mike    |       | xxx   |       | xxx   | xxx   |
      -------------------------------------------------------------
```

Recall that PROC OPTMILP puts a character string in the macro variable _OROPTMILP_ 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 following command:

```
%put &_OROPTMILP_;
```

This command produces the output shown in Output 11.4.2.

Output 11.4.2 _OROPTMILP_ Macro Variable

```
STATUS=OK   ALGORITHM=BAC   SOLUTION_STATUS=OPTIMAL   OBJECTIVE=211000
RELATIVE_GAP=0   ABSOLUTE_GAP=0   PRIMAL_INFEASIBILITY=0
BOUND_INFEASIBILITY=0   INTEGER_INFEASIBILITY=0   BEST_BOUND=211000   NODES=1
ITERATIONS=71   PRESOLVE_TIME=0.02   SOLUTION_TIME=0.06
```

From this output you learn, for example, that at termination the solution is integer-optimal and has an objective value of 211,000.

References

Achterberg, T., Koch, T., and Martin, A. (2003), "MIPLIB 2003," http://miplib.zib.de/.

Achterberg, T., Koch, T., and Martin, A. (2005), "Branching Rules Revisited," *Operations Research Letters*, 33, 42–54.

Andersen, E. D. and Andersen, K. D. (1995), "Presolving in Linear Programming," *Mathematical Programming*, 71, 221–245.

Atamturk, A. (2004), "Sequence Independent Lifting for Mixed-Integer Programming," *Operations Research*, 52, 487–490.

Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. (1998), "An Updated Mixed Integer Programming Library: MIPLIB 3.0," *Optima*, 58, 12–15.

Dantzig, G. B., Fulkerson, R., and Johnson, S. M. (1954), "Solution of a Large-Scale Traveling Salesman Problem," *Operations Research*, 2, 393–410.

Gondzio, J. (1997), "Presolve Analysis of Linear Programs prior to Applying an Interior Point Method," *INFORMS Journal on Computing*, 9, 73–91.

Land, A. H. and Doig, A. G. (1960), "An Automatic Method for Solving Discrete Programming Problems," *Econometrica*, 28, 497–520.

Linderoth, J. T. and Savelsbergh, M. W. P. (1998), "A Computational Study of Search Strategies for Mixed Integer Programming," *INFORMS Journal on Computing*, 11, 173–187.

Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. (1999), "Cutting Planes in Integer and Mixed Integer Programming," DP 9953, CORE, Université Catholique de Louvain, 1999.

Savelsbergh, M. W. P. (1994), "Preprocessing and Probing Techniques for Mixed Integer Programming Problems," *ORSA Journal on Computing*, 6, 445–454.

Chapter 12
The OPTQP Procedure

Contents

Overview: OPTQP Procedure

The OPTQP procedure solves quadratic programs—problems with quadratic objective function and a collection of linear constraints, including lower or upper bounds (or both) on the decision variables.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

$$\begin{aligned} \min \quad & \tfrac{1}{2}\,\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \;\{\geq, =, \leq\}\; \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

$\mathbf{Q}$	$\in$	$\mathbb{R}^{n \times n}$	is the quadratic (also known as Hessian) matrix
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the constraints matrix
$\mathbf{x}$	$\in$	$\mathbb{R}^{n}$	is the vector of decision variables
$\mathbf{c}$	$\in$	$\mathbb{R}^{n}$	is the vector of linear objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^{m}$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^{n}$	is the vector of lower bounds on the decision variables
$\mathbf{u}$	$\in$	$\mathbb{R}^{n}$	is the vector of upper bounds on the decision variables

Number of variables (columns)

The quadratic matrix $\mathbf{Q}$ is assumed to be symmetric; that is,

$$q_{ij} = q_{ji}, \quad \forall i, j = 1, \dots, n$$

Indeed, it is easy to show that even if $\mathbf{Q} \neq \mathbf{Q}^{\mathrm{T}}$, the simple modification

$$\tilde{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} + \mathbf{Q}^{\mathrm{T}})$$

produces an equivalent formulation $\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \mathbf{x}^{\mathrm{T}}\tilde{\mathbf{Q}}\mathbf{x}$; hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, $\mathbf{Q}$ is also required to be positive semidefinite,

$$\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^{n}$$

for minimization type of models; it is required to be negative semidefinite for the maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

$$\mathbf{Q} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

or as a consequence of physical laws, and so on. See Figure 12.1 for examples of convex, concave, and nonconvex objective functions.

Figure 12.1 Examples of Convex, Concave, and Nonconvex Objective Functions
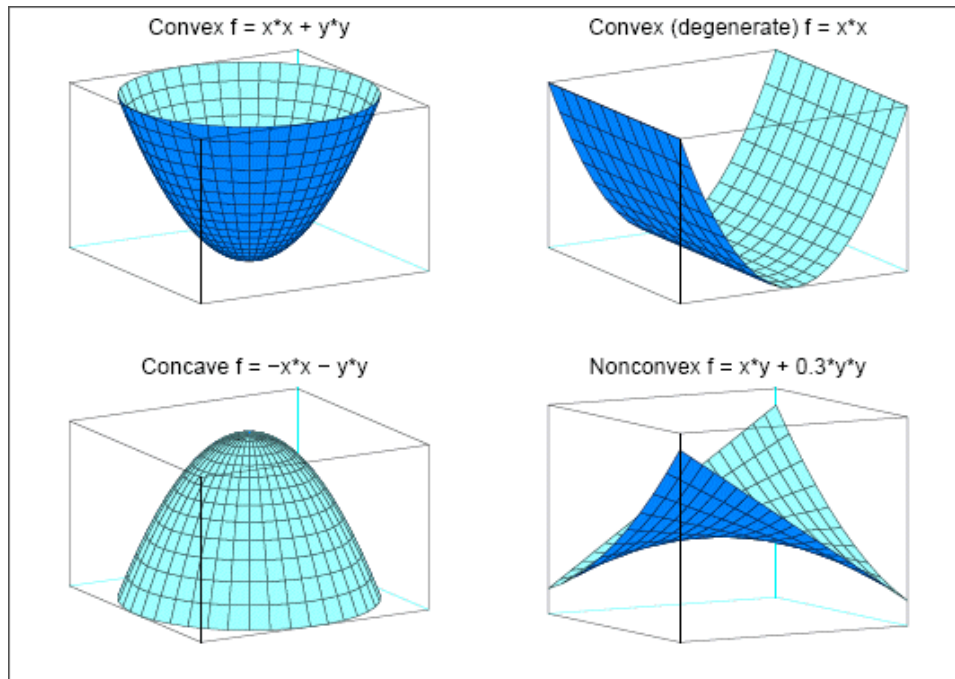
The order of constraints is insignificant. Some or all components of **l** or **u** (lower and upper bounds, respectively) can be omitted.

Getting Started: OPTQP Procedure

Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function $f(x_1, x_2)$ on the nonnegative quadrant, subject to two constraints:

$$
\begin{aligned}
\min \quad & 2x_1 + 3x_2 + x_1^2 + 10x_2^2 + 2.5x_1x_2 \\
\text{subject to} \quad & x_1 - x_2 \leq 1 \\
& x_1 + 2x_2 \geq 100 \\
& x_1 \geq 0 \\
& x_2 \geq 0
\end{aligned}
$$

The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

$$
\mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 100 \end{bmatrix}, \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}
$$

Carefully construct the quadratic matrix **Q**. Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

$$
\frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{Q}\mathbf{x} \equiv \frac{1}{2}\sum_{i,j=1}^{n} x_i\, q_{ij}\, x_j = \frac{1}{2}\sum_{i=1}^{n} q_{ii}\, x_i^2 + \sum_{i>j} x_i\, q_{ij}\, x_j
$$

The first expression

$$\frac{1}{2} \sum_{i=1}^{n} q_{ii} \, x_i^2$$

sums the main-diagonal elements. Thus, in this case you have

$$q_{11} = 2, \quad q_{22} = 20$$

Notice that the main-diagonal values are doubled in order to accommodate the 1/2 factor. Now the second term

$$\sum_{i>j} x_i \, q_{ij} \, x_j$$

sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal ($x_i \, x_j$, $i \neq j$) term in the objective function is $2.5 \, x_1 \, x_2$, so you have

$$q_{21} = 2.5$$

Notice that you do not need to specify the upper triangular part of the quadratic matrix.

Finally, the matrix of constraints is as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}$$

The SAS input data set with a quadratic programming system (QPS) format for the preceding problem can be expressed in the following manner:

```
data gsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
   datalines;
NAME        .        EXAMPLE    .            .        .
ROWS        .        .          .            .        .
N           OBJ      .          .            .        .
L           R1       .          .            .        .
G           R2       .          .            .        .
COLUMNS     .        .          .            .        .
            X1       R1         1.0          R2       1.0
            X1       OBJ        2.0          .        .
            X2       R1         -1.0         R2       2.0
            X2       OBJ        3.0          .        .
RHS         .        .          .            .        .
            RHS      R1         1.0          .        .
            RHS      R2         100          .        .
RANGES      .        .          .            .        .
BOUNDS      .        .          .            .        .
QUADOBJ     .        .          .            .        .
            X1       X1         2.0          .        .
            X1       X2         2.5          .        .
            X2       X2         20           .        .
ENDATA      .        .          .            .        .
;
```

For more details about the QPS-format data set, see Chapter 15, "The MPS-Format SAS Data Set."

Alternatively, if you have a QPS-format flat file named gs.qps, then the following call to the SAS macro %MPS2SASD translates that file into a SAS data set, named gsdata:

```
%mps2sasd(mpsfile =gs.qps, outdata = gsdata);
```

NOTE: The SAS macro %MPS2SASD is provided in SAS/OR software. See "Converting an MPS/QPS-Format File: %MPS2SASD" on page 608 for details.

You can use the following call to PROC OPTQP:

```
proc optqp data=gsdata
   primalout = gspout
   dualout   = gsdout;
run;
```

The procedure output is displayed in Figure 12.2.

Figure 12.2 Procedure Output

```
                    The OPTQP Procedure

                 Performance Information

          Execution Mode        Single-Machine
          Number of Threads              4


                    Problem Summary

          Problem Name                      EXAMPLE
          Objective Sense              Minimization
          Objective Function                    OBJ
          RHS                                   RHS

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

          Number of Constraints                   2
          LE  (<=)                                1
          EQ  (=)                                 0
          GE  (>=)                                1
          Range                                   0

          Constraint Coefficients                 4

          Hessian Diagonal Elements               2
          Hessian Elements Above the Diagonal     1
```

Figure 12.2 *continued*

```
                       Solution Summary

            Solver                             QP
            Algorithm              Interior Point
            Objective Function              OBJ
            Solution Status             Optimal
            Objective Value               15018

            Primal Infeasibility     1.573013E-16
            Dual Infeasibility       1.234237E-14
            Bound Infeasibility                 0
            Duality Gap              3.633377E-16
            Complementarity                     0

            Iterations                          6
            Presolve Time                    0.00
            Solution Time                    0.40
```

The optimal primal solution is displayed in Figure 12.3.

Figure 12.3 Optimal Solution

```
             Objective                                        Linear
             Function      RHS      Variable    Variable    Objective
     Obs        ID         ID         Name        Type     Coefficient

      1         OBJ        RHS         X1           N            2
      2         OBJ        RHS         X2           N            3


             Lower       Upper     Variable    Variable
     Obs     Bound       Bound       Value      Status

      1        0       1.7977E308      34          O
      2        0       1.7977E308      33          O
```

The SAS log shown in Figure 12.4 provides information about the problem, convergence information after each iteration, and the optimal objective value.

Figure 12.4 Iteration Log

```
NOTE: The problem EXAMPLE has 2 variables (0 free, 0 fixed).
NOTE: The problem has 2 constraints (1 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 4 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian
      elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 2 constraints, and 4 constraint
      coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                        Primal        Bound         Dual
      Iter   Complement  Duality Gap     Infeas       Infeas       Infeas
         0  3.58625E+03  4.88230E+00  1.02509E+00  1.03539E+02  2.31419E-15
         1  1.93453E+03  9.62224E-01  4.41582E-01  4.46020E+01  5.78549E-16
         2  2.21403E+03  1.22966E-01  4.41582E-03  4.46020E-01  5.59264E-15
         3  5.00197E+01  3.22723E-03  4.41582E-05  4.46020E-03  9.65019E-15
         4  4.99735E-01  3.23322E-05  4.41582E-07  4.46020E-05  2.51482E-14
         5  4.99724E-03  3.23323E-07  4.41582E-09  4.46020E-07  3.07007E-14
         6  0.00000E+00  3.63338E-16  1.57301E-16  0.00000E+00  1.23424E-14
NOTE: Optimal.
NOTE: Objective = 15018.
NOTE: The Interior Point solve time is 0.01 seconds.
NOTE: The data set WORK.GSPOUT has 2 observations and 9 variables.
NOTE: The data set WORK.GSDOUT has 2 observations and 10 variables.
```

See the section "Interior Point Algorithm: Overview" on page 487 and the section "Iteration Log for the OPTQP Procedure" on page 489 for more details about convergence information given by the iteration log.

Syntax: OPTQP Procedure

The following statements are available in the OPTQP procedure:

PROC OPTQP < *options* > **;**
    PERFORMANCE < *performance-options* > **;**

Functional Summary

Table 12.1 outlines the options available for the OPTQP procedure classified by function.

Table 12.1   Options in the OPTQP Procedure

Description	Option
Data Set Options	
Specifies a QPS-format input SAS data set	DATA=
Specifies a dual solution output SAS data set	DUALOUT=
Specifies whether the QP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution output SAS data set	PRIMALOUT=
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL
Control Options	
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the type of presolve	PRESOLVER=
Enables or disables iteration log	LOGFREQ=
Enables or disables printing summary	PRINTLEVEL=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=
Specifies units of CPU time or real time	TIMETYPE=

PROC OPTQP Statement

The following options can be specified in the PROC OPTQP statement.

DATA=_SAS-data-set_

  specifies the input SAS data set. This data set can also be created from a QPS-format flat file by using the SAS macro %MPS2SASD. If the DATA= option is not specified, PROC OPTQP uses the most recently created SAS data set. See Chapter 15, "The MPS-Format SAS Data Set," for more details.

DUALOUT=_SAS-data-set_

DOUT=_SAS-data-set_

  specifies the output data set to contain the dual solution. See the section "Output Data Sets" on page 485 for details.

LOGFREQ=_k_

PRINTFREQ=_k_

  specifies that the printing of the solution progress to the iteration log should occur after every _k_ iterations. The print frequency, _k_, is an integer between zero and the largest four-byte, signed integer,

which is $2^{31} - 1$. The value $k = 0$ disables the printing of the progress of the solution. The default value of this option is 1.

MAXITER=k

specifies the maximum number of predictor-corrector iterations performed by the interior point algorithm (see the section "Interior Point Algorithm: Overview" on page 487). The value k is an integer between 1 and the largest four-byte, signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=t

specifies an upper limit of t seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of t can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OBJSENSE=$option$

specifies whether the QP model is a minimization or a maximization problem. You specify OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set; see the section "ROWS Section" on page 601 for details. If the objective sense is specified differently in these two places, this option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTQP interprets and solves the quadratic program as a minimization problem.

PRESOLVER=$number \mid string$

PRESOL=$number \mid string$

specifies one of the following presolve options:

number	string	Description
0	NONE	Disables the presolver.
−1	AUTOMATIC	Applies the presolver by using default setting.
1	BASIC	Applies the basic presolver.
2	MODERATE	Applies the moderate presolver.
3	AGGRESSIVE	Applies the aggressive presolver.

You can specify the option either by a word or by integers from −1 to 3. The default option is AUTOMATIC.

PRIMALOUT=$SAS\text{-}data\text{-}set$

POUT=$SAS\text{-}data\text{-}set$

specifies the output data set to contain the primal solution. See the section "Output Data Sets" on page 485 for details.

PRINTLEVEL=0 | 1

specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

For details about the ODS tables created by PROC OPTQP, see the section "ODS Tables" on page 489.

SAVE_ONLY_IF_OPTIMAL

specifies that the PRIMALOUT= and DUALOUT= data sets be saved only if the final solution obtained by the solver at termination is optimal. If the PRIMALOUT= or DUALOUT= option is specified, and this option is not specified, then the output data sets will only contain solution values at optimality. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets will not contain an intermediate solution.

STOP_DG=δ

specifies the desired relative duality gap, $\delta \in [1E–9, 1E–4]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 487 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in [1E–9, 1E–4]$. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 487 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in [1E–9, 1E–4]$. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 487 for details.

TIMETYPE=number | string

specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTQP_ macro variable in a PROC OPTQP call. Table 12.3 describes the valid values of the TIMETYPE= option.

Table 12.3 Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The default value of the TIMETYPE= option depends on the value of the NTHREADS= option in the PERFORMANCE statement. See the section "PERFORMANCE Statement" on page 28 for more information about the NTHREADS= option.

If you specify a value greater than 1 for the NTHREADS= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for the NTHREADS= option, the default value of the TIMETYPE= option is CPU.

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for single-machine model, and requests detailed results about the performance characteristics of the OPTQP procedure.

The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics." The OPTQP procedure only supports the deterministic mode of the PARALLELMODE= option in the PERFORMANCE statement.

Details: OPTQP Procedure

Output Data Sets

This section describes the PRIMALOUT= and DUALOUT= output data sets. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets do not contain an intermediate solution.

Definitions of Variables in the PRIMALOUT= Data Set

The PRIMALOUT= data set contains the primal solution to the quadratic programming (QP) model. The variables in the data set have the following names and meanings.

_OBJ_ID_

specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. See the section "ROWS Section" on page 601 for details.

NOTE: PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_

specifies the name of the variable that contains the right-hand-side value of each constraint. See the section "ROWS Section" on page 601 for details.

VAR

specifies the name of the decision variable.

TYPE

specifies the type of the decision variable. _TYPE_ can take one of the following values:

N   nonnegative variable

D   bounded variable with either lower or upper bound

F   free variable

X   fixed variable

O   other

OBJCOEF

specifies the coefficient of the decision variable in the linear component of the objective function.

LBOUND
> specifies the lower bound on the decision variable.

UBOUND
> specifies the upper bound on the decision variable.

VALUE
> specifies the value of the decision variable.

STATUS
> specifies the status of the decision variable. _STATUS_ can indicate one of the following two cases:
>
> O   The QP problem is optimal.
>
> I   The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

Definitions of Variables in the DUALOUT= Data Set

The DUALOUT= data set contains the dual solution to the QP model. Information about the objective rows of the QP problems is not included. The variables in the data set have the following names and meanings.

_OBJ_ID_
> specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. See the section "ROWS Section" on page 601 for details.
>
> **NOTE:** PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
> specifies the name of the variable that contains the right-hand-side value of each constraint. See the section "ROWS Section" on page 601 for details.

ROW
> specifies the name of the constraint. See the section "ROWS Section" on page 601 for details.

TYPE
> specifies the type of the constraint. _TYPE_ can take one of the following values:
>
> L   "less than or equals" constraint
>
> E   equality constraint
>
> G   "greater than or equals" constraint
>
> R   ranged constraint (both "less than or equals" and "greater than or equals")
>
> See the sections "ROWS Section" on page 601 and "RANGES Section (Optional)" on page 604 for details.

RHS

    specifies the value of the right-hand side of the constraints. It takes a missing value for a ranged constraint.

_L_RHS_

    specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_

    specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

VALUE

    specifies the value of the dual variable associated with the constraint.

STATUS

    specifies the status of the constraint. _STATUS_ can indicate one of the following two cases:

    O   The QP problem is optimal.

    I   The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

ACTIVITY

    specifies the value of a constraint. In other words, the value of _ACTIVITY_ for the ith constraint is equal to $\mathbf{a}_i^\mathrm{T}\mathbf{x}$, where $\mathbf{a}_i$ refers to the ith row of the constraints matrix and $\mathbf{x}$ denotes the vector of current decision variable values.

Interior Point Algorithm: Overview

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

$$
\begin{array}{rl}
\min & \frac{1}{2}\mathbf{x}^\mathrm{T}\mathbf{Q}\mathbf{x} + \mathbf{c}^\mathrm{T}\mathbf{x} \\
\text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\
& \mathbf{x} \geq \mathbf{0}
\end{array}
$$

The corresponding dual is as follows:

$$
\begin{array}{rlccccc}
\max & -\frac{1}{2}\mathbf{x}^\mathrm{T}\mathbf{Q}\mathbf{x} & + & \mathbf{b}^\mathrm{T}\mathbf{y} & & & \\
\text{subject to} & -\mathbf{Q}\mathbf{x} & + & \mathbf{A}^\mathrm{T}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\
& & & \mathbf{y} & \geq & \mathbf{0} \\
& & & \mathbf{w} & \geq & \mathbf{0}
\end{array}
$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of slack variables in the dual problem.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$
\begin{aligned}
\mathbf{Ax} - \mathbf{s} &= \mathbf{b} \quad \text{(primal feasibility)} \\
-\mathbf{Qx} + \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} &= \mathbf{c} \quad \text{(dual feasibility)} \\
\mathbf{WXe} &= \mathbf{0} \quad \text{(complementarity)} \\
\mathbf{SYe} &= \mathbf{0} \quad \text{(complementarity)} \\
\mathbf{x,\ y,\ w,\ s} &\geq \mathbf{0}
\end{aligned}
$$

where $\mathbf{e} \equiv (1,\ldots,1)^{\mathrm{T}}$ is of appropriate dimension and $\mathbf{s} \in \mathbb{R}^{m}$ is the vector of primal slack variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding x, y, w, and s on the main diagonal and zero elsewhere, as in the following example:

$$
\mathbf{X} \equiv
\begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

$$
\begin{bmatrix}
\mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\
\mathbf{A}^{\mathrm{T}} & -\mathbf{Q} - \mathbf{X}^{-1}\mathbf{W}
\end{bmatrix}
\begin{bmatrix}
\Delta\mathbf{y} \\
\Delta\mathbf{x}
\end{bmatrix}
=
\begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTQP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point solver is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector v:

- relative primal infeasibility measure α:

$$
\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}
$$

- relative dual infeasibility measure β:

$$
\beta = \frac{\|\mathbf{Qx} + \mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}
$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x} - \mathbf{b}^T \mathbf{y}|}{|\frac{1}{2}\mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Iteration Log for the OPTQP Procedure

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter	indicates the iteration number.
Complement	indicates the (absolute) complementarity.
Duality Gap	indicates the (relative) duality gap.
Primal Infeas	indicates the (relative) primal infeasibility measure.
Bound Infeas	indicates the (relative) bound infeasibility measure.
Dual Infeas	indicates the (relative) dual infeasibility measure.

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the option MAXITER= or MAXTIME=. If the complementarity or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

ODS Tables

PROC OPTQP creates three ODS (Output Delivery System) tables by default. The first table, ProblemSummary, is a summary of the input QP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see the *SAS Output Delivery System: User's Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. See the section "Problem Statistics" on page 492 for more information.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 12.4 lists all the ODS tables that can be produced by the OPTQP procedure, along with the statement and option specifications required to produce each table.

Table 12.4 ODS Tables Produced by PROC OPTQP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input QP problem	PROC OPTQP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTQP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTQP	PRINTLEVEL=2
PerformanceInfo	List of performance options and their values	PROC OPTQP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical output of PROC OPTQP is shown in Output 12.5.

Figure 12.5 Typical OPTQP Output

```
                    The OPTQP Procedure

                  Performance Information

          Execution Mode        Single-Machine
          Number of Threads              4


                     Problem Summary

        Problem Name                        BANDM
        Objective Sense              Minimization
        Objective Function                 ....1
        RHS                            ZZZZ0001

        Number of Variables                  472
        Bounded Above                          0
        Bounded Below                        472
        Bounded Above and Below                0
        Free                                   0
        Fixed                                  0

        Number of Constraints                305
        LE  (<=)                               0
        EQ  (=)                              305
        GE  (>=)                               0
        Range                                  0

        Constraint Coefficients             2494

        Hessian Diagonal Elements             25
        Hessian Elements Above the Diagonal   16
```

Figure 12.5 *continued*

```
                          Solution Summary

           Solver                                QP
           Algorithm                 Interior Point
           Objective Function               ....1
           Solution Status               Optimal
           Objective Value           16352.342037

           Primal Infeasibility       1.127493E-12
           Dual Infeasibility         1.684333E-12
           Bound Infeasibility        4.520378E-16
           Duality Gap                 1.25305E-12
           Complementarity            1.2956352E-8

           Iterations                          23
           Presolve Time                     0.00
           Solution Time                     0.23
```

You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTQP runs. The output data sets that correspond to the preceding output are shown in Output 12.6, where you can also find (in the row following the heading of each data set in the display) the variable names that are used in the table definition (template) of each table.

Figure 12.6 ODS Output Data Sets

```
                              Problem Summary

     Obs    Label1                                cValue1              nValue1

      1     Problem Name                          BANDM                      .
      2     Objective Sense                       Minimization               .
      3     Objective Function                    ....1                      .
      4     RHS                                   ZZZZ0001                   .
      5                                                                      .
      6     Number of Variables                   472               472.000000
      7     Bounded Above                         0                          0
      8     Bounded Below                         472               472.000000
      9     Bounded Above and Below               0                          0
     10     Free                                  0                          0
     11     Fixed                                 0                          0
     12                                                                      .
     13     Number of Constraints                 305               305.000000
     14     LE  (<=)                              0                          0
     15     EQ  (=)                               305               305.000000
     16     GE  (>=)                              0                          0
     17     Range                                 0                          0
     18                                                                      .
     19     Constraint Coefficients               2494             2494.000000
     20                                                                      .
     21     Hessian Diagonal Elements             25                 25.000000
     22     Hessian Elements Above the Diagonal   16                 16.000000


                              Solution Summary

       Obs    Label1                       cValue1              nValue1

        1     Solver                    QP                            .
        2     Algorithm                 Interior Point                .
        3     Objective Function        ....1                         .
        4     Solution Status           Optimal                       .
        5     Objective Value           16352.342037              16352
        6                                                             .
        7     Primal Infeasibility      1.127493E-12      1.127493E-12
        8     Dual Infeasibility        1.684333E-12      1.684333E-12
        9     Bound Infeasibility       4.520378E-16      4.520378E-16
       10     Duality Gap               1.25305E-12        1.25305E-12
       11     Complementarity           1.2956352E-8      1.2956352E-8
       12                                                             .
       13     Iterations                23                  23.000000
       14     Presolve Time                   0.00                   0
       15     Solution Time                   0.23            0.230999
```

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the

remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTQP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Output 12.7 demonstrates the contents of the ODS table ProblemStatistics.

Figure 12.7 ODS Table ProblemStatistics

```
                        The OPTQP Procedure

                        Problem Statistics

        Number of Constraint Matrix Nonzeros          4
        Maximum Constraint Matrix Coefficient         2
        Minimum Constraint Matrix Coefficient         1
        Average Constraint Matrix Coefficient       1.25

        Number of Linear Objective Nonzeros           2
        Maximum Linear Objective Coefficient          3
        Minimum Linear Objective Coefficient          2
        Average Linear Objective Coefficient        2.5

        Number of Lower Triangular Hessian Nonzeros   1
        Number of Diagonal Hessian Nonzeros           2
        Maximum Hessian Coefficient                  20
        Minimum Hessian Coefficient                   2
        Average Hessian Coefficient                6.75

        Number of RHS Nonzeros                        2
        Maximum RHS                                 100
        Minimum RHS                                   1
        Average RHS                                50.5

        Maximum Number of Nonzeros per Column         2
        Minimum Number of Nonzeros per Column         2
        Average Number of Nonzeros per Column         2

        Maximum Number of Nonzeros per Row            2
        Minimum Number of Nonzeros per Row            2
        Average Number of Nonzeros per Row            2
```

Macro Variable _OROPTQP_

The OPTQP procedure defines a macro variable named _OROPTQP_. This variable contains a character string that indicates the status of the procedure. The various terms of the variable are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take the following value:

| IP | The interior point algorithm produced the solution data. |

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The maximum time limit was reached.
FAILED	The solver failed to converge, possibly due to numerical issues.
NONCONVEX	The quadratic matrix is nonconvex (minimization).
NONCONCAVE	The quadratic matrix is nonconcave (maximization).

OBJECTIVE

indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY

indicates the (relative) infeasibility of the primal constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 487 for details.

DUAL_INFEASIBILITY

indicates the (relative) infeasibility of the dual constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 487 for details.

BOUND_INFEASIBILITY

indicates the (relative) violation by the solution of the lower or upper bounds (or both). See the section "Interior Point Algorithm: Overview" on page 487 for details.

DUALITY_GAP

indicates the (relative) duality gap. See the section "Interior Point Algorithm: Overview" on page 487 for details.

COMPLEMENTARITY

indicates the (absolute) complementarity at the optimal solution. See the section "Interior Point Algorithm: Overview" on page 487 for details.

ITERATIONS

indicates the number of iterations required to solve the problem.

PRESOLVE_TIME

indicates the time taken for preprocessing (seconds).

SOLUTION_TIME

indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTQP Procedure

This section contains examples that illustrate the use of the OPTQP procedure. Example 12.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTQP. Example 12.2 and Example 12.3 explain in detail how to model the portfolio optimization and selection problems.

Example 12.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $m > n$. Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes $\|\mathbf{Ax} - \mathbf{b}\|_2^2$.

This problem is called a least squares problem for the following reason. Let $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ be defined as previously. Let $k_i(x)$ be the kth component of the vector $\mathbf{Ax} - \mathbf{b}$:

$$k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i, \ i = 1, 2, \ldots, m$$

By definition of the Euclidean norm, the objective function can be expressed as follows:

$$\|\mathbf{Ax} - \mathbf{b}\|_2^2 = \sum_{i=1}^{m} k_i(x)^2$$

Therefore, the function you minimize is the sum of squares of *m* terms $k_i(x)$; hence the term least squares. The following example is an illustration of the *linear* least squares problem; that is, each of the terms k_i is a linear function of *x*.function $\sum_{ij} a_{ij} x_j$ plus a constant, $-b_i$.

Consider the following least squares problem defined by

$$A = \begin{bmatrix} 4 & 0 \\ -1 & 1 \\ 3 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

This translates to the following set of linear equations:

$$4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1$$

The corresponding least squares problem is

$$\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2$$

The preceding objective function can be expanded to

$$\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2$$

In addition, you impose the following constraint so that the equation $3x_1 + 2x_2 = 1$ is satisfied within a tolerance of 0.1:

$$0.9 \le 3x_1 + 2x_2 \le 1.1$$

You can create the QPS-format input data set by using the following SAS statements:

```
data lsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
   datalines;
NAME         .      LEASTSQ       .           .        .
ROWS         .      .             .           .        .
N            OBJ    .             .           .        .
G            EQ3    .             .           .        .
COLUMNS      .      .             .           .        .
.            X1     OBJ          -14          EQ3      3
.            X2     OBJ          -4           EQ3      2
RHS          .      .             .           .        .
.            RHS    OBJ          -2           EQ3      0.9
RANGES       .      .             .           .        .
.            RNG    EQ3           0.2         .        .
BOUNDS       .      .             .           .        .
FR           BND1   X1            .           .        .
FR           BND1   X2            .           .        .
QUADOBJ      .      .             .           .        .
.            X1     X1            52          .        .
.            X1     X2            10          .        .
.            X2     X2            10          .        .
ENDATA       .      .             .           .        .
;
```

The decision variables x_1 and x_2 are free, so they have bound type FR in the BOUNDS section of the QPS-format data set.

You can use the following SAS statements to solve the least squares problem:

```
proc optqp data=lsdata
   printlevel = 0
   primalout = lspout;
run;
```

The optimal solution is displayed in Output 12.1.1.

Output 12.1.1 Solution to the Least Squares Problem

```
                        Primal Solution

            Objective                              Linear
            Function    RHS    Variable   Variable  Objective
    Obs        ID        ID      Name       Type    Coefficient

     1        OBJ       RHS      X1          F         -14
     2        OBJ       RHS      X2          F          -4


                       Upper      Variable    Variable
    Obs    Lower Bound   Bound      Value      Status

     1    -1.7977E308  1.7977E308   0.23810       O
     2    -1.7977E308  1.7977E308   0.16190       O
```

The iteration log is shown in Output 12.1.2.

Output 12.1.2 Iteration Log

```
NOTE: The problem LEASTSQ has 2 variables (2 free, 0 fixed).
NOTE: The problem has 1 constraints (0 LE, 0 EQ, 0 GE, 1 range).
NOTE: The problem has 2 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian
      elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 1 constraints, and 2 constraint
      coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                    Primal       Bound        Dual
      Iter   Complement  Duality Gap  Infeas      Infeas      Infeas
         0  1.91812E-02  5.89357E-03  1.96367E-08  0.00000E+00  3.53901E-04
         1  9.04861E-04  2.83115E-04  7.25763E-10  0.00000E+00  1.30555E-05
         2  1.53700E-05  4.94409E-06  7.77909E-12  0.00000E+00  1.30555E-07
         3  1.53570E-07  4.93971E-08  8.81570E-14  0.00000E+00  1.30562E-09
NOTE: Optimal.
NOTE: Objective = 0.0095238095.
NOTE: The Interior Point solve time is 0.01 seconds.
NOTE: The data set WORK.LSPOUT has 2 observations and 9 variables.
```

Example 12.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio's total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $\mathcal{B}$ be the amount of capital you have, $\mathbf{R}$ be the random vector of asset returns over some period, and $\mathbf{r}$ be the expected value of $\mathbf{R}$. Let G be the minimum growth you hope to obtain, and $\mathcal{C}$ be the covariance matrix of $\mathbf{R}$. The objective function is $\mathrm{Var}\left(\sum_{i=1}^{n} x_i R_i\right)$, which can be equivalently denoted as $\mathbf{x}^T \mathcal{C} \mathbf{x}$.

Assume, for example, $n = 4$. Let $\mathcal{B} = 10{,}000$, $G = 1000$, $\mathbf{r} = [0.05, -0.2, 0.15, 0.30]$, and

$$
\mathcal{C} = \begin{bmatrix}
0.08 & -0.05 & -0.05 & -0.05 \\
-0.05 & 0.16 & -0.02 & -0.02 \\
-0.05 & -0.02 & 0.35 & 0.06 \\
-0.05 & -0.02 & 0.06 & 0.35
\end{bmatrix}
$$

The QP formulation can be written as follows:

$$
\begin{aligned}
\min \quad & 0.08x_1^2 - 0.1x_1x_2 - 0.1x_1x_3 - 0.1x_1x_4 + \\
& 0.16x_2^2 - 0.04x_2x_3 - 0.04x_2x_4 + 0.35x_3^2 + \\
& 0.12x_3x_4 + 0.35x_4^2
\end{aligned}
$$

subject to

$$
\begin{array}{lrcrcrcrcl}
(\text{budget}) & x_1 & + & x_2 & + & x_3 & + & x_4 & \leq & 10000 \\
(\text{growth}) & 0.05x_1 & - & 0.2x_2 & + & 0.15x_3 & + & 0.30x_4 & \geq & 1000 \\
& & & & & & x_1, x_2, x_3, x_4 & \geq & 0
\end{array}
$$

The corresponding QPS-format input data set is as follows:

```
data portdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME    .           PORT        .           .           .
ROWS    .           .           .           .           .
N       OBJ.FUNC    .           .           .           .
L       BUDGET      .           .           .           .
G       GROWTH      .           .           .           .
COLUMNS .           .           .           .           .
.       X1          BUDGET      1.0         GROWTH      0.05
.       X2          BUDGET      1.0         GROWTH      -.20
.       X3          BUDGET      1.0         GROWTH      0.15
.       X4          BUDGET      1.0         GROWTH      0.30
RHS     .           .           .           .           .
.       RHS         BUDGET      10000       .           .
.       RHS         GROWTH      1000        .           .
```

```
RANGES  .        .          .             .          .
BOUNDS  .        .          .             .          .
QUADOBJ .        .          .             .          .
   .    X1       X1       0.16            .          .
   .    X1       X2      -.10             .          .
   .    X1       X3      -.10             .          .
   .    X1       X4      -.10             .          .
   .    X2       X2       0.32            .          .
   .    X2       X3      -.04             .          .
   .    X2       X4      -.04             .          .
   .    X3       X3       0.70            .          .
   .    X3       X4       0.12            .          .
   .    X4       X4       0.70            .          .
ENDATA  .        .          .             .          .
   ;
```

Use the following SAS statements to solve the problem:

```
proc optqp data=portdata
  primalout = portpout
  printlevel = 0
  dualout   = portdout;
run;
```

The optimal solution is shown in Output 12.2.1.

Output 12.2.1 Portfolio Optimization

```
                        The OPTQP Procedure
                          Primal Solution

                 Objective                                   Linear
                 Function      RHS      Variable   Variable  Objective
         Obs        ID          ID        Name       Type    Coefficient

          1     OBJ.FUNC       RHS        X1          N           0
          2     OBJ.FUNC       RHS        X2          N           0
          3     OBJ.FUNC       RHS        X3          N           0
          4     OBJ.FUNC       RHS        X4          N           0


                 Lower        Upper       Variable   Variable
         Obs     Bound        Bound        Value      Status

          1        0       1.7977E308     3452.86       O
          2        0       1.7977E308        0.00       O
          3        0       1.7977E308     1068.81       O
          4        0       1.7977E308     2223.45       O
```

Thus, the minimum variance portfolio that earns an expected return of at least 10% is $x_1 = 3452.86$, $x_2 = 0$, $x_3 = 1068.81$, $x_4 = 2223.45$. Asset 2 gets nothing, because its expected return is -20% and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop

the nonnegativity assumption? You need to update the BOUNDS section in the existing QPS-format data set to indicate that the decision variables are free.

```
   ...
   RANGES .        .       .        .        .
   BOUNDS .        .       .        .        .
   FR     BND1     X1      .        .        .
   FR     BND1     X2      .        .        .
   FR     BND1     X3      .        .        .
   FR     BND1     X4      .        .        .
   QUADOBJ .       .       .        .        .
   ...
```

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and "more than 100%" in high-mean assets. You can see in the optimal solution displayed in Output 12.2.2 that the decision variable x_2, denoting Asset 2, is equal to -1563.61, which means short sale of that asset.

Output 12.2.2 Portfolio Optimization with Short-Sale Option

The OPTQP Procedure
Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient
1	OBJ.FUNC	RHS	X1	F	0
2	OBJ.FUNC	RHS	X2	F	0
3	OBJ.FUNC	RHS	X3	F	0
4	OBJ.FUNC	RHS	X4	F	0

Obs	Lower Bound	Upper Bound	Variable Value	Variable Status
1	−1.7977E308	1.7977E308	1684.35	O
2	−1.7977E308	1.7977E308	−1563.61	O
3	−1.7977E308	1.7977E308	682.51	O
4	−1.7977E308	1.7977E308	1668.95	O

Example 12.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector $\mathbf{c} = [200, 300, 500]$, the amount of asset bought and sold is denoted by b_i and s_i, respectively, and the net investment in each asset is denoted by x_i and is defined by the following relation:

$$x_i - b_i + s_i = c_i, \quad i = 1, 2, 3$$

Suppose you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix $\mathcal{C}$ be defined as

$$\mathcal{C} = \begin{bmatrix} 0.027489 & -0.00874 & -0.00015 \\ -0.00874 & 0.109449 & -0.00012 \\ -0.00015 & -0.00012 & 0.000766 \end{bmatrix}$$

Assume that you hope to obtain at least 12% growth. Let $\mathbf{r} = [1.109048, 1.169048, 1.074286]$ be the vector of expected return on the three assets, and let $\mathcal{B}=1000$ be the available funds. Mathematically, this problem can be written in the following manner:

$$\min \quad 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2$$
$$-0.00024x_2x_3 + 0.000766x_3^2$$

subject to

(return)	$\sum_{i=1}^{3} r_i x_i$	$\geq$	$1.12\mathcal{B}$
(budget)	$\sum_{i=1}^{3} x_i + \sum_{i=1}^{3} 0.01(b_i + s_i)$	$=$	$\mathcal{B}$
(balance)	$x_i - b_i + s_i$	$=$	$c_i, \quad i = 1, 2, 3$
	x_i, b_i, s_i	$\geq$	$0, \quad i = 1, 2, 3$

The QPS-format input data set is as follows:

```
data potrdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME       .           POTRAN    .            .           .
ROWS       .           .         .            .           .
N          OBJ.FUNC    .         .            .           .
G          RETURN      .         .            .           .
E          BUDGET      .         .            .           .
E          BALANC1     .         .            .           .
E          BALANC2     .         .            .           .
E          BALANC3     .         .            .           .
COLUMNS    .           .         .            .           .
   .       X1          RETURN    1.109048     BUDGET      1.0
   .       X1          BALANC1   1.0          .           .
   .       X2          RETURN    1.169048     BUDGET      1.0
   .       X2          BALANC2   1.0          .           .
```

```
       .        X3        RETURN    1.074286    BUDGET     1.0
       .        X3        BALANC3   1.0          .          .
       .        B1        BUDGET     .01        BALANC1    -1.0
       .        B2        BUDGET     .01        BALANC2    -1.0
       .        B3        BUDGET     .01        BALANC3    -1.0
       .        S1        BUDGET     .01        BALANC1     1.0
       .        S2        BUDGET     .01        BALANC2     1.0
       .        S3        BUDGET     .01        BALANC3     1.0
RHS             .         .          .           .          .
       .        RHS       RETURN    1120         .          .
       .        RHS       BUDGET    1000         .          .
       .        RHS       BALANC1   200          .          .
       .        RHS       BALANC2   300          .          .
       .        RHS       BALANC3   500          .          .
RANGES          .         .          .           .          .
BOUNDS          .         .          .           .          .
QUADOBJ   .     .         .          .           .          .
       .        X1        X1        0.054978     .          .
       .        X1        X2        -.01748      .          .
       .        X1        X3        -.0003       .          .
       .        X2        X2        0.218898     .          .
       .        X2        X3        -.00024      .          .
       .        X3        X3        0.001532     .          .
ENDATA          .         .          .           .          .
;
```

Use the following SAS statements to solve the problem:

```
proc optqp data=potrdata
  primalout = potrpout
  printlevel = 0
  dualout   = potrdout;
run;
```

The optimal solution is displayed in

Output 12.3.1 Portfolio Selection with Transactions

```
                      The OPTQP Procedure
                      Primal Solution

             Objective                                    Linear
             Function    RHS     Variable   Variable    Objective
    Obs         ID       ID        Name       Type      Coefficient

     1       OBJ.FUNC    RHS        X1          N            0
     2       OBJ.FUNC    RHS        X2          N            0
     3       OBJ.FUNC    RHS        X3          N            0
     4       OBJ.FUNC    RHS        B1          N            0
     5       OBJ.FUNC    RHS        B2          N            0
     6       OBJ.FUNC    RHS        B3          N            0
     7       OBJ.FUNC    RHS        S1          N            0
     8       OBJ.FUNC    RHS        S2          N            0
     9       OBJ.FUNC    RHS        S3          N            0


             Lower        Upper      Variable    Variable
    Obs      Bound        Bound        Value      Status

     1         0       1.7977E308    397.584        O
     2         0       1.7977E308    406.115        O
     3         0       1.7977E308    190.165        O
     4         0       1.7977E308    197.584        O
     5         0       1.7977E308    106.115        O
     6         0       1.7977E308      0.000        O
     7         0       1.7977E308      0.000        O
     8         0       1.7977E308      0.000        O
     9         0       1.7977E308    309.835        O
```

References

Freund, R. W. (1991), "On Polynomial Preconditioning and Asymptotic Convergence Factors for Indefinite Hermitian Matrices," *Linear Algebra and Its Applications*, 154–156, 259–288.

Freund, R. W. and Jarre, F. (1997), "A QMR-Based Interior Point Algorithm for Solving Linear Programs," *Mathematical Programming*, 76, 183–210.

Freund, R. W. and Nachtigal, N. M. (1996), "QMRPACK: A Package of QMR Algorithms," *ACM Transactions on Mathematical Software*, 22, 46–77.

Vanderbei, R. J. (1999), "LOQO: An Interior Point Code for Quadratic Programming," *Optimization Methods and Software*, 11, 451–484.

Wright, S. J. (1996), *Primal-Dual Interior Point Algorithms*, Philadelphia: SIAM Publications.

Chapter 13
The Decomposition Algorithm

Contents

Overview: Decomposition Algorithm

The SAS/OR decomposition algorithm (DECOMP) provides an alternative method for solving linear programs (LPs) and mixed integer linear programs (MILPs) by exploiting the ability to efficiently solve a relaxation of

the original problem. The algorithm is available as an option in the OPTMODEL, OPTLP and OPTMILP procedures and is based on the methodology described in Galati (2009).

A standard linear or mixed integer linear program has the formulation

$$
\begin{array}{lllll}
\text{minimize} & \mathbf{c}^\top\mathbf{x} & + & \mathbf{f}^\top\mathbf{y} & \\
\text{subject to} & \mathbf{Dx} & + & \mathbf{By} & \{\geq, =, \leq\} \quad \mathbf{d} \quad \text{(master)} \\
& \mathbf{Ax} & & & \{\geq, =, \leq\} \quad \mathbf{b} \quad \text{(subproblem)} \\
& l_i^x \leq x_i \leq u_i^x, & x_i \in \mathbb{Z} & i \in \mathcal{S}^x & \\
& l_i^y \leq y_i \leq u_i^y, & y_i \in \mathbb{Z} & i \in \mathcal{S}^y &
\end{array}
$$

where

$\mathbf{x} \in \mathbb{R}^n$	is the vector of structural variables
$\mathbf{y} \in \mathbb{R}^s$	is the vector of master-only structural variables
$\mathbf{c} \in \mathbb{R}^n$	is the vector of objective function coefficients that are associated with variables $\mathbf{x}$
$\mathbf{f} \in \mathbb{R}^s$	is the vector of objective function coefficients that are associated with variables $\mathbf{y}$
$\mathbf{D} \in \mathbb{R}^{t \times n}$	is the matrix of master constraint coefficients that are associated with variables $\mathbf{x}$
$\mathbf{B} \in \mathbb{R}^{t \times s}$	is the matrix of master constraint coefficients that are associated with variables $\mathbf{y}$
$\mathbf{A} \in \mathbb{R}^{m \times n}$	is the matrix of subproblem constraint coefficients
$\mathbf{d} \in \mathbb{R}^t$	is the vector of master constraints' right-hand sides
$\mathbf{b} \in \mathbb{R}^m$	is the vector of subproblem constraints' right-hand sides
$\mathbf{l}^x \in \mathbb{R}^n$	is the vector of lower bounds on variables $\mathbf{x}$
$\mathbf{u}^x \in \mathbb{R}^n$	is the vector of upper bounds on variables $\mathbf{x}$
$\mathbf{l}^y \in \mathbb{R}^s$	is the vector of lower bounds on variables $\mathbf{y}$
$\mathbf{u}^y \in \mathbb{R}^s$	is the vector of upper bounds on variables $\mathbf{y}$
$\mathcal{S}^x$	is a subset of the set $\{1, \ldots, n\}$ of indices on variables $\mathbf{x}$
$\mathcal{S}^y$	is a subset of the set $\{1, \ldots, s\}$ of indices on variables $\mathbf{y}$

A relaxation of the preceding mathematical program can be formed by removing the master constraints, which are defined by the matrices $\mathbf{D}$ and $\mathbf{B}$. The resulting constraint system, defined by the matrix $\mathbf{A}$, forms the subproblem, which can often be solved much more efficiently than the entire original problem. This is the one of the key motivators for using the decomposition algorithm.

The decomposition algorithm works by finding convex combinations of extreme points of the subproblem polyhedron that satisfy the constraints defined in the master. For MILP subproblems, the strength of the relaxation is another important motivator for using this method. If the subproblem polyhedron defines feasible solutions that are close to the original feasible space, the chance of success for the algorithm increases.

The region that defines the subproblem space is often separable. That is, the formulation of the preceding mathematical program can be written in *block-angular* form as follows:

$$
\begin{array}{llllllll}
\text{minimize} & \mathbf{c}^1\mathbf{x}^1 & + & \mathbf{c}^2\mathbf{x}^2 & + & \ldots & + & \mathbf{c}^\kappa\mathbf{x}^\kappa & + & \mathbf{f}^\top\mathbf{y} \\
\text{subject to} & \mathbf{D}^1\mathbf{x}^1 & + & \mathbf{D}^2\mathbf{x}^2 & + & \ldots & & \mathbf{D}^\kappa\mathbf{x}^\kappa & + & \mathbf{By} & \{\geq, =, \leq\} & \mathbf{d} \\
& \mathbf{A}^1\mathbf{x}^1 & & & & & & & & & \{\geq, =, \leq\} & \mathbf{b}^1 \\
& & & \mathbf{A}^2\mathbf{x}^2 & & & & & & & \{\geq, =, \leq\} & \mathbf{b}^2 \\
& & & & & \ddots & & & & & \{\geq, =, \leq\} & \vdots \\
& & & & & & & \mathbf{A}^\kappa\mathbf{x}^\kappa & & & \{\geq, =, \leq\} & \mathbf{b}^\kappa \\
& l_i^x \leq x_i \leq u_i^x, & x_i \in \mathbb{Z} & i \in \mathcal{S}^x \\
& l_i^y \leq y_i \leq u_i^y, & y_i \in \mathbb{Z} & i \in \mathcal{S}^y
\end{array}
$$

where $K = \{1, \ldots, \kappa\}$ defines a partition of the constraints (and variables) into independent subproblems (blocks) such that $\mathbf{A} = [\mathbf{A}^1 \ldots \mathbf{A}^\kappa], \mathbf{D} = [\mathbf{D}^1 \ldots \mathbf{D}^\kappa], \mathbf{c} = [\mathbf{c}^1 \ldots \mathbf{c}^\kappa]$, and $\mathbf{x} = [\mathbf{x}^1 \ldots \mathbf{x}^\kappa]$. This type of

structure is fairly common in modeling mathematical programs. For example, consider a model that defines a workplace with separate departmental restrictions (defined as the subproblem constraints), which are coupled together by a company-wide budget across departments (defined as the master constraint). By relaxing the budget (master) constraint, the decomposition algorithm can take advantage of the fact that the decoupled subproblems are separable, and it can process them in parallel. A special case of block-angular form, called *block-diagonal*, occurs when the set of master constraints is empty. In this special case, the subproblem matrices define the entire original problem.

An important indicator of a problem that is well suited for decomposition is the amount by which the subproblems cover the original problem with respect to both variables and constraints in the original presolved model. This value, expressed as a percentage of the original model is known as the *coverage*. For LPs, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. For MILPs, the correlation between performance and coverage is more difficult to determine, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system. Regardless, it is unlikely that the decomposition algorithm outperforms more standard methods (such as branch-and-cut) for problems with small coverage.

The primary input and output for the decomposition algorithm are identical to those needed and produced by the OPTLP, OPTMILP, and OPTMODEL procedures. For more information, see the sections "Data Input and Output" on page 373, "Data Input and Output" on page 433, "Details: LP Solver" on page 192, and "Details: MILP Solver" on page 259. The only additional input that can be provided for the decomposition algorithm is an explicit definition of the partition of the subproblem constraints. The following section gives a simple example of providing this input for both PROC OPTMILP and PROC OPTMODEL.

Getting Started: Decomposition Algorithm

This example illustrates how you can use the decomposition algorithm to solve a simple mixed integer linear program. Suppose you want to solve the following problem:

$$
\begin{array}{rlcccccccccl}
\max & x_{11} & + & 2x_{21} & + & x_{31} & + & & + & x_{22} & + & x_{32} \\
\text{subject to} & x_{11} & & & & & & x_{12} & & & & & \geq & 1 & (m) \\
& 5x_{11} & + & 7x_{21} & + & 4x_{31} & & & & & & & \leq & 11 & (s1) \\
& & & & & & & x_{12} & + & 2x_{22} & + & x_{32} & \leq & 2 & (s2) \\
& x_{ij} \in \{0,1\} & i \in \{1,\ldots,3\}, & j \in \{1,\ldots,2\}
\end{array}
$$

It is obvious from the structure of the problem that if constraint (m) is removed, then the remaining constraints (s1) and (s2) decompose into two independent subproblems. The next two sections describe how to solve this MILP by using the decomposition algorithm in the OPTMODEL procedure and OPTMILP procedure, respectively.

Solving a MILP with DECOMP and PROC OPTMODEL

The following statements use the OPTMODEL procedure and the decomposition algorithm to solve the MILP:

```
proc optmodel;
   var x{i in 1..3, j in 1..2} binary;

   max f =    x[1,1] + 2*x[2,1] +   x[3,1]
                    +   x[2,2] +   x[3,2];

   con m :    x[1,1] +   x[1,2]               >=   1;
   con s1:  5*x[1,1] + 7*x[2,1] + 4*x[3,1] <= 11;
   con s2:    x[1,2] + 2*x[2,2] +   x[3,2] <=   2;

   s1.block = 0;
   s2.block = 1;

   solve with milp / presolver=none decomp=(logfreq=1);
   print x;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. The solution summary and optimal solution are displayed in Figure 13.1.

Figure 13.1 Solution Summary and Optimal Solution

```
                      The OPTMODEL Procedure

                         Solution Summary

             Solver                           MILP
             Algorithm              Decomposition
             Objective Function                 f
             Solution Status             Optimal
             Objective Value                   4

             Relative Gap                      0
             Absolute Gap                      0
             Primal Infeasibility             0
             Bound Infeasibility              0
             Integer Infeasibility            0

             Best Bound                        4
             Nodes                             1
             Iterations                        4
             Presolve Time                  0.01
             Solution Time                  0.36
```

Figure 13.1 *continued*

```
                          x
                          1    2

                    1     0    1
                    2     1    0
                    3     1    1
```

The iteration log, which displays the problem statistics, the progress of the solution, and the optimal objective value, is shown in Figure 13.2.

Figure 13.2 Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 6 binary and 0 integer variables.
NOTE: The problem has 3 linear constraints (2 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 8 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 2 disjoint blocks.
NOTE: The decomposition subproblems cover 6 (100.00%) variables and 2 (66.67%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best        Master         Best        LP       IP   CPU   Real
                   Bound     Objective      Integer       Gap      Gap  Time   Time
NOTE: Starting phase 1.
         1        0.0000        1.0000           .  1.00e+00         .    0      0
         2        0.0000        0.0000           .     0.00%         .    0      0
NOTE: Starting phase 2.
         .        4.0000        3.0000      3.0000    25.00%    25.00%    0      0
         3        4.0000        3.0000      3.0000    25.00%    25.00%    0      0
         4        4.0000        4.0000      4.0000     0.00%     0.00%    0      0
       Node   Active   Sols         Best         Best       Gap    CPU    Real
                                 Integer        Bound             Time    Time
          0        0      2      4.0000       4.0000     0.00%       0       0
NOTE: The Decomposition algorithm used 2 threads.
NOTE: The Decomposition algorithm time is 0.02 seconds.
NOTE: Optimal.
NOTE: Objective = 4.
```

Solving a MILP with DECOMP and PROC OPTMILP

Alternatively, to solve the MILP with the OPTMILP procedure, create a corresponding SAS data set that uses the mathematical programming system (MPS) format as follows:

```
data mpsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME      .         mpsdata  .        .         .
ROWS      .         .        .        .         .
MAX       f         .        .        .         .
G         m         .        .        .         .
L         s1        .        .        .         .
L         s2        .        .        .         .
COLUMNS   .         .        .        .         .
   .      .MRK0000  'MARKER' .        'INTORG'  .
   .      x[1,1]    f        1        m         1
   .      x[1,1]    s1       5        .         .
   .      x[2,1]    f        2        s1        7
   .      x[3,1]    f        1        s1        4
   .      x[1,2]    m        1        s2        1
   .      x[2,2]    f        1        s2        2
   .      x[3,2]    f        1        s2        1
   .      .MRK0001  'MARKER' .        'INTEND'  .
RHS       .         .        .        .         .
   .      .RHS.     m        1        .         .
   .      .RHS.     s1       11       .         .
   .      .RHS.     s2       2        .         .
BOUNDS    .         .        .        .         .
UP        .BOUNDS.  x[1,1]   1        .         .
UP        .BOUNDS.  x[2,1]   1        .         .
UP        .BOUNDS.  x[3,1]   1        .         .
UP        .BOUNDS.  x[1,2]   1        .         .
UP        .BOUNDS.  x[2,2]   1        .         .
UP        .BOUNDS.  x[3,2]   1        .         .
ENDATA    .         .        .        .         .
;
```

Next, use the following SAS data set to define the subproblem blocks:

```
data blocks;
   input _row_ $ _block_;
   datalines;
s1 0
s2 1
;
```

Now, you can use the following OPTMILP statements to solve this MILP:

```
proc optmilp
   data      = mpsdata
   presolver = none;
   decomp
      logfreq = 1
```

```
        blocks  = blocks;
    run;
```

Syntax: Decomposition Algorithm

You can specify the decomposition algorithm either by using options in a SOLVE statement in the OPT-MODEL procedure or by using statements in the OPTLP and OPTMILP procedures. Except for the fact that you use SOLVE statement options in PROC OPTMODEL or you use statements in PROC OPTLP and PROC OPTMILP, the syntax is identical.

The following decomposition algorithm options are available in the SOLVE statement in the OPTMODEL procedure:

> **SOLVE WITH LP** / *< options >*
> > *<* **DECOMP=(** *< decomp-options >* **)** *>*
> > *<* **DECOMP_MASTER=(** *< decomp-master-options >* **)** *>*
> > *<* **DECOMP_SUBPROB=(** *< decomp-subprob-options >* **)** *>* **;**
>
> **SOLVE WITH MILP** / *< options >*
> > *<* **DECOMP=(** *< decomp-options >* **)** *>*
> > *<* **DECOMP_MASTER=(** *< decomp-master-options >* **)** *>*
> > *<* **DECOMP_MASTER_IP=(** *< decomp-master-ip-options >* **)** *>*
> > *<* **DECOMP_SUBPROB=(** *< decomp-subprob-options >* **)** *>* **;**

The following statements are available in the OPTLP procedure:

> **PROC OPTLP** *< options >* **;**
> > **DECOMP** *< decomp-options >* **;**
> > **DECOMP_MASTER** *< decomp-master-options >* **;**
> > **DECOMP_SUBPROB** *< decomp-subprob-options >* **;**

The following statements are available in the OPTMILP procedure:

> **PROC OPTMILP** *< options >* **;**
> > **DECOMP** *< decomp-options >* **;**
> > **DECOMP_MASTER** *< decomp-master-options >* **;**
> > **DECOMP_MASTER_IP** *< decomp-master-ip-options >* **;**
> > **DECOMP_SUBPROB** *< decomp-subprob-options >* **;**

Decomposition Algorithm Options in the PROC OPTLP Statement or the SOLVE WITH LP Statement in PROC OPTMODEL

To solve a linear program, you can specify the decomposition algorithm in a SOLVE WITH LP statement in the OPTMODEL procedure or in a PROC OPTLP statement in the OPTLP procedure. To control the overall decomposition algorithm, you can specify one or more of the LP solver options shown in Table 13.1. (As indicated, you can specify some options only in the PROC OPTLP statement.)

The options in Table 13.1 control the overall process flow for solving a linear program, and they are equivalent to the options that are used in PROC OPTLP and PROC OPTMODEL with standard methods. These options

are called main solver options in this chapter. They are described in detail in the section "Syntax: LP Solver" on page 185 and the section "Syntax: OPTLP Procedure" on page 365.

Table 13.1 Options in the PROC OPTLP Statement or SOLVE WITH LP Statement

Description	option
Data Set Options (OPTLP procedure only)	
Specifies the input data set	DATA=
Specifies the dual solution output data set	DUALOUT=
Specifies whether the model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution output data set	PRIMALOUT=
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies how frequently to print the solution progress	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=
Specifies whether time units are CPU time or real time	TIMETYPE=
Algorithm Options	
Enables or disables scaling of the problem	SCALE=

Decomposition Algorithm Options in the PROC OPTMILP Statement or the SOLVE WITH MILP Statement in PROC OPTMODEL

To solve a mixed integer linear program, you can specify the decomposition algorithm in a SOLVE WITH MILP statement in the OPTMODEL procedure or in a PROC OPTMILP statement in the OPTMILP procedure. To control the overall decomposition algorithm, you can specify one or more of the MILP solver options shown in Table 13.2. (As indicated, you can specify some options only in the PROC OPTMILP statement.)

The options in Table 13.2 control the overall process flow for solving a mixed integer linear program, and they are equivalent to the options used in the OPTMILP and OPTMODEL procedures with standard methods. These options are called main solver options in this chapter. They are described in detail in the section "Syntax: MILP Solver" on page 249 and the section "Syntax: OPTMILP Procedure" on page 421.

Table 13.2 Options in the PROC OPTMILP Statement or SOLVE WITH MILP Statement

Description	option
Data Set Options (OPTMILP procedure only)	
Specifies the input data set	DATA=
Specifies the constraint activities output data set	DUALOUT=
Specifies whether the model is a maximization or minimization problem	OBJSENSE=

Description	*option*
Specifies the primal solution input data set (warm start)	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=
Specifies the maximum violation of variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies how frequently to print the node log	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used when deciding on the optimality of nodes in the branch-and-bound tree	OPTTOL=
Uses the input primal solution (warm start) (OPTMODEL procedure only)	PRIMALIN
Enables or disables printing summary (OPTMILP procedure only)	PRINTLEVEL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on a relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=

DECOMP Statement

DECOMP < *decomp-options* > **;**

The DECOMP statement controls the overall decomposition algorithm.

Table 13.3 summarizes the *decomp-options* available in the DECOMP statement. These options control the overall decomposition algorithm process flow during the solution of an LP or a MILP. (As indicated, you can specify the data set options only in the OPTLP or OPTMILP procedure, and you can specify some control options only for a MILP.)

Table 13.3 Options in the DECOMP Statement

Description	*decomp-option*
Data Set Options (OPTLP and OPTMILP procedures only)	
Specifies the blocks input data set	BLOCKS=
Control Options	
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=
Specifies the frequency of removing ineffective columns from the master LP	COMPRESSFREQ=

Table 13.3 (continued)

Description	*decomp-option*
Specifies whether to initialize the columns by solving each block with the original cost vector	INITVARS=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of blocks to allow	MAXBLOCKS=
Specifies the maximum number of new columns to allow into the master each pass	MAXCOLSPASS=
Specifies the maximum amount of time spent in the decomposition algorithm	MAXTIME=
Specifies the decomposition algorithm method	METHOD=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Control Options (MILP only)	
Specifies how frequently to print the continuous iteration log	LOGFREQ=
Specifies whether the master problem is solved as a MILP with the current set of columns at the beginning of phase II	MASTER_IP_BEG=
Specifies whether the master problem is solved as a MILP with the current set of columns at the end of phase II	MASTER_IP_END=
Specifies the frequency of solving the master problem as a MILP with the current set of columns	MASTER_IP_FREQ=
Specifies the maximum number of outer iterations for the decomposition algorithm	MAXITER=

The following list describes the *decomp-options* in detail.

ABSOBJGAP=*number*

> specifies a stopping criterion for the continuous bound of the decomposition. When the absolute difference between the master objective and the best dual bound falls below the value of *number*, the decomposition algorithm stops adding columns. The value of *number* can be any nonnegative number. The default value is the value of the OPTTOL= main solver option.

BLOCKS=*SAS-data-set*

> specifies (for OPTLP and OPTMILP procedures only) the input data set that contains block definitions to be used by the decomposition algorithm if METHOD=USER. See the section "The BLOCKS= Data Set in PROC OPTMILP and PROC OPTLP" on page 525 for more information. To specify blocks in PROC OPTMODEL, use the `.block` constraint suffix instead (see the section "The `.block` Constraint Suffix in PROC OPTMODEL" on page 525).

COMPRESSFREQ=*number*

> removes ineffective columns from the master LP after every *number* of iterations. The frequency, *number*, is an integer between 0 and the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 0.

INITVARS=*number | string*

> specifies whether to initialize the columns by using the original cost vector to solve each block.
>
> Table 13.4 describes the valid values of the INITVARS= option.

Table 13.4 Values for INITVARS= Option

number	string	Description
0	OFF	Disables initializing the columns by using the original cost vector to solve each block.
1	ON	Enables initializing the columns by using the original cost vector to solve each block.

The default is ON.

LOGFREQ=number

specifies (for MILP problems only) how often to print information in the continuous iteration log. The value of *number* can be any nonnegative number up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is 10. If *number* is set to 0, then the iteration log is disabled. If *number* is positive, then an entry is made in the log at the first iteration, at the last iteration, and at intervals that are dictated by the value of *number*. An entry is also made each time a better integer solution or improved bound is found.

LOGLEVEL=number | string

controls the amount of information that is displayed in the SAS log by the decomposition algorithm. Table 13.5 and Table 13.6 provide the valid values for this option and a description of what is displayed in the log when an LP and a MILP, respectively, is solved.

Table 13.5 Values for LOGLEVEL= Option for an LP

number	string	Description
−1	AUTOMATIC	Prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.
0	NONE	Turns off printing of all of the decomposition algorithm messages to the SAS log.
1	BASIC	Prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.
2	MODERATE	Prints the continuous iteration log and summary information for each iteration at the interval dictated by the LOGFREQ= main solver option.
3	AGGRESSIVE	Prints the continuous iteration log and detailed information for each iteration at the interval dictated by the LOGFREQ= main solver option.

Table 13.6 Values for LOGLEVEL= Option for a MILP

number	string	Description
−1	AUTOMATIC	Prints the continuous iteration log for the root node at the interval dictated by the LOGFREQ= option in the DECOMP statement. Prints the branch-and-bound node log at the interval dictated by the LOGFREQ= main solver option.
0	NONE	Turns off printing of all of the decomposition algorithm messages to the SAS log.

Table 13.6 (continued)

number	string	Description
1	BASIC	Prints the continuous iteration log for each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.
2	MODERATE	Prints the continuous iteration log and summary information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.
3	AGGRESSIVE	Prints the continuous iteration log and detailed information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.

The default is AUTOMATIC for both LPs and MILPs.

MASTER_IP_BEG=_number_ | _string_

specifies (for MILP problems only) whether the master problem is solved as a MILP with the current set of columns at the beginning of phase II. Table 13.7 describes the valid values of the MASTER_IP_BEG= option.

Table 13.7 Values for MASTER_IP_BEG= Option

number	string	Description
0	OFF	Disables solving the master as a MILP at the beginning of phase II.
1	ON	Enables solving the master as a MILP at the beginning of phase II.

The default is ON in the root node and 0 elsewhere.

MASTER_IP_END=_number_ | _string_

specifies (for MILP problems only) whether the master problem is solved as a MILP with the current set of columns at the end of phase II. Table 13.8 describes the valid values of the MASTER_IP_END= option.

Table 13.8 Values for MASTER_IP_END= Option

number	string	Description
0	OFF	Disables solving the master as a MILP at the end of phase II.
1	ON	Enables solving the master as a MILP at the end of phase II.

The default is ON in the root node and 0 elsewhere.

MASTER_IP_FREQ=_number_

solves the master problem (for MILP problems only) as a MILP with the current set of columns after every _number_ iterations. The frequency, _number_, is an integer between 0 and the largest four-byte signed integer, which is $2^{31} - 1$. The default is 10 in the root node and 0 elsewhere.

MAXBLOCKS=*number*

specifies the maximum number of blocks to allow. If the defined number of blocks exceeds *number*, the algorithm creates superblocks using a very simple round-robin scheme. The value *number* can be any integer between 1 and the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXCOLSPASS=*number*

specifies the maximum number of new columns to allow into the master at each pass. This option is disabled on the initial pass if INITVARS=1. The value *number* can be any integer between 1 and the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXITER=*number*

specifies (for MILP problems only) the maximum number of outer iterations for the decomposition algorithm. The value *number* can be any integer between 1 and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=*number*

specifies an upper limit of *number* seconds of time for the decomposition algorithm. The value of the TIMETYPE= main solver option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

METHOD=*string*

specifies the decomposition algorithm method as shown in Table 13.9.

Table 13.9 Values for METHOD= Option

string	Description
USER	The user defines which rows belong to which blocks (subproblems). In PROC OPTMODEL, use the **.block** constraint suffix. In PROC OPTLP and PROC OPTMILP, use the BLOCKS= data set instead.
NETWORK	The algorithm attempts to find an embedded network similar to what is described in "The Network Simplex Algorithm" on page 192. The weakly connected components of this network are used as the blocks.
AUTO	The algorithm attempts to find a block structure in the constraint matrix. For the current release, METHOD=AUTO finds block-diagonal structure only (not block-angular structures); unless your problem separates into completely independent problems with no linking constraints, this method finds only one block and hence is equivalent to calling the MILP solver directly.

The default is USER if blocks are defined and NETWORK otherwise.

RELOBJGAP=_number_

specifies the relative objective gap as a stopping criterion. The relative objective gap is based on the master objective (MasterObjective) and the best dual bound (BestBound); it is equal to

$$| \text{MasterObjective} - \text{BestBound} | / (1\text{E}{-}10 + | \text{BestBound} |)$$

When this value becomes smaller than the specified gap size *number*, the decomposition algorithm stops adding columns. The value of *number* can be any nonnegative number. For LP, the default value is 0; for MILP, the default value is 1e-4.

DECOMP_MASTER Statement

> **DECOMP_MASTER** < *decomp-master-options* > ;

> **MASTER** < *decomp-master-options* > ;

The DECOMP_MASTER statement controls the master problem.

Table 13.10 summarizes the options available in the DECOMP_MASTER statement. These options control the master LP solver in the decomposition algorithm during the solution of an LP or a MILP. (As indicated, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section "LP Solver Options" on page 186 and the section "PROC OPTLP Statement" on page 366.

The following default values differ from the LP solver defaults: ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART. These different defaults are motivated by the fact that primal feasibility of the master problem is preserved when columns are added, so a warm start from the previous optimal basis tends to be more efficient than solving the master from scratch in each iteration.

Table 13.10   Options in the DECOMP_MASTER Statement

Description	decomp-master-option
Algorithm Option	
Specifies the master algorithm	ALGORITHM=
Presolve Option	
Specifies, for the first master solve only, the type of presolve	INITPRESOLVER=
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies how frequently to print the solution progress	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=
Specifies whether time units are CPU time or real time	TIMETYPE=
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=
Specifies the queue size for determining the entering variable	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Interior Point Algorithm Options	

Table 13.10 (continued)

Description	*decomp-master-option*
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on a duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=

In addition to the options listed in Table 13.10, you can specify the following *decomp-master-option* in the DECOMP_MASTER statement.

INITPRESOLVER=number | string

INITPRESOL=number | string

      specifies, for the first master solve only, presolve conditions as shown in Table 13.11.

Table 13.11 Values for INITPRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of presolve processing.
0	NONE	Disables presolver.
1	BASIC	Performs minimal presolve processing.
2	MODERATE	Applies a higher level of presolve processing.
3	AGGRESSIVE	Applies the highest level of presolve processing.

      The default is AUTOMATIC.

DECOMP_MASTER_IP Statement

      DECOMP_MASTER_IP < *decomp-master-ip-options* > **;**

      MASTER_IP < *decomp-master-ip-options* > **;**

For mixed integer linear programming problems, the DECOMP_MASTER_IP statement controls the (restricted) master problem, which is solved as a MILP with the current set of columns in an effort to obtain an integer-feasible solution.

Table 13.12 summarizes the options available in the DECOMP_MASTER_IP statement. These options control the MILP solver that is used to solve the integer version of the master problem. For descriptions of these options, see the section "MILP Solver Options" on page 251 and the section "PROC OPTMILP Statement" on page 423.

Table 13.12 Options in the DECOMP_MASTER_IP Statement

Description	*decomp-master-ip-option*
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=

Table 13.12 (continued)

Description	*decomp-master-ip-option*
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies how frequently to print the node log	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used when deciding on the optimality of nodes in the branch-and-bound tree	OPTTOL=
Specifies whether to use the previous best primal solution as a warm start	PRIMALIN=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on a relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the stopping criterion based on the target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=

In addition to the *decomp-master-ip-options* specified in Table 13.12, you can specify the following *decomp-*

master-ip-option in the DECOMP_MASTER_IP statement.

PRIMALIN=*number* | *string*

PIN=*number* | *string*

> specifies whether the MILP solver is to use the previous best solution's variables values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. When it is difficult to find a good integer-feasible solution for a problem, warm start can reduce solution time significantly. Table 13.13 describes the valid values of the PRIMALIN= option.

Table 13.13 Values for PRIMALIN= Option

number	*string*	**Description**
0	OFF	Ignores the previous solution.
1	ON	Starts from the previous solution.

> The default is ON.

DECOMP_SUBPROB Statement

> **DECOMP_SUBPROB** < *decomp-subprob-options* > **;**
>
> **SUBPROB** < *decomp-subprob-options* > **;**

The DECOMP_SUBPROB statement controls the subproblem.

Table 13.14 summarizes the options available for the decomposition algorithm in the DECOMP_SUBPROB statement when the subproblem algorithm chosen is an LP algorithm. (As indicated, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section "LP Solver Options" on page 186 and the section "PROC OPTLP Statement" on page 366.

The following default values differ from the LP solver defaults: ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART (when METHOD=USER is specified in the DECOMP statement), and AL-GORITHM=NETWORK_PURE (when METHOD=NETWORK is specified in the DECOMP statement). For METHOD=USER, these defaults are motivated by the fact that primal feasibility of the subproblem is preserved when the objective is changed, so a warm start from the previous optimal basis tends to be more efficient than solving the subproblem from scratch in each iteration. For METHOD=NETWORK, the specialized pure network solver is usually the most efficient choice because each subproblem is a pure network.

Table 13.14 Options in the DECOMP_SUBPROB Statement Used with an LP Algorithm

Description	*decomp-subprob-option*
Algorithm Option	
Specifies the subproblem algorithm	ALGORITHM=
Presolve Option	
Specifies, for the first master solve only, the type of presolve	INITPRESOLVER=
Specifies the type of presolve	PRESOLVER=

Table 13.14 (continued)

Description	decomp-subprob-option
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies how frequently to print the solution progress	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=
Specifies whether time units are CPU time or real time	TIMETYPE=
Simplex Algorithm Options	
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=
Specifies the queue size for determining entering variable	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Interior Point Algorithm Options	
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=

Table 13.15 summarizes the options available in the DECOMP_SUBPROB statement when the subproblem algorithm chosen is a MILP algorithm. When the subproblem consists of multiple blocks (a block-diagonal structure), these settings apply to all subproblems. For descriptions of these options, see the section "MILP Solver Options" on page 251 and the section "PROC OPTMILP Statement" on page 423.

Table 13.15 Options in the DECOMP_SUBPROB Statement Used with a MILP Algorithm

Description	Option
Algorithm Option	
Specifies the subproblem algorithm	ALGORITHM=
Presolve Option	
Specifies, for the first subproblem solve only, the type of presolve	INITPRESOLVER=
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies how frequently to print the node log	LOGFREQ=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=

Description	Option
Specifies the tolerance used when deciding on the optimality of nodes in the branch-and-bound tree	OPTTOL=
Specifies whether to use the previous best primal solution as a warm start	PRIMALIN=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=

In addition to the *decomp-subprob-options* specified in Table 13.14 and Table 13.15, you can specify the following *decomp-subprob-options* in the DECOMP_SUBPROB statement.

ALGORITHM=*string*

SOLVER=*string*

SOL=*string*

> specifies one of the algorithms shown in Table 13.16 (the valid abbreviated value for each *string* is shown in parentheses).

Table 13.16   Values for ALGORITHM= Option

string	Description
PRIMAL (PS)	Uses the primal simplex algorithm.

string	Description
DUAL (DS)	Uses the dual simplex algorithm.
NETWORK (NS)	Uses the network simplex algorithm.
NETWORK_PURE (NSPURE)	Uses the network simplex algorithm for pure networks.
INTERIORPOINT (IP)	Uses the interior point algorithm.
MILP	Uses the mixed integer linear solver.

For mixed integer programming problems, the default is NETWORK_PURE if METHOD=NETWORK and MILP otherwise. For linear programming problems, the default is NETWORK_PURE if METHOD=NETWORK, DUAL if METHOD=AUTO, and PS otherwise.

INITPRESOLVER=number | string

INITPRESOL=number | string

specifies, for the first subproblem solve only, presolve conditions as listed in Table 13.17.

Table 13.17 Values for INITPRESOLVER= Option

number	string	Description
−1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default is AUTOMATIC.

PRIMALIN=number | string

PIN=number | string

specifies (for MILP problems only) whether the MILP solver is to use the values of the previous best solution's variables as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. When it is difficult to find a good integer-feasible solution for a problem, warm start can reduce solution time significantly. Table 13.18 describes the valid values of the PRIMALIN= option.

Table 13.18 Values for PRIMALIN= Option

number	string	Description
0	OFF	Ignores the previous solution.
1	ON	Starts from the previous solution.

The default is ON.

Details: Decomposition Algorithm

Data Input

This subsection describes the format for describing the partition of the constraint system that defines the subproblem blocks. In the OPTLP and OPTMILP procedures, partitioning is done by using a data set specified in the BLOCKS= data option in the DECOMP statement. In PROC OPTMODEL, partitioning is done by using the `.block` suffix on constraints.

The blocks must be disjoint with respect to variables. If two blocks contain a nonzero coefficient for the same variable, the decomposition algorithm produces an error that contains information about where the blocks overlap.

The BLOCKS= Data Set in PROC OPTMILP and PROC OPTLP

The BLOCKS= data set has two required variables:

ROW
    specifies the constraint (row) names of the problem. The values should be a subset of the row names in the DATA= data set for the current problem.

BLOCK
    specifies the numeric block identifier for each constraint in the problem. A missing observation or missing value indicates a master (linking) constraint that does not appear in any block. Listing the linking constraints is optional. The block identifiers must start from 0 and be consecutive.

See the section "Solving a MILP with DECOMP and PROC OPTMILP" on page 510 for an example of using this BLOCKS= data set with PROC OPTMILP.

The `.block` Constraint Suffix in PROC OPTMODEL

The `.block` constraint suffix specifies the numeric block identifier for each constraint in the problem. The block identifiers do not need to start from 0, nor do they need to be consecutive. Master (linking) constraints can be identified by using a missing value. Listing the linking constraints is optional.

See the section "Solving a MILP with DECOMP and PROC OPTMODEL" on page 508 for an example of using the `.block` constraint suffix with PROC OPTMODEL.

Decomposition Algorithm

The decomposition algorithm for LPs is based on the original Dantzig-Wolfe method (Dantzig and Wolfe 1960). Embedding this method in the context of a branch-and-bound algorithm for MILPs is described in Barnhart, Hane, and Vance (2000) and is often referred to as *branch-and-price*. The design of a framework that allows for building a generic branch-and-price solver that requires only the original (compact) formulation and the constraint partition was first proposed independently by Ralphs and Galati (2006) and Vanderbeck and Savelsbergh (2006). This method is also commonly referred to as *column generation*, although the algorithm implemented here is only one specific variant of this wider class of algorithms.

The algorithm setup starts by forming various components that are used iteratively during the solver process. These components include the master problem (controlled by options in the DECOMP_MASTER statement), one subproblem for each block (controlled by options in the DECOMP_SUBPROB statement) and, for MILPs, the integer version of the master problem (controlled by options in the DECOMP_MASTER_IP statement).

The master problem is a linear program that is defined over a potentially large number of variables that represent the weights of a convex combination. The points in the convex combination satisfy the constraints that are defined in the subproblem. The master constraints of the original problem are enforced in this reformulated space. In this sense, the decomposition algorithm takes the intersection of two polyhedra: one defined by original master constraints and one defined by the subproblem constraints. Since the set of variables needed to define the intersection of the polyhedra can be large, the algorithm works on a restricted subset and generates only those variables (columns) that have good potential with respect to feasibility and optimality. This generation is done by using the dual information that is obtained by solving the master problem to *price out* new variables. These new variables are generated by solving the subproblems over the appropriate cost vector (the reduced cost in the original space). This generation is similar to the revised simplex method, except that the variable space is exponentially large and therefore is generated implicitly by solving an optimization problem. This idea of generating variables as needed is the reason why this method is often referred to as *column generation*.

Similar to the two-phase simplex algorithm, the algorithm first introduces slack variables and solves a phase I problem to find a feasible solution. After the algorithm finds a feasible solution, it switches to a phase II problem to search for an optimal solution. The process of solving the master to generate pricing information and then solving one or more subproblems to generate candidate variables is repeated until there are no longer any improving variables and the method has converged.

For MILPs, this process is then used as a bounding method in a branch-and-bound algorithm, as described in the section "Branch-and-Bound Algorithm" on page 435. The strength of the subproblem polyhedron is one of the key reasons why decomposition can often solve problems that the standard branch-and-cut algorithm cannot solve in a reasonable amount of time. Since the points used in the convex combination are solutions (extreme points) of the subproblem (typically a MILP itself), then the bounds obtained can often be much stronger than the bounds obtained from standard branch-and-bound methods that are based on the LP relaxation. The subproblem polyhedron intersected with the continuous master polyhedron can provide a very good approximation of the true convex hull of the original integer program.

For more information about the algorithm process flow and the framework design, see Galati (2009).

Parallel Execution

At each iteration of the decomposition method, the subproblem is solved over the reduced cost that is derived from the dual information that solving the master problem provides. As discussed in the section "Overview: Decomposition Algorithm" on page 505, the subproblem often has a block-diagonal structure that enables the solver to process each block independently.

The decomposition algorithm can be run in either a single-machine or a distributed computing environment. In single-machine mode, the computation is executed by multiple threads on a single computer. You can specify options for parallel execution in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 28 of Chapter 4, "Shared Concepts and Topics." You can control the number of threads that are used by specifying the NTHREADS= option in the PERFORMANCE statement.

In distributed mode, the computation is executed in a distributed computing environment. You can control the number of grid nodes (machines) that are used by specifying the NODES= option in the PERFORMANCE statement. The decomposition algorithm supports only the deterministic mode of the PARALLELMODE= option in the PERFORMANCE statement. The default mode of operation is single-machine mode, where the number of concurrent threads is based on the number of CPUs (cores) on the machine (subject to any system configuration limitations).

NOTE: The SAS High-Performance Optimization license is required to invoke the DECOMP option in distributed mode.

Log for the Decomposition Algorithm

The following subsections describe what to expect in the SAS log when you run the decomposition algorithm.

Setup Information in the SAS Log

In the setup phase of the algorithm, information about the method you choose and the structure of the model is written to the SAS log. One of the most important pieces of information displayed in the log is the number of disjoint blocks and the coverage of those blocks with respect to both variables and constraints in the original presolved model. As explained in the section "Overview: Decomposition Algorithm" on page 505, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. However, this is not always a straightforward indicator for MILPs, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system.

After the structural information is written to the log, the algorithm begins and the iteration log is displayed.

Iteration Log for LPs

When the decomposition algorithm solves LPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space.

The following information is written to the iteration log:

Iter	indicates the iteration number.
Best Bound	indicates the best dual bound found so far.
Master Objective	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Gap	indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm is to convergence. If the best bound is 0, or if the relative gap is greater than 1000%, then the absolute gap is written.
CPU Time	indicates the CPU time elapsed (in seconds).
Real Time	indicates the real time elapsed (in seconds).

Entries are made in the log at a frequency that is specified in the LOGFREQ= option. If LOGFREQ=0, then the iteration log is disabled. If the LOGFREQ= value is positive, then an entry is made in the log at the first

iteration, at the last iteration, and at intervals that are specified by the LOGFREQ= value. An entry is also made each time an improved bound is found.

The behavior of objective values in the iteration log depends on both the current phase and on which solver you choose. In phase I, the master formulation has an artificial objective value that decreases to 0 when a feasible solution is found. In phase II, the decomposition algorithm maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

When you specify LOGLEVEL=MODERATE or LOGLEVEL=AGGRESSIVE in the DECOMP statement, information about the subproblem solves is written before each iteration line.

Iteration Log for MILPs

When the decomposition algorithm solves MILPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space, in addition to the global convergence of the branch-and-bound algorithm for finding an optimal integer solution.

You can control the amount of information at each node by using the LOGLEVEL= option in the DECOMP statement. By default, the continuous iteration log for the root node is written at the interval specified in the LOGFREQ= option in the DECOMP statement. Then the branch-and-bound node log is written at the interval specified in the LOGFREQ= main solver option.

When the algorithm solves MILPs, the continuous iteration log is similar to the iteration log described in the section "Iteration Log for LPs" on page 527 except that information about integer-feasible solutions is also displayed. The following information is printed in the continuous iteration log when the algorithm solves MILPs:

Iter	indicates the iteration number.
Best Bound	indicates the best dual bound found so far.
Master Objective	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Best Integer	indicates the objective of the best integer-feasible solution found so far.
LP Gap	indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm for this particular node is to convergence. If the best bound is 0, or if the relative gap is greater than 1000%, then the absolute gap is displayed.
IP Gap	indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the best bound is 0, or if the relative gap is greater than 1000%, then the absolute gap is displayed.
CPU Time	indicates the CPU time elapsed (in seconds).
Real Time	indicates the real time elapsed (in seconds).

After the root node is complete, the algorithm then moves into the branch-and-bound phase. By default, it displays the branch-and-bound node log and suppresses the continuous iteration log. The following information is printed in the branch-and-bound node log when the algorithm solves MILPs:

Node   indicates the sequence number of the current node in the search tree.

Active   indicates the current number of active nodes in the branch-and-bound tree.

Sols   indicates the number of feasible solutions found so far.

Best Integer   indicates the objective of the best integer-feasible solution found so far.

Best Bound   indicates the best dual bound found so far.

Gap   indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the best bound is 0, or if the relative gap is greater than 1000%, then the absolute gap is displayed.

CPU Time   indicates the CPU time elapsed (in seconds).

Real Time   indicates the real time elapsed (in seconds).

If the LOGLEVEL= option in the DECOMP statement is set to BASIC, MODERATE or AGGRESSIVE, then the continuous iteration log is displayed for each branch-and-bound node at the interval specified in the LOGFREQ= option in the DECOMP statement.

Additional information can be displayed to the log by specifying the LOGLEVEL= option in each of the algorithmic component statements (DECOMP_MASTER, DECOMP_MASTER_IP, and DECOMP_SUBPROB). By default, the individual component log levels are all disabled.

Examples: Decomposition Algorithm

Example 13.1: Multicommodity Flow Problem

This example demonstrates how to use the decomposition algorithm to find a minimum-cost multicommodity flow (MMCF) in a directed network. This type of problem was motivation for the development of the original Dantzig-Wolfe decomposition method (Dantzig and Wolfe 1960).

Let $G = (N, A)$ be a directed graph, and let K be a set of commodities. For each link $(i, j) \in A$ and each commodity k, associate a cost per unit of flow, designated by c_{ij}^k. The demand (or supply) at each node $i \in N$ for commodity k is designated as b_i^k, where $b_i^k \geq 0$ denotes a supply node and $b_i^k < 0$ denotes a demand node. Define decision variables x_{ij}^k that denote the amount of commodity k sent from node i and node j. The amount of total flow, across all commodities, that can be sent across each link is bounded above by u_{ij}.

The problem can be modeled as a linear programming problem as follows:

minimize $\quad\displaystyle\sum_{k\in K}\sum_{(i,j)\in A} c_{ij}^k x_{ij}^k$

subject to

$$\sum_{k\in K} x_{ij}^k \le u_{ij} \qquad (i,j)\in A \qquad\qquad \text{(capacity)}$$

$$\sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k = b_i^k \qquad i\in N,\ k\in K \qquad\qquad \text{(balance)}$$

$$x_{ij}^k \ge 0 \qquad (i,j)\in A,\ k\in K$$

In this formulation, constraints (capacity) limit the total flow across all commodities on each arc. The constraints (balance) ensure that the flow of commodities leaving each supply node and entering each demand node are balanced.

Consider the directed graph in Figure 13.3 which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 13.3 Example Network with Two Commodities
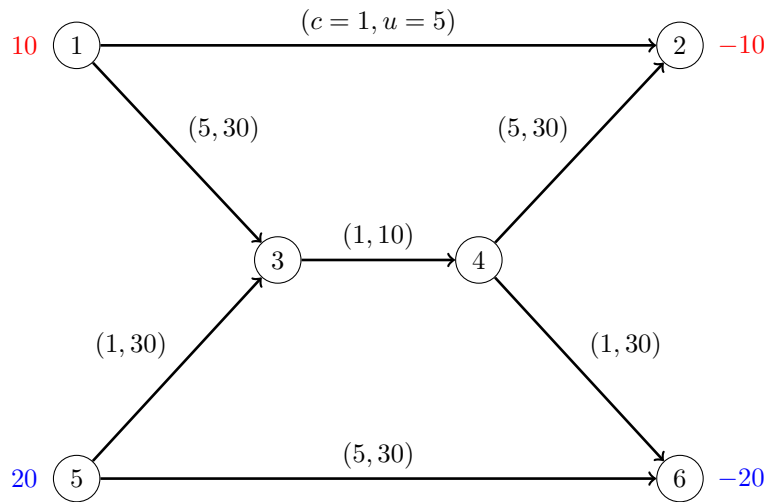
The goal in this example is to minimize the total cost of sending two commodities across the network while satisfying all supplies and demands and respecting arc capacities. If there were no arc capacities linking the two commodities, you could solve a separate minimum-cost network flow problem for each commodity one at a time.

The following data set arc_comm_data provides the cost c_{ij}^k of sending a unit of commodity k along arc (i,j):

```
data arc_comm_data;
   input k i j cost;
   datalines;
1 1 2 1
1 1 3 5
1 5 3 1
1 5 6 5
1 3 4 1
```

```
1 4 2 5
1 4 6 1
2 1 2 1
2 1 3 5
2 5 3 1
2 5 6 5
2 3 4 1
2 4 2 5
2 4 6 1
;
```

Next, the data set arc_data provides the capacity u_{ij} for each arc:

```
data arc_data;
   input i j capacity;
   datalines;
1 2   5
1 3 30
5 3 30
5 6 30
3 4 10
4 2 30
4 6 30
;
```

Lastly, the data set supply_data provides the nonzero supply (or demand) b_i^k for each node and each commodity:

```
data supply_data;
   input k i supply;
   datalines;
1 1   10
1 2 -10
2 5   20
2 6 -20
;
```

The following PROC OPTMODEL statements find the minimum-cost multicommodity flow:

```
proc optmodel;
   set <num,num,num> ARC_COMM;
   num cost {ARC_COMM};
   read data arc_comm_data into ARC_COMM=[i j k] cost;

   set ARCS = setof {<i,j,k> in ARC_COMM} <i,j>;
   set COMMODITIES = setof {<i,j,k> in ARC_COMM} k;
   set NODES = union {<i,j> in ARCS} {i,j};

   num capacity {ARCS};
   read data arc_data into [i j] capacity;

   num supply {NODES, COMMODITIES} init 0;
   read data supply_data into [i k] supply;

   var Flow {<i,j,k> in ARC_COMM} >= 0;
```

```
min TotalCost =
    sum {<i,j,k> in ARC_COMM} cost[i,j,k] * Flow[i,j,k];
con BalanceCon {i in NODES, k in COMMODITIES}:
    sum {<(i),j,(k)> in ARC_COMM} Flow[i,j,k]
     - sum {<j,(i),(k)> in ARC_COMM} Flow[j,i,k] = supply[i,k];
con CapacityCon {<i,j> in ARCS}:
    sum {<(i),(j),k> in ARC_COMM} Flow[i,j,k] <= capacity[i,j];
```

Because each (balance) constraint involves variables for only one commodity, a decomposition by commodity is a natural choice. In both the OPTLP and OPTMILP procedures, the block identifiers must be consecutive integers starting from 0. In PROC OPTMODEL, the block identifiers only need to be numeric. The following **FOR** loop populates the **.block** constraint suffix with block identifier $k - 1$ for commodity k:

```
for{i in NODES, k in COMMODITIES}
    BalanceCon[i,k].block = k - 1;
```

The **.block** constraint suffix for the linking (capacity) constraints is left missing, so these constraints become part of the master problem.

The following SOLVE statement uses the DECOMP= option to invoke the decomposition algorithm:

```
solve with LP / presolver=none decomp=() subprob=(algorithm=nspure);
print Flow;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. Because each subproblem is a pure network flow problem, you can use the ALGORITHM=NSPURE option in the SUBPROB= option to request that a network simplex algorithm for pure networks be used instead of the default algorithm, which for linear programming subproblems is primal simplex.

It turns out for this example that if you specify METHOD=NETWORK (instead of the default METHOD=USER) in the DECOMP= option, the network extractor finds the same blocks, one per commodity. To invoke the METHOD=NETWORK option, simply change the SOLVE statement as follows:

```
solve with LP / presolver=none decomp=(method=network);
```

In this case, the default subproblem solver is NSPURE.

The optimal solution and solution summary are displayed in Output 13.1.1.

Output 13.1.1 Solution Summary and Optimal Solution

```
                    The OPTMODEL Procedure

                     Solution Summary

         Solver                              LP
         Algorithm                Decomposition
         Objective Function           TotalCost
         Solution Status               Optimal
         Objective Value                    150

         Primal Infeasibility                 0
         Dual Infeasibility                   0
         Bound Infeasibility                  0

         Iterations                           4
         Presolve Time                     0.00
         Solution Time                     0.11


              [1]     [2]     [3]     Flow

               1       2       1        5
               1       2       2        0
               1       3       1        5
               1       3       2        0
               3       4       1        5
               3       4       2        5
               4       2       1        5
               4       2       2        0
               4       6       1        0
               4       6       2        5
               5       3       1        0
               5       3       2        5
               5       6       1        0
               5       6       2       15
```

The optimal solution is shown on the network in Figure 13.4.

Figure 13.4 Optimal Flow on Network with Two Commodities
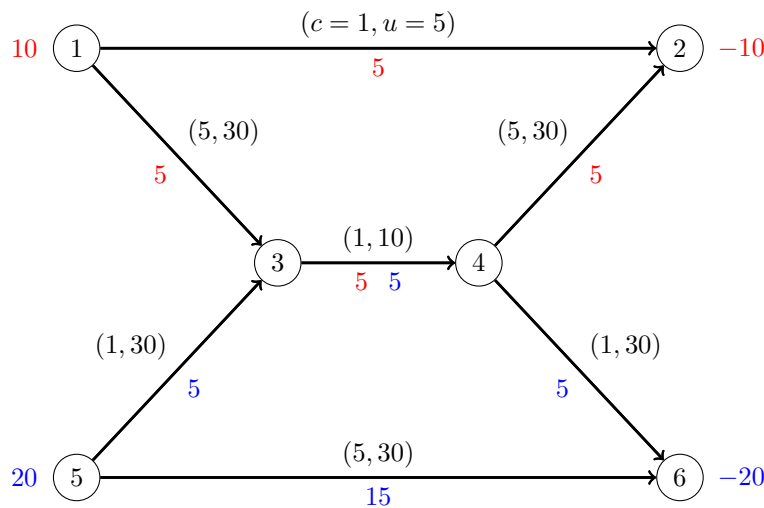
The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.1.2.

Output 13.1.2 Log

```
NOTE: There were 14 observations read from the data set WORK.ARC_COMM_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 14 variables (0 free, 0 fixed).
NOTE: The problem has 19 linear constraints (7 LE, 12 EQ, 0 GE, 0 range).
NOTE: The problem has 42 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 2 disjoint blocks.
NOTE: The decomposition subproblems cover 14 (100.00%) variables and 12 (63.16%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
       Iter           Best        Master         Gap        CPU        Real
                      Bound      Objective                   Time       Time
NOTE: Starting phase 1.
          1         0.0000       15.0000    1.50e+01        0.0        0.0
          3         0.0000        0.0000       0.00%        0.0        0.0
NOTE: Starting phase 2.
          4       150.0000      150.0000       0.00%        0.0        0.0
NOTE: The Decomposition algorithm used 2 threads.
NOTE: The Decomposition algorithm time is 0.01 seconds.
NOTE: Optimal.
NOTE: Objective = 150.
```

Example 13.2: Generalized Assignment Problem

The generalized assignment problem (GAP) is that of finding a maximum profit assignment from n tasks to m machines such that each task is assigned to precisely one machine subject to capacity restrictions on the machines. With each possible assignment, associate a binary variable x_{ij}, which, if set to 1, indicates that machine i is assigned to task j. For ease of notation, define two index sets $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$. A GAP can be formulated as a MILP as follows:

$$\text{maximize} \quad \sum_{i \in M} \sum_{j \in N} p_{ij} x_{ij}$$

$$\text{subject to} \quad \sum_{i \in M} x_{ij} = 1 \qquad j \in N \qquad \text{(assignment)}$$

$$\sum_{j \in N} w_{ij} x_{ij} \leq b_i \qquad i \in M \qquad \text{(knapsack)}$$

$$x_{ij} \in \{0, 1\} \qquad i \in M, \ j \in N$$

In this formulation, constraints (assignment) ensure that each task is assigned to exactly one machine. Inequalities (knapsack) ensure that for each machine, the capacity restrictions are met.

Consider the following example taken from Koch et al. (2011) with $n = 24$ tasks to be assigned to $m = 8$ machines. The data set profit_data provides the profit for assigning a particular task to a particular machine:

```
%let NumTasks    = 24;
%let NumMachines = 8;

data profit_data;
   input p1-p&NumTasks;
   datalines;
25 23 20 16 19 22 20 16 15 22 15 21 20 23 20 22 19 25 25 24 21 17 23 17
16 19 22 22 19 23 17 24 15 24 18 19 20 24 25 25 19 24 18 21 16 25 15 20
20 18 23 23 23 17 19 16 24 24 17 23 19 22 23 25 23 18 19 24 20 17 23 23
16 16 15 23 15 15 25 22 17 20 19 16 17 17 20 17 17 18 16 18 15 25 22 17
17 23 21 20 24 22 25 17 22 20 16 22 21 23 24 15 22 25 18 19 19 17 22 23
24 21 23 17 21 19 19 17 18 24 15 15 17 18 15 24 19 21 23 24 17 20 16 21
18 21 22 23 22 15 18 15 21 22 15 23 21 25 25 23 20 16 25 17 15 15 18 16
19 24 18 17 21 18 24 25 18 23 21 15 24 23 18 18 23 23 16 20 20 19 25 21
;
```

The data set weight_data provides the amount of resources used by a particular task when assigned to a particular machine:

```
data weight_data;
   input w1-w&NumTasks;
   datalines;
 8 18 22  5 11 11 22 11 17 22 11 20 13 13  7 22 15 22 24  8  8 24 18  8
24 14 11 15 24  8 10 15 19 25  6 13 10 25 19 24 13 12  5 18 10 24  8  5
22 22 21 22 13 16 21  5 25 13 12  9 24  6 22 24 11 21 11 14 12 10 20  6
13  8 19 12 19 18 10 21  5  9 11  9 22  8 12 13  9 25 19 24 22  6 19 14
25 16 13  5 11  8  7  8 25 20 24 20 11  6 10 10  6 22 10 10 13 21  5 19
19 19  5 11 22 24 18 11  6 13 24 24 22  6 22  5 14  6 16 11  6  8 18 10
```

```
    24 10   9 10   6 15   7 13 20   8   7   9 24   9 21   9 11 19 10   5 23 20   5 21
     6  9   9   5 12 10 16 15 19 18 20 18 16 21 11 12 22 16 21 25   7 14 16 10
    ;
```

Finally, the data set capacity_data provides the resource capacity for each machine:

```
    data capacity_data;
       input b @@;
       datalines;
    36 35 38 34 32 34 31 34
    ;
```

The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```
    proc optmodel;
       /* declare index sets */
       set TASKS    = 1..&NumTasks;
       set MACHINES = 1..&NumMachines;

       /* declare parameters */
       num profit   {MACHINES, TASKS};
       num weight   {MACHINES, TASKS};
       num capacity {MACHINES};

       /* read data sets to populate data */
       read data profit_data   into [i=_n_] {j in TASKS} <profit[i,j]=col('p'||j)>;
       read data weight_data   into [i=_n_] {j in TASKS} <weight[i,j]=col('w'||j)>;
       read data capacity_data into [_n_] capacity=b;
```

The following statements declare the optimization model:

```
       /* declare decision variables */
       var Assign {MACHINES, TASKS} binary;

       /* declare objective */
       max TotalProfit =
          sum {i in MACHINES, j in TASKS} profit[i,j] * Assign[i,j];

       /* declare constraints */
       con AssignmentCon {j in TASKS}:
          sum {i in MACHINES} Assign[i,j] = 1;

       con KnapsackCon {i in MACHINES}:
          sum {j in TASKS} weight[i,j] * Assign[i,j] <= capacity[i];
```

The following statements use two different decompositions to solve the problem. The first decomposition defines each assignment constraint as a block and uses the pure network simplex solver for the subproblem. The second decomposition defines each knapsack constraint as a block and uses the MILP solver for the subproblem.

```
       /* each assignment constraint defines a block */
       for{j in TASKS}
          AssignmentCon[j].block = j;

       solve with milp / logfreq=1000
          decomp        =()
```

```
        decomp_subprob=(algorithm=nspure);

    /* each knapsack constraint defines a block */
    for{j in TASKS}
        AssignmentCon[j].block = .;
    for{i in MACHINES}
        KnapsackCon[i].block = i;

    solve with milp / decomp=();
quit;
```

The solution summaries are displayed in Output 13.2.1.

Output 13.2.1 Solution Summaries

```
                        The OPTMODEL Procedure

                          Solution Summary

          Solver                               MILP
          Algorithm                   Decomposition
          Objective Function            TotalProfit
          Solution Status   Optimal within Relative Gap
          Objective Value                        563

          Relative Gap                  0.0000994814
          Absolute Gap                  0.0560135845
          Primal Infeasibility                     0
          Bound Infeasibility                      0
          Integer Infeasibility                    0

          Best Bound                    563.05601358
          Nodes                                 8809
          Iterations                            9366
          Presolve Time                         0.01
          Solution Time                        44.06


                          Solution Summary

            Solver                             MILP
            Algorithm                 Decomposition
            Objective Function          TotalProfit
            Solution Status                 Optimal
            Objective Value                     563

            Relative Gap                          0
            Absolute Gap                          0
            Primal Infeasibility                  0
            Bound Infeasibility                   0
            Integer Infeasibility                 0

            Best Bound                          563
            Nodes                                 7
            Iterations                           44
            Presolve Time                      0.01
            Solution Time                      0.97
```

The iteration log for both decompositions is shown in Output 13.2.2. This example is interesting because it shows the tradeoff between the strength of the relaxation and the difficulty of its resolution. In the first decomposition, the subproblems are totally unimodular and can be solved trivially. Consequently, each iteration of the decomposition algorithm is very fast. However, the bound obtained is as weak as the bound found in direct methods (the LP bound). The weaker bound leads to the need to enumerate more nodes overall. Alternatively, in the second decomposition, the subproblem is the knapsack problem, which is solved using MILP. In this case, the bound is much tighter and the problem solves in very few nodes. The tradeoff, of course, is that each iteration takes longer because solving the knapsack problem is not trivial. Another interesting aspect of this problem is that the subproblem coverage in the second decomposition is much smaller than that of the first decomposition. However, when dealing with MILP, it is not always the size of the coverage that determines the overall effectiveness of a particular choice of decomposition.

Output 13.2.2 Log

```
NOTE: There were 8 observations read from the data set WORK.PROFIT_DATA.
NOTE: There were 8 observations read from the data set WORK.WEIGHT_DATA.
NOTE: There were 8 observations read from the data set WORK.CAPACITY_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 192 variables (0 free, 0 fixed).
NOTE: The problem has 192 binary and 0 integer variables.
NOTE: The problem has 32 linear constraints (8 LE, 24 EQ, 0 GE, 0 range).
NOTE: The problem has 384 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The subproblem solver chosen is an LP solver but at least one block has integer
      variables.
NOTE: The decomposition subproblems consist of 24 disjoint blocks.
NOTE: The decomposition subproblems cover 192 (100.00%) variables and 24 (75.00%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter         Best        Master        Best        LP        IP   CPU   Real
                   Bound     Objective     Integer       Gap       Gap  Time   Time
NOTE: Starting phase 1.
         1       0.0000        8.9248          .  8.92e+00         .     0      0
         4       0.0000        0.0000          .     0.00%         .     0      0
NOTE: Starting phase 2.
         5     574.0000      561.1588          .     2.24%         .     0      0
         6     568.8833      568.5610          .     0.06%         .     0      0
         7     568.6464      568.6464          .     0.00%         .     0      0
         .     568.6464      568.6464    562.0000     0.00%     1.17%     0      0
NOTE: Starting branch and bound.
       Node   Active   Sols        Best          Best       Gap      CPU   Real
                                Integer         Bound               Time   Time
          0        1      1    562.0000      568.6464     1.17%        0      0
       1000      860      1    562.0000      565.1615     0.56%        5      4
       2000     1534      1    562.0000      564.5238     0.45%       10      9
       3000     1986      1    562.0000      564.1515     0.38%       16     14
       4000     2258      1    562.0000      563.8829     0.33%       21     19
       5000     2270      1    562.0000      563.6617     0.29%       27     24
       6000     2104      1    562.0000      563.4776     0.26%       32     29
       7000     1694      1    562.0000      563.3076     0.23%       38     34
       8000     1092      1    562.0000      563.1618     0.21%       44     39
       8242      884      2    563.0000      563.1363     0.02%       46     41
       8808      318      2    563.0000      563.0560     0.01%       49     44
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 44.19 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 563.
```

Output 13.2.2 *continued*

```
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 8 disjoint blocks.
NOTE: The decomposition subproblems cover 192 (100.00%) variables and 8 (25.00%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter        Best        Master         Best        LP       IP   CPU  Real
                  Bound     Objective      Integer      Gap      Gap  Time  Time
NOTE: Starting phase 1.
        1        0.0000        7.0000           .  7.00e+00        .    0     0
        5        0.0000        0.0000           .     0.00%        .    0     0
NOTE: Starting phase 2.
        6      672.9221      499.7295           .    25.74%        .    0     0
        8      653.1045      520.3731           .    20.32%        .    0     0
        9      607.0000      528.1905           .    12.98%        .    0     0
       10      607.0000      539.1556           .    11.18%        .    0     0
        .      607.0000      547.7045      547.0000    9.77%    9.88%    0     0
       12      605.4928      548.8623      547.0000    9.35%    9.66%    0     0
       13      603.2619      552.0952      547.0000    8.48%    9.33%    0     0
       14      595.3226      555.3548      547.0000    6.71%    8.12%    0     0
       15      590.4167      557.7500      547.0000    5.53%    7.35%    0     0
       16      579.6429      558.5893      547.0000    3.63%    5.63%    0     0
       17      576.5000      560.8750      547.0000    2.71%    5.12%    0     0
       18      570.0000      563.4286      547.0000    1.15%    4.04%    0     0
       19      566.2500      563.5000      547.0000    0.49%    3.40%    0     0
       20      565.0000      564.0000      547.0000    0.18%    3.19%    0     0
        .      565.0000      564.0000      562.0000    0.18%    0.53%    0     0
       22      564.0000      564.0000      562.0000    0.00%    0.35%    0     0
        .      564.0000      564.0000      562.0000    0.00%    0.35%    0     0
NOTE: Starting branch and bound.
         Node  Active   Sols       Best          Best       Gap     CPU   Real
                                 Integer        Bound             Time   Time
            0       1      3     562.0000      564.0000     0.35%      0      0
            5       1      4     563.0000      563.5000     0.09%      1      0
            6       0      4     563.0000      563.0000     0.00%      1      0
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.90 seconds.
NOTE: Optimal.
NOTE: Objective = 563.
```

Example 13.3: Block-Diagonal Structure and METHOD=AUTO in Single-Machine Mode

This example demonstrates how you can use the METHOD=AUTO option in the DECOMP statement to execute the decomposition algorithm in single-machine mode.

Consider a mixed integer linear program that is defined by the MPS data set mpsdata. In this case, the structure of the model is unknown and only the MPS data set is provided to you.

The following PROC OPTMILP statements attempt to solve the problem by using standard methods and a 15-second time limit:

```
proc optmilp
   data    = mpsdata
   logfreq = 2000
   maxtime = 15;
run;
```

The solution summary is shown in Output 13.3.1.

Output 13.3.1 Solution Summary

```
                    The OPTMILP Procedure

                      Solution Summary

          Solver                             MILP
          Algorithm                 Branch and Cut
          Objective Function              R0001298
          Solution Status       Time Limit Reached
          Objective Value                      133

          Relative Gap               0.3965751025
          Absolute Gap               37.767026308
          Primal Infeasibility                  0
          Bound Infeasibility                   0
          Integer Infeasibility                 0

          Best Bound                 95.232973692
          Nodes                              2268
          Iterations                        76252
          Presolve Time                      0.03
          Solution Time                     14.96
```

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 13.3.2.

Output 13.3.2 Log

```
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block
      covers 25.08% of the rows in the problem. The DECOMP option with METHOD=AUTO is
      recommended for solving problems with this structure.
          Node    Active    Sols    BestInteger       BestBound        Gap     Time
             0         1       1    231.0000000               0      231.0        0
             0         1       1    231.0000000      91.4479396    152.60%        0
             0         1       2    198.0000000      91.8607875    115.54%        0
             0         1       2    198.0000000      92.0019485    115.21%        0
             0         1       2    198.0000000      92.0404035    115.12%        0
             0         1       2    198.0000000      92.0404035    115.12%        1
NOTE: The MILP solver added 6 cuts with 32 cut coefficients at the root.
           787        36       3    135.0000000      95.2329737     41.76%        6
           957       185       4    133.0000000      95.2329737     39.66%        7
          2000      1075       4    133.0000000      95.2329737     39.66%       13
          2281      1298       4    133.0000000      95.2329737     39.66%       14
NOTE: CPU time limit reached.
NOTE: Objective of the best integer solution found = 133.
```

A note in the log suggests that you can use the decomposition algorithm because of the structure of the problem. The following PROC OPTMILP statements use the METHOD=AUTO option in the DECOMP statement in single-machine mode. The PERFORMANCE statement specifies the number of threads to be used.

```
proc optmilp
   data       = mpsdata;
   decomp
     loglevel = 2
     method   = auto;
   subprob
     loglevel = 2;
   performance
     nthreads = 4;
run;
```

The performance information and solution summary are displayed in Output 13.3.3.

Output 13.3.3 Performance Information and Solution Summary

```
                    The OPTMILP Procedure

                  Performance Information

          Execution Mode        Single-Machine
          Number of Threads     4


                     Solution Summary

          Solver                          MILP
          Algorithm              Decomposition
          Objective Function         R0001298
          Solution Status             Optimal
          Objective Value                  120

          Relative Gap                       0
          Absolute Gap                       0
          Primal Infeasibility               0
          Bound Infeasibility                0
          Integer Infeasibility              0

          Best Bound                       120
          Nodes                              1
          Iterations                         1
          Presolve Time                   0.03
          Solution Time                   2.48
```

The iteration log, which contains the problem statistics and the progress of the solution, is shown in . When you specify NTHREADS=4 in the PERFORMANCE statement in single-machine mode, each block is processed simultaneously on each of four threads.

Output 13.3.4 Log

```
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The OPTMILP procedure is executing in single-machine mode.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The DECOMP method value AUTO is applied.
NOTE: The decomposition subproblems consist of 4 disjoint blocks.
NOTE: The decomposition subproblems cover 351 (100.00%) variables and 1260 (100.00%)
      constraints.
NOTE: Block 0 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 1 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 2 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 3 has 87 (24.79%) variables and 312 (24.76%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
NOTE: ------------------------------------------------------------------------------
NOTE: Starting to process node 0.
NOTE: ------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------
NOTE: The subproblem solver for 4 blocks at iteration 0 is starting.
NOTE: ------------------------------------------------------------------------------
NOTE: The subproblem solver for block 0 at iteration 0 is starting on thread 1.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 88 variables, 316 constraints, and 948 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node   Active    Sols    BestInteger        BestBound      Gap     Time
            0        1       1            -16              -56   71.43%        0
            0        1       1            -16   -34.1037735849   53.08%        0
            0        1       1            -16   -34.1037735849   53.08%        0
           41       34       2            -22   -32.8571428571   33.04%        0
           80       49       3            -24   -31.5256410256   23.87%        0
          100       56       3            -24   -30.8823529412   22.29%        0
          134       35       4            -27         -30.1875   10.56%        0
          200       34       4            -27            -28.4    4.93%        1
          253        0       4            -27              -27    0.00%        1
NOTE: Optimal.
NOTE: Objective = -27.
NOTE: The subproblem solver for block 0 used 1.26 (cpu: 4.88) seconds.
```

Output 13.3.4 *continued*

```
NOTE: --------------------------------------------------------------------------------
NOTE: --------------------------------------------------------------------------------
NOTE: The subproblem solver for block 1 at iteration 0 is starting on thread 2.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 88 variables, 316 constraints, and 948 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node  Active   Sols    BestInteger       BestBound      Gap    Time
            0       1      1              0             -59  100.00%       0
            0       1      1              0  -35.3671970624  100.00%       0
            0       1      2            -12  -35.3671970624   66.07%       0
            0       1      3            -22  -35.3671970624   37.80%       0
            0       1      3            -22  -35.3671970624   37.80%       0
           29      25      4            -25  -34.6201565606   27.79%       0
          100      71      5            -25  -33.3904056593   25.13%       0
          166      89      7            -27  -32.8646661367   17.84%       1
          186      52      8            -30  -32.3042378242    7.13%       1
          200      48      8            -30  -32.0393180237    6.37%       1
          267       0      8            -30             -30    0.00%       1
NOTE: Optimal.
NOTE: Objective = -30.
NOTE: The subproblem solver for block 1 used 1.43 (cpu: 5.40) seconds.
NOTE: --------------------------------------------------------------------------------
NOTE: --------------------------------------------------------------------------------
NOTE: The subproblem solver for block 3 at iteration 0 is starting on thread 4.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 87 variables, 312 constraints, and 936 constraint
      coefficients.
NOTE: The MILP solver is called.
         Node  Active   Sols    BestInteger       BestBound      Gap    Time
            0       1      1            -16             -57   71.93%       0
            0       1      1            -16  -33.8743589744   52.77%       0
            0       1      1            -16  -33.8133333333   52.68%       0
            0       1      1            -16  -33.7577927973   52.60%       0
            0       1      1            -16  -33.7153309512   52.54%       0
            0       1      2            -19  -33.7153309512   43.65%       0
            0       1      2            -19  -33.7153309512   43.65%       0
NOTE: The MILP solver added 3 cuts with 18 cut coefficients at the root.
            6       6      3            -20  -33.5685920578   40.42%       0
           56      44      4            -22  -32.7647058824   32.85%       0
           81      58      5            -23  -32.4088337685   29.03%       0
          100      69      5            -23  -32.1129411765   28.38%       0
          200     114      6            -23  -30.3005780347   24.09%       1
          290     129      8            -24  -29.3214285714   18.15%       1
          299      83      9            -26  -29.2063492063   10.98%       1
          300      82      9            -26  -29.1785714286   10.89%       1
          400      40      9            -26  -27.1111111111    4.10%       1
          449       0      9            -26             -26    0.00%       1
NOTE: Optimal.
NOTE: Objective = -26.
NOTE: The subproblem solver for block 3 used 1.88 (cpu: 6.30) seconds.
```

Output 13.3.4 *continued*

```
NOTE: -------------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------------
NOTE: The subproblem solver for block 2 at iteration 0 is starting on thread 3.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 88 variables, 316 constraints, and 948 constraint
      coefficients.
NOTE: The MILP solver is called.
        Node   Active    Sols    BestInteger       BestBound      Gap     Time
           0        1       1             -9             -59   84.75%        0
           0        1       1             -9  -36.2067307692   75.14%        0
           0        1       2            -14  -35.8906956361   60.99%        0
           0        1       2            -14  -35.7582989209   60.85%        0
           0        1       2            -14  -35.7308331027   60.82%        0
           0        1       2            -14  -35.7308331027   60.82%        0
NOTE: The MILP solver added 3 cuts with 14 cut coefficients at the root.
          53       42       4            -19  -34.6324177369   45.14%        0
          67       51       6            -20  -34.3787806049   41.82%        0
          70       53       7            -21  -34.3787806049   38.92%        0
         100       72       8            -21  -34.0784347526   38.38%        0
         157      102       9            -22   -33.326652221   33.99%        0
         200      124       9            -22  -32.8793103448   33.09%        1
         233      130      11            -24  -32.5099569429   26.18%        1
         244      104      12            -27        -32.4375   16.76%        1
         267       90      13            -28  -32.0592920354   12.66%        1
         300      100      13            -28  -31.6450098814   11.52%        1
         400       69      14            -28  -29.6666666667    5.62%        1
         489        0      14            -28             -28    0.00%        1
NOTE: Optimal.
NOTE: Objective = -28.
NOTE: The subproblem solver for block 2 used 1.98 (cpu: 6.41) seconds.
NOTE: -------------------------------------------------------------------------------
NOTE: The subproblem solver for 4 blocks used 1.99 (cpu: 6.41) seconds.
NOTE: -------------------------------------------------------------------------------
NOTE: The initial column pool after generating initial variables contains 4 columns.
      Iter        Best         Master         Best        LP        IP    CPU   Real
                  Bound      Objective      Integer       Gap       Gap   Time   Time
NOTE: Starting phase 2.
         1     120.0000       120.0000     120.0000     0.00%     0.00%      6      1
NOTE: The number of active nodes is 0.
NOTE: The objective value of the best integer feasible solution is 120.0000 and the
      best bound is 120.0000.
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 2.00 seconds.
NOTE: Optimal.
NOTE: Objective = 120.
```

In this case, the solver found that, after presolve, the constraint matrix decomposed into block-diagonal form. That is, all the constraints are covered by subproblem blocks, leaving the set of master constraints empty. With no coupling constraints, the problem decomposes into four completely independent problems. If you specify LOGLEVEL=2 in the DECOMP statement, the log displays the size of each block. The blocks in this case are nicely balanced, allowing parallel execution to be efficient, and the solver finds the optimal solution almost immediately.

Example 13.4: Block-Diagonal Structure and METHOD=AUTO in Distributed Mode

This example demonstrates how you can use the METHOD=AUTO option in the DECOMP statement to execute the decomposition algorithm in distributed mode.

As in Example 13.3, consider a mixed integer linear program that is defined by the MPS data set mpsdata. In this case, the structure of the model is unknown and only the MPS data set is provided to you.

The following PROC OPTMILP statements use the METHOD=AUTO option in distributed mode. The PERFORMANCE statement specifies the number of threads and nodes to be used.

```
proc optmilp
   data       = mpsdata;
   decomp
     loglevel = 2
     method   = auto;
   performance
     details
     nthreads = 1
     nodes    = 4;
run;
```

The performance information and solution summary are displayed in Output 13.4.1.

Output 13.4.1 Performance Information and Solution Summary

```
               The OPTMILP Procedure


              Performance Information

    Host Node                    <<your grid host>>
    Execution Mode               Distributed
    Grid Mode                    Symmetric
    Number of Compute Nodes      4
    Number of Threads per Node   1
```

Output 13.4.1 *continued*

```
                    Solution Summary

        Solver                           MILP
        Algorithm               Decomposition
        Objective Function          R0001298
        Solution Status              Optimal
        Objective Value                  120

        Relative Gap                       0
        Absolute Gap                       0
        Primal Infeasibility               0
        Bound Infeasibility                0
        Integer Infeasibility              0

        Best Bound                       120
        Nodes                              1
        Iterations                         1
        Presolve Time                   0.03
        Solution Time                   1.20
```

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 13.4.2. When you specify NODES=4 and NTHREADS=1 in the PERFORMANCE statement in distributed mode, each block is processed simultaneously on each of four grid nodes.

Output 13.4.2 Log

```
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The OPTMILP procedure is executing in the distributed computing environment
      with 4 worker nodes.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The DECOMP method value AUTO is applied.
NOTE: The decomposition subproblems consist of 4 disjoint blocks.
NOTE: The decomposition subproblems cover 351 (100.00%) variables and 1260 (100.00%)
      constraints.
NOTE: Block 0 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 1 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 2 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 3 has 87 (24.79%) variables and 312 (24.76%) constraints.
NOTE: ---------------------------------------------------------------
NOTE: Starting to process node 0.
NOTE: ---------------------------------------------------------------
NOTE: The subproblem solver for 4 blocks at iteration 0 is starting.
NOTE: The subproblem solver for 4 blocks used 1.24 (cpu: 0.00) seconds.
NOTE: The initial column pool after generating initial variables contains 4 columns.
      Iter          Best        Master         Best      LP      IP  Real
                   Bound     Objective       Integer     Gap     Gap  Time
NOTE: Starting phase 2.
         1      120.0000      120.0000      120.0000   0.00%   0.00%     1
NOTE: The number of active nodes is 0.
NOTE: The objective value of the best integer feasible solution is 120.0000 and the
      best bound is 120.0000.
NOTE: The Decomposition algorithm time is 1.25 seconds.
NOTE: Optimal.
NOTE: Objective = 120.
```

Example 13.5: Resource Allocation Problem

This example describes a model for selecting tasks to be run on a shared resource (Gamrath 2010). Consider a set I of tasks and a resource capacity C. Each item $i \in I$ has a profit p_i, a resource utilization level w_i, a starting period s_i, and an ending period e_i. The time horizon considered is from the earliest starting time to the latest ending time of all tasks. With each task, associate a binary variable x_i, which, if set to 1, indicates that the task is running from its start time until just before its end time. A task consumes capacity if it is running. The goal is to select which tasks to run in order to maximize profit while not exceeding the shared resource capacity. Let $S = \{s_i \mid i \in I\}$ define the set of start times for all tasks, and let $L_s = \{i \in I \mid s_i \leq s < e_i\}$ define the set of tasks that are running at each start time $s \in S$. The problem

can be modeled as a mixed integer linear programming problem as follows:

maximize $\quad \sum_{i \in I} p_i x_i$

subject to $\quad \sum_{i \in L_s} w_i x_i \leq C \qquad\qquad s \in S \qquad\qquad$ (capacity)

$\qquad\qquad\qquad x_i \in \{0, 1\} \qquad\qquad i \in I$

In this formulation, constraints (capacity) ensure that the running tasks do not exceed the resource capacity. To illustrate, consider the following five-task example with data: $p_i = (6, 8, 5, 9, 8)$, $w_i = (8, 5, 3, 4, 3)$, $s_i = (1, 3, 5, 7, 8)$, $e_i = (5, 8, 9, 17, 10)$, and $C = 10$. The formulation leads to a constraint matrix that has a *staircase structure* that is determined by tasks coming on and offline:

$$
\begin{array}{rccccccccc}
\text{maximize} & 6x_1 & + & 8x_2 & + & 5x_3 & + & 9x_4 & + & 8x_5 \\
\text{subject to} & 8x_1 & & & & & & & & & \leq & 10 \\
& 8x_1 & + & 5x_2 & & & & & & & \leq & 10 \\
& & & 5x_2 & + & 3x_3 & & & & & \leq & 10 \\
& & & 5x_2 & + & 3x_3 & + & 4x_4 & & & \leq & 10 \\
& & & & & 3x_3 & + & 4x_4 & + & 3x_5 & \leq & 10 \\
\end{array}
$$

$$x_i \in \{0, 1\} \quad i \in I$$

Lagrangian Decomposition

This formulation clearly has no decomposable structure. However, you can use a common modeling technique known as *Lagrangian decomposition* to bring the model into block-angular form. Lagrangian decomposition works by first partitioning the constraints into blocks. Then, each original variable is split into multiple copies of itself, one copy for each block in which the variable has a nonzero coefficient in the constraint matrix. Constraints are added to enforce the equality of each copy of the original variable. Then, the original constraints can be written in block-angular form by using the duplicate variables.

To apply Lagrangian decomposition to the resource allocation problem, define a set B of blocks and let S_b define the set of start times for a given block b, such that $S = \cup_{b \in B} S_b$. Given this partition of start times, let B_i define the set of blocks in which task $i \in I$ is scheduled to be running. Now, for each task $i \in I$, define duplicate variables x_i^b for each $b \in B_i$. Let m_i define the minimum block index for each class of variable that represents task i. The problem can now be modeled in block-angular form as follows:

maximize $\quad \sum_{i \in I} p_i x_i^{m_i}$

subject to $\qquad\quad x_i^b = x_i^{m_i} \qquad\qquad i \in I, \ b \in B_i \setminus \{m_i\} \qquad$ (linking)

$\qquad\qquad \sum_{i \in L_s} w_i x_i^b \leq C \qquad\qquad b \in B, \ s \in S_b \qquad\qquad$ (capacity)

$\qquad\qquad\qquad x_i^b \in \{0, 1\} \qquad\qquad i \in I, \ b \in B_i$

In this formulation, constraints (linking) ensure that the duplicate variables are equal to the original variables.

Now, the five-task example has been transformed from a staircase structure to a block-angular structure:

$$
\begin{array}{llllllll}
\text{maximize} & 6x_1^1 & + & 8x_2^1 & + & 5x_3^2 & + & 9x_4^2 & + & & & & 8x_5^3 \\
\text{subject to} & & & x_2^1 & - & x_2^2 & & & & & & & & = & 0 \\
& & & & & x_3^2 & - & & & x_3^3 & & & & = & 0 \\
& & & & & & & x_4^2 & & & - & x_4^3 & & = & 0 \\
& 8x_1^1 & & & & & & & & & & & & \leq & 10 \\
& 8x_1^1 & + & 5x_2^1 & & & & & & & & & & \leq & 10 \\
& & & & & 5x_2^2 & + & 3x_3^2 & & & & & & \leq & 10 \\
& & & & & 5x_2^2 & + & 3x_3^2 & + & 4x_4^2 & & & & \leq & 10 \\
& & & & & & & & & 3x_3^3 & + & 4x_4^3 & + & 3x_5^3 & \leq & 10
\end{array}
$$

$$x_i^b \in \{0,1\} \quad i \in I, \ b \in B_i$$

To show how to apply Lagrangian decomposition in PROC OPTMODEL, consider the following data set TaskData from Caprara, Furini, and Malaguti (2010) which consists of $|I| = 2697$ tasks:

```
data TaskData;
   input profit weight start end;
   datalines;
100 74   1   12
 98 32   1    9
 73 27   1   22
 98 51   1   31
...
 23 40 2684 2689
 36 85 2685 2687
 65 44 2686 2689
 18 36 2687 2689
 88 57 2688 2689
;
```

Using the MILP Solver Directly in PROC OPTMODEL

The following PROC OPTMODEL statements read in the data and solve the original staircase formulation by calling the MILP solver directly:

```
%macro SetupData(task_data=, capacity=);
   set TASKS;
   num capacity=&capacity;
   num profit{TASKS}, weight{TASKS}, start{TASKS}, end{TASKS};

   read data &task_data into TASKS=[_n_] profit weight start end;
   /* the set of start times */

   set STARTS = setof{i in TASKS} start[i];
   /* the set of tasks i that are active at a given start time s */
   set TASKS_START{s in STARTS}
      = {i in TASKS: start[i] <= s < end[i]};
%mend SetupData;

%macro ResourceAllocation_Direct(task_data=, capacity=);
   proc optmodel;
```

```
      %SetupData(task_data=&task_data,capacity=&capacity);

      /* select task i to come online from period [start to end) */
      var x{TASKS} binary;

      /* maximize the total profit of running tasks */
      max TotalProfit = sum{i in TASKS} profit[i] * x[i];

      /* enforce that the shared resource capacity is not exceeded */
      con CapacityCon{s in STARTS}:
         sum{i in TASKS_START[s]} weight[i] * x[i] <= capacity;

      solve;
   quit;
%mend ResourceAllocation_Direct;

%ResourceAllocation_Direct(task_data=TaskData, capacity=100);
```

The problem summary and solution summary are displayed in Output 13.5.1.

Output 13.5.1 Problem Summary and Solution Summary

```
                     The OPTMODEL Procedure

                        Problem Summary

           Objective Sense            Maximization
           Objective Function          TotalProfit
           Objective Type                   Linear

           Number of Variables                2697
           Bounded Above                         0
           Bounded Below                         0
           Bounded Below and Above            2697
           Free                                  0
           Fixed                                 0
           Binary                             2697
           Integer                               0

           Number of Constraints              2688
           Linear LE (<=)                     2688
           Linear EQ (=)                         0
           Linear GE (>=)                        0
           Linear Range                          0

           Constraint Coefficients           26880
```

Output 13.5.1 *continued*

```
                    Solution Summary

       Solver                                  MILP
       Algorithm                     Branch and Cut
       Objective Function                 TotalProfit
       Solution Status     Optimal within Relative Gap
       Objective Value              62523.999618

       Relative Gap                     1.5493144E-8
       Absolute Gap                    0.0009686933
       Primal Infeasibility                        0
       Bound Infeasibility            1.776357E-15
       Integer Infeasibility          7.3474993E-6

       Best Bound                      62524.000587
       Nodes                                     441
       Iterations                              26866
       Presolve Time                            1.53
       Solution Time                          261.86
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.5.2.

Output 13.5.2 Log

```
NOTE: There were 2697 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2697 variables (0 free, 0 fixed).
NOTE: The problem has 2697 binary and 0 integer variables.
NOTE: The problem has 2688 linear constraints (2688 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 26880 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 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 2697 variables, 2688 constraints, and 26880
      constraint coefficients.
NOTE: The MILP solver is called.
         Node   Active   Sols    BestInteger       BestBound       Gap    Time
            0        1      3   54182.0000000          145710    62.82%       2
            0        1      3   54182.0000000   73230.2096818    26.01%       2
            0        1      6   60619.0000000   70135.8936905    13.57%       3
            0        1      7   61216.0000000   68409.0687245    10.51%       4
            0        1      7   61216.0000000   67068.0202769     8.73%       5
            0        1      7   61216.0000000   66038.0918000     7.30%       6
            0        1      7   61216.0000000   65190.7829592     6.10%       7
            0        1      7   61216.0000000   64529.6958613     5.14%       8
            0        1      7   61216.0000000   64125.0294474     4.54%       9
            0        1      7   61216.0000000   63815.0136796     4.07%      10
            0        1      7   61216.0000000   63608.2185226     3.76%      10
            0        1      7   61216.0000000   63404.4123225     3.45%      10
            0        1      7   61216.0000000   63294.7183240     3.28%      11
            0        1      7   61216.0000000   63177.6963133     3.11%      11
            0        1      7   61216.0000000   63113.0842876     3.01%      11
            0        1      7   61216.0000000   63066.7776464     2.93%      11
            0        1      7   61216.0000000   62999.6013180     2.83%      11
            0        1      7   61216.0000000   62956.5482043     2.76%      11
            0        1      7   61216.0000000   62927.3931008     2.72%      11
            0        1      7   61216.0000000   62910.5094398     2.69%      12
            0        1      7   61216.0000000   62896.8720091     2.67%      12
            0        1      7   61216.0000000   62881.4558122     2.65%      12
            0        1      7   61216.0000000   62865.7212258     2.62%      12
            0        1      8   61282.0000000   62865.7212258     2.52%      12
            0        1      8   61282.0000000   62865.7212258     2.52%      12
NOTE: The MILP solver added 1257 cuts with 8396 cut coefficients at the root.
           63       61     12   62424.0000000   62814.4025787     0.62%      47
          100       97     12   62424.0000000   62807.4025787     0.61%      65
          161      151     13   62446.0000000   62758.4665642     0.50%      99
          200      187     13   62446.0000000   62739.3314203     0.47%     112
          243      175     15   62465.0000000   62718.8175688     0.40%     130
          251      124     17   62493.0000000   62717.7127884     0.36%     131
          252      110     18   62517.0000000   62715.6822161     0.32%     131
          300      102     18   62517.0000000   62603.6811810     0.14%     136
          400       26     18   62517.0000000   62524.0404223     0.01%     137
          440       36     19   62523.9996179   62524.0005866     0.00%     139
NOTE: Optimal within relative gap.
NOTE: Objective = 62523.999618.
```

Using the Decomposition Algorithm in PROC OPTMODEL

To transform this data into block-angular form, first sort the task data to help reduce the number of duplicate variables needed in the reformulation as follows:

```
proc sort data=TaskData;
   by start end;
run;
```

Then, create the partition of constraints into blocks of size **block_size** as follows:

```
%macro ResourceAllocation_Decomp(task_data=, capacity=, block_size=);
   proc optmodel;
      %SetupData(task_data=&task_data,capacity=&capacity);
      /* partition into blocks of size blocks_size */
      num block_size = &block_size;
      num num_blocks = ceil( card(TASKS) / block_size );
      set BLOCKS     = 1..num_blocks;

      /* the set of starts s for which task i is active */
      set STARTS_TASK{i in TASKS} = {s in STARTS: start[i] <= s < end[i]};

      /* partition the start times into blocks of size block_size */
      set STARTS_BLOCK{BLOCKS} init {};
      num block_id    init 1;
      num block_count init 0;
      for{s in STARTS} do;
         STARTS_BLOCK[block_id] = STARTS_BLOCK[block_id] union {s};
         block_count = block_count + 1;
         if(mod(block_count, block_size) = 0) then
             block_id = block_id + 1;
      end;
```

Then, the following PROC OPTMODEL statements define the block-angular formulation and solve the problem by using the decomposition algorithm, the PRESOLVER=BASIC option, and **block_size=40**. Because this reformulation is equivalent to the original staircase formulation, disabling some of the advanced presolver techniques ensures that the model maintains block-angularity.

```
         /* blocks in which task i is online */
         set BLOCKS_TASK{i in TASKS} =
            {b in BLOCKS: card(STARTS_BLOCK[b] inter STARTS_TASK[i]) > 0};

         /* minimum block id in which task i is online */
         num min_block{i in TASKS} = min{b in BLOCKS_TASK[i]} b;

         /* select task i to come online from period [start to end)
            in each block */
         var x{i in TASKS, b in BLOCKS_TASK[i]} binary;

         /* maximize the total profit of running tasks */
         max TotalProfit = sum{i in TASKS} profit[i] * x[i,min_block[i]];

         /* enforce that task selection is consistent across blocks */
         con LinkDupVarsCon{i in TASKS, b in BLOCKS_TASK[i] diff {min_block[i]}}:
            x[i,b] = x[i,min_block[i]];
```

```
        /* enforce that the shared resource capacity is not exceeded */
        con CapacityCon{b in BLOCKS, s in STARTS_BLOCK[b]}:
            sum{i in TASKS_START[s]} weight[i] * x[i,b] <= capacity;

        /* define blocks for decomposition algorithm */
        for{b in BLOCKS, s in STARTS_BLOCK[b]} CapacityCon[b,s].block = b;

        solve with milp / presolver=basic decomp=();
    quit;
%mend ResourceAllocation_Decomp;

%ResourceAllocation_Decomp(task_data=TaskData, capacity=100, block_size=40);
```

The problem summary and solution summary are displayed in Output 13.5.3. Compared to the original formulation, the number of variables and constraints is increased by the number of duplicate variables.

Output 13.5.3 Problem Summary and Solution Summary

```
                      The OPTMODEL Procedure

                        Problem Summary

            Objective Sense            Maximization
            Objective Function          TotalProfit
            Objective Type                   Linear

            Number of Variables                3300
            Bounded Above                         0
            Bounded Below                         0
            Bounded Below and Above            3300
            Free                                  0
            Fixed                                 0
            Binary                             3300
            Integer                               0

            Number of Constraints              3291
            Linear LE  (<=)                    2688
            Linear EQ  (=)                      603
            Linear GE  (>=)                       0
            Linear Range                          0

            Constraint Coefficients           28086
```

Output 13.5.3 *continued*

```
                     Solution Summary

        Solver                                   MILP
        Algorithm                       Decomposition
        Objective Function                 TotalProfit
        Solution Status     Optimal within Relative Gap
        Objective Value                   62524.000668

        Relative Gap                       0.0000640802
        Absolute Gap                       4.0068056181
        Primal Infeasibility               6.661338E-16
        Bound Infeasibility                8.881784E-16
        Integer Infeasibility                 8.4375E-6

        Best Bound                        62528.007474
        Nodes                                         4
        Iterations                                   55
        Presolve Time                              1.66
        Solution Time                             47.86
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.5.4.

Output 13.5.4 Log

```
NOTE: There were 2697 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3300 variables (0 free, 0 fixed).
NOTE: The problem has 3300 binary and 0 integer variables.
NOTE: The problem has 3291 linear constraints (2688 LE, 603 EQ, 0 GE, 0 range).
NOTE: The problem has 28086 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 3300 variables, 3291 constraints, and 28086
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 68 disjoint blocks.
NOTE: The decomposition subproblems cover 3300 (100.00%) variables and 2688 (81.68%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter        Best        Master        Best        LP        IP   CPU   Real
                  Bound     Objective     Integer       Gap       Gap  Time   Time
NOTE: Starting phase 1.
         1      0.0000        0.0000           .      0.00%         .     1      0
NOTE: Starting phase 2.
         .  65890.0001    54007.0000   54007.0000    18.03%    18.03%     1      0
         3  65890.0001    54659.0000   54659.0000    17.05%    17.05%     6      2
         4  65890.0001    55249.0000   55249.0000    16.15%    16.15%     8      2
         6  65890.0001    55639.0000   55639.0000    15.56%    15.56%    14      4
        10  65890.0001    57285.3335   55639.0000    13.06%    15.56%    24      7
         .  65890.0001    58309.2001   57390.0000    11.51%    12.90%    24      7
        16  65298.6567    61601.4676   57390.0000     5.66%    12.11%    41     12
        17  64224.7955    61814.7507   57390.0000     3.75%    10.64%    44     13
        18  63589.7549    62035.7507   57390.0000     2.44%     9.75%    47     14
        19  63458.0056    62107.9174   57390.0000     2.13%     9.56%    49     15
        20  63458.0056    62280.8344   57390.0000     1.86%     9.56%    52     16
         .  63458.0056    62280.8344   61498.0006     1.86%     3.09%    52     16
        21  63265.1802    62280.8344   61498.0006     1.56%     2.79%    56     17
        22  63003.0064    62407.2675   61498.0006     0.95%     2.39%    59     19
        23  62744.1043    62418.2678   61498.0006     0.52%     1.99%    62     20
        25  62682.0867    62468.8343   61498.0006     0.34%     1.89%    70     22
        26  62623.1709    62491.5343   61498.0006     0.21%     1.80%    73     23
        30  62623.1709    62507.0163   61498.0006     0.19%     1.80%    87     27
         .  62623.1709    62507.0163   61965.0006     0.19%     1.05%    87     27
        31  62589.3380    62507.0163   61965.0006     0.13%     1.00%    90     28
        32  62574.3405    62517.8345   61965.0006     0.09%     0.97%    93     29
        33  62550.3388    62521.8345   61965.0006     0.05%     0.94%    97     30
        34  62537.3384    62521.8345   61965.0006     0.02%     0.92%    99     31
        35  62532.6716    62526.6679   61965.0006     0.01%     0.91%   102     32
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
NOTE: Starting branch and bound.
```

Output 13.5.4 *continued*

Node	Active	Sols	Best Integer	Best Bound	Gap	CPU Time	Real Time
0	1	9	61965.0006	62532.6716	0.91%	102	32
3	1	11	62524.0007	62528.0075	0.01%	161	51

```
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 51.12 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 62524.000668.
```

Using a Hybrid Method in PROC OPTMODEL

The decomposition algorithm solves the problem in fewer nodes due to the stronger bound obtained by the reformulation. However, it takes longer than the direct method to find a good feasible solution. The fact that the direct method seems to quickly find good feasible solutions but has weaker bounds motivates the use of a hybrid algorithm. In the macro `%ResourceAllocation_Decomp`, replace the statement,

```
solve with milp / presolver=basic decomp=();
```

with the following statements:

```
solve with milp / relobjgap=0.1;
solve with milp / presolver=basic primalin decomp=();
```

These statements use the direct method with RELOBJGAP=0.1 to find a good starting solution and then use that result to seed the initial columns of the decomposition algorithm.

The solution summaries are displayed in Output 13.5.5.

Output 13.5.5 Solution Summaries

```
                    The OPTMODEL Procedure

                      Solution Summary

        Solver                                    MILP
        Algorithm                      Branch and Cut
        Objective Function                  TotalProfit
        Solution Status    Optimal within Relative Gap
        Objective Value                          61276

        Relative Gap                      0.0863603883
        Absolute Gap                      5792.0202769
        Primal Infeasibility                         0
        Bound Infeasibility                          0
        Integer Infeasibility                        0

        Best Bound                        67068.020277
        Nodes                                        1
        Iterations                                2523
        Presolve Time                             1.40
        Solution Time                             5.07
```

Output 13.5.5 *continued*

```
                         Solution Summary

              Solver                                   MILP
              Algorithm                       Decomposition
              Objective Function                 TotalProfit
              Solution Status     Optimal within Relative Gap
              Objective Value                  62524.000477

              Relative Gap                     0.0000559987
              Absolute Gap                     3.5014588178
              Primal Infeasibility             1.421085E-14
              Bound Infeasibility              7.327472E-15
              Integer Infeasibility            8.3333333E-6

              Best Bound                       62527.501936
              Nodes                                       3
              Iterations                                 39
              Presolve Time                            1.63
              Solution Time                           35.56
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.5.6.

Output 13.5.6 Log

```
NOTE: There were 2697 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3300 variables (0 free, 0 fixed).
NOTE: The problem has 3300 binary and 0 integer variables.
NOTE: The problem has 3291 linear constraints (2688 LE, 603 EQ, 0 GE, 0 range).
NOTE: The problem has 28086 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 603 variables and 603 constraints.
NOTE: The MILP presolver removed 1206 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 2697 variables, 2688 constraints, and 26880
      constraint coefficients.
NOTE: The MILP solver is called.
        Node   Active    Sols     BestInteger        BestBound       Gap    Time
           0        1       3   54609.0000000           145710    62.52%       2
           0        1       3   54609.0000000    73230.2096818    25.43%       2
           0        1       6   60619.0000000    70135.8936905    13.57%       3
           0        1       7   61276.0000000    68409.0687245    10.43%       4
           0        1       7   61276.0000000    67068.0202769     8.64%       5
NOTE: The MILP solver added 498 cuts with 2624 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 61276.
```

Output 13.5.6 *continued*

```
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 3300 variables, 3291 constraints, and 28086
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 68 disjoint blocks.
NOTE: The decomposition subproblems cover 3300 (100.00%) variables and 2688 (81.68%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter        Best        Master        Best        LP        IP    CPU   Real
                  Bound      Objective     Integer      Gap       Gap   Time   Time
NOTE: Starting phase 1.
         1       0.0000       0.0000            .        0.00%       .      1      0
NOTE: Starting phase 2.
         2   65889.9999   61419.0000   61419.0000      6.79%    6.79%      3      1
         3   65889.9999   61451.0000   61451.0000      6.74%    6.74%      5      1
         6   65889.9999   61524.0000   61524.0000      6.63%    6.63%     12      4
        10   65889.9999   61814.0000   61524.0000      6.19%    6.63%     21      7
         .   65889.9999   61995.0000   61561.0000      5.91%    6.57%     21      7
        11   65705.2771   61995.0000   61561.0000      5.65%    6.31%     23      7
        13   63865.3250   62218.3572   61561.0000      2.58%    3.61%     28      9
        14   63453.5848   62378.3334   61561.0000      1.69%    2.98%     31     10
        15   62892.5013   62419.1251   61561.0000      0.75%    2.12%     34     11
        16   62892.5013   62487.0001   62487.0001      0.64%    0.64%     37     12
        17   62892.5013   62494.0004   62494.0004      0.63%    0.63%     39     12
        18   62710.3361   62521.0015   62494.0004      0.30%    0.34%     42     13
        19   62636.0029   62525.0015   62494.0004      0.18%    0.23%     44     14
        20   62620.3345   62527.3348   62494.0004      0.15%    0.20%     47     15
         .   62620.3345   62528.3338   62519.0004      0.15%    0.16%     48     15
        21   62579.3348   62528.3338   62519.0004      0.08%    0.10%     51     16
        22   62534.3344   62528.3338   62519.0004      0.01%    0.02%     54     17
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
NOTE: Starting branch and bound.
       Node   Active   Sols         Best         Best       Gap     CPU   Real
                               Integer        Bound              Time   Time
          0        1      8   62519.0004   62534.3344     0.02%      54     17
          2        0      9   62524.0005   62527.5019     0.01%     102     33
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 33.22 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 62524.000477.
```

By using this hybrid method, you can take advantage of the direct method, which finds a good feasible solution quickly, and the strong bounds provided by the decomposition algorithm. The overall time to solve the model by using the hybrid method is faster than either of the other two.

The Tradeoff between Coverage and Subproblem Difficulty

The reformulation of this resource allocation problem provides a nice example of the potential tradeoffs in modeling a problem for use with the decomposition algorithm. As seen in Example 13.2, the strength of the bound is an important factor in the overall performance of the algorithm, but it is not always correlated to the magnitude of the subproblem coverage. In this example, the block size determines the number of blocks. Moreover, it determines the number of linking variables that are needed in the reformulation. At one extreme, if the block size is set to be $|S|$, then the number of blocks is 1, and the number of copies of original variables is 0. Using one block would be equivalent to the original staircase formulation and would not yield a model conducive to decomposition. As the number of blocks is increased, the number of linking variables increases (the size of the master problem), the strength of the decomposition bound decreases, and the difficulty of solving the subproblems decreases. In addition, as the number of blocks and their relative difficulty change, the efficient utilization of your machine's parallel architecture can be affected.

The previous section used a block size of 40. The following statement calls the decomposition algorithm and uses a block size of 130:

```
%ResourceAllocation_Decomp(task_data=TaskData, capacity=100, block_size=130);
```

The solution summary is displayed in Output 13.5.7.

Output 13.5.7 Solution Summary

```
                       The OPTMODEL Procedure

                          Solution Summary

           Solver                               MILP
           Algorithm                   Decomposition
           Objective Function             TotalProfit
           Solution Status       Conditionally Optimal
           Objective Value              62524.000312

           Relative Gap                 0.0000746713
           Absolute Gap                 4.6690962869
           Primal Infeasibility         9.5560876E-6
           Bound Infeasibility          1.776357E-15
           Integer Infeasibility        9.5560876E-6

           Best Bound                   62528.669409
           Nodes                                   1
           Iterations                             22
           Presolve Time                        2.60
           Solution Time                       27.04
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.5.8.

Output 13.5.8 Log

```
NOTE: There were 2697 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2877 variables (0 free, 0 fixed).
NOTE: The problem has 2877 binary and 0 integer variables.
NOTE: The problem has 2868 linear constraints (2688 LE, 180 EQ, 0 GE, 0 range).
NOTE: The problem has 27240 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 2877 variables, 2868 constraints, and 27240
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 21 disjoint blocks.
NOTE: The decomposition subproblems cover 2877 (100.00%) variables and 2688 (93.72%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter         Best         Master          Best         LP         IP   CPU   Real
                   Bound     Objective       Integer        Gap        Gap  Time   Time
NOTE: Starting phase 1.
         1       0.0000        0.0000             .       0.00%         .     2      0
NOTE: Starting phase 2.
         .   63337.0002   51369.0000    51369.0000      18.90%    18.90%     2      0
         3   63337.0002   51989.0000    51989.0000      17.92%    17.92%     8      2
        10   63337.0002   61546.0016    51989.0000       2.83%    17.92%    29     10
         .   63337.0002   61824.1123    60178.9998       2.39%     4.99%    29     10
        13   63124.2360   62171.2306    60178.9998       1.51%     4.67%    38     13
        14   63106.9999   62354.0001    60178.9998       1.19%     4.64%    40     14
        15   62791.3011   62392.5003    60178.9998       0.64%     4.16%    43     15
        16   62676.0007   62491.0002    60178.9998       0.30%     3.98%    46     16
        17   62596.0020   62524.0003    62524.0003       0.12%     0.12%    49     17
        18   62568.5024   62524.0003    62524.0003       0.07%     0.07%    52     18
        19   62568.5024   62524.0003    62524.0003       0.07%     0.07%    54     19
        20   62531.0027   62524.0003    62524.0003       0.01%     0.01%    57     20
         .   62531.0027   62524.0003    62524.0003       0.01%     0.01%    57     20
        22   62528.6694   62524.0003    62524.0003       0.01%     0.01%    63     22
NOTE: The Decomposition algorithm stopped on the integer RELOBJGAP= option.
         Node   Active   Sols         Best           Best        Gap      CPU    Real
                                   Integer          Bound                Time    Time
            0        0     11   62524.0003    62528.6694      0.01%       63      22
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 22.57 seconds.
NOTE: Conditional optimal.
NOTE: Objective = 62524.000312.
```

This version of the model provides a stronger bound and yields better overall performance.

Example 13.6: ATM Cash Management in Single-Machine Mode

This example describes an optimization model that is used in the management of cash flow for a bank's automated teller machine (ATM) network. The goal of the model is to determine a replenishment schedule for the bank to use in allocating cash inventory at its branches when servicing a preassigned subset of ATMs. Given a history of withdrawals per day for each ATM, the bank can use SAS forecasting tools to predict the expected cash need. The modeling of this prediction depends on various seasonal factors, including the days of the week, weeks of the month, holidays, typical salary disbursement days, location of the ATMs, and other demographic data. The prediction is a parametric mixture of models whose parameters depend on each ATM.

The optimization model performs a polynomial regression that minimizes the error (measured by the L_1 norm) between the predicted and actual withdrawals. The parameter settings in the regression determine the replenishment policy. The amount of cash that is allocated to each day is subject to a budget constraint. In addition, a constraint for each ATM limits the number of days that a *cash-out* (a situation in which the cash flow is less than the predicted withdrawal) can occur. The goal is to determine a policy for cash distribution that balances the predicted inventory levels while satisfying the budget and cash-out constraints. By keeping too much cash on hand for ATM fulfillment, the bank loses an investment opportunity. Moreover, regulatory agencies in many countries enforce a minimum cash reserve ratio at branch banks; according to regulatory policy, the cash in ATMs or in transit does not contribute toward this threshold.

Mixed Integer Nonlinear Programming Formulation

The most natural formulation for this model is in the form of a mixed integer nonlinear program (MINLP). Let A denote the set of ATMs and D denote the set of days that are used in the training data. The predictive model fit is defined by the following data for each ATM a on each day d: $c_{ad}, c_{ad}^x, c_{ad}^y, c_{ad}^z$, and c_{ad}^u. The model-fitting parameters define the variables (x_a, y_a, u_a) for each ATM that, when applied to the predictive model, estimate the necessary cash flow per day per ATM. In addition, define a surrogate variable f_{ad} for each ATM on each day that defines the cash inventory (replenished from the branch) minus withdrawals. The variable f_{ad} also represents the error in the regression model. Let B_d define the budget per day, K_a define the limit on cash-outs per ATM, and w_{ad} define the historical withdrawals at a particular ATM on a particular day. Then the following MINLP models this problem:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{a \in A} \sum_{d \in D} |f_{ad}| \\
\text{subject to} \quad & c_{ad}^x x_a + c_{ad}^y y_a + c_{ad}^z x_a y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad} & a \in A,\ d \in D & \quad \text{(cash)} \\
& \sum_{a \in A} (f_{ad} + w_{ad}) \leq B_d & d \in D & \quad \text{(budget)} \\
& |\{d \in D \mid f_{ad} < 0\}| \leq K_a & a \in A & \quad \text{(count)} \\
& x_a, y_a \in [0, 1] & a \in A & \\
& u_a \geq 0 & a \in A & \\
& f_{ad} \geq -w_{ad} & a \in A,\ d \in D &
\end{aligned}
$$

The cash constraint defines the surrogate variable f_{ad}, which gives the estimated net cash flow. The budget and count inequalities ensure that the solution satisfies the budget and cash-out constraints, respectively.

To express this model in a more standard form, you can first use some standard model reformulations to linearize the absolute value and the cash-out constraint (count).

Linearization of Absolute Value

A well-known reformulation for linearizing the absolute value of a variable is to introduce one variable for each side of the absolute value. The following systems are equivalent:

$$
\begin{array}{l}
\text{minimize} \quad |y| \\
\text{subject to} \quad Ay \ \le \ b
\end{array}
\qquad \text{is equivalent to} \qquad
\begin{array}{ll}
\text{minimize} \quad y^+ + y^- \\
\text{subject to} \quad A(y^+ - y^-) \ \le \ b \\
\qquad\qquad\quad\ \ y^+, y^- \qquad\ \ge \ 0
\end{array}
$$

Let f_{ad}^+ and f_{ad}^- represent the positive and negative parts, respectively, of the net cash flow f_{ad}. Then you can rewrite the model, removing the absolute value, as the following:

$$
\begin{array}{ll}
\text{minimize} & \displaystyle\sum_{a \in A}\sum_{d \in D} (f_{ad}^+ + f_{ad}^-) \\[2em]
\text{subject to} & c_{ad}^x x_a + c_{ad}^y y_a + c_{ad}^z x_a y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- \qquad a \in A,\ d \in D \\[1em]
& \displaystyle\sum_{a \in A} (f_{ad}^+ - f_{ad}^- + w_{ad}) \le B_d \qquad\qquad\qquad d \in D \\[1em]
& |\{d \in D \mid (f_{ad}^+ - f_{ad}^-) < 0\}| \le K_a \qquad\qquad a \in A \\[0.5em]
& x_a, y_a \in [0, 1] \qquad\qquad\qquad\qquad\qquad a \in A \\[0.5em]
& u_a \ge 0 \qquad\qquad\qquad\qquad\qquad\qquad\ a \in A \\[0.5em]
& f_{ad}^+ \ge 0 \qquad\qquad\qquad\qquad\qquad\quad\ a \in A,\ d \in D \\[0.5em]
& f_{ad}^- \in [0, w_{ad}] \qquad\qquad\qquad\qquad\ \ a \in A,\ d \in D
\end{array}
$$

Modeling the Cash-Out Constraints

To count the number of times a cash-out occurs, you need to introduce a binary variable to keep track of when this event occurs. Let v_{ad} be an indicator variable that takes the value 1 when the net cash flow is negative. You can model the implication $f_{ad}^- > 0 \Rightarrow v_{ad} = 1$, or its contrapositive $v_{ad} = 0 \Rightarrow f_{ad}^- \le 0$, by adding the constraint

$$
f_{ad}^- \le w_{ad} v_{ad} \quad a \in A,\ d \in D
$$

Now you can model the cash-out constraint by counting the number of days that the net-cash flow is negative for each ATM, as follows:

$$
\sum_{d \in D} v_{ad} \le K_a \quad a \in A
$$

The MINLP model can now be written as follows:

minimize
$$\sum_{a \in A} \sum_{d \in D} \left(f_{ad}^+ + f_{ad}^- \right)$$

subject to
$$c_{ad}^x x_a + c_{ad}^y y_a + c_{ad}^z x_a y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- \qquad a \in A, \ d \in D$$

$$\sum_{a \in A} \left(f_{ad}^+ - f_{ad}^- + w_{ad} \right) \le B_d \qquad d \in D$$

$$f_{ad}^- \le w_{ad} v_{ad} \qquad a \in A, \ d \in D$$

$$\sum_{d \in D} v_{ad} \le K_a \qquad a \in A$$

$$x_a, y_a \in [0, 1] \qquad a \in A$$

$$u_a \ge 0 \qquad a \in A$$

$$f_{ad}^+ \ge 0 \qquad a \in A, \ d \in D$$

$$f_{ad}^- \in [0, w_{ad}] \qquad a \in A, \ d \in D$$

$$v_{ad} \in \{0, 1\} \qquad a \in A, \ d \in D$$

This MINLP is difficult to solve, in part because the prediction function is not convex. Another approach is to use mixed integer linear programming (MILP) to formulate an approximation of the problem, as described in the next section.

Mixed Integer Linear Programming Approximation

Because the predictive model is a forecast, finding the optimal parameters that are based on nondeterministic data is not of primary importance. Rather, you want to provide as good a solution as possible in a reasonable amount of time. So using MILP to approximate the MINLP is perfectly acceptable. In the original problem you have products of two continuous variables that are both bounded by 0 (lower bound) and 1 (upper bound). This arrangement enables you to create an approximate linear model by using a few standard modeling reformulations.

Discretization of Continuous Variables
The first step is to discretize one of the continuous variables x_a. The goal is to transform the product $x_a y_a$ of a continuous variable and another continuous variable instead to the product of a continuous variable and a binary variable. This transformation enables you to linearize the product form.

You must assume some level of approximation by defining a binary variable (from some discrete set) for each possible setting of the continuous variable For example, if you let $n = 10$, then you allow x to be chosen from the set $\{0.0, 0.1, 0.2, 0.3, ..., 1.0\}$. Let $T = \{0, 1, 2, ..., n\}$ represent the possible steps and $c_t = t/n$. Then you apply the following transformation to variable x_a:

$$\sum_{t \in T} c_t x_{at} = x_a$$

$$\sum_{t \in T} x_{at} = 1$$

$$x_{at} \in \{0, 1\} \quad t \in T$$

The MINLP model can now be approximated as the following:

$$\text{minimize} \quad \sum_{a \in A} \sum_{d \in D} \left(f_{ad}^+ + f_{ad}^- \right)$$

$$\text{subject to} \quad c_{ad}^x \sum_{t \in T} c_t x_{at} + c_{ad}^y y_a +$$

$$c_{ad}^z \sum_{t \in T} c_t x_{at} y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- \qquad a \in A, \ d \in D$$

$$\sum_{t \in T} x_{at} = 1 \qquad a \in A$$

$$\sum_{a \in A} \left(f_{ad}^+ - f_{ad}^- + w_{ad} \right) \le B_d \qquad d \in D$$

$$f_{ad}^- \le w_{ad} v_{ad} \qquad a \in A, \ d \in D$$

$$\sum_{d \in D} v_{ad} \le K_a \qquad a \in A$$

$$y_a \in [0, 1] \qquad a \in A$$

$$u_a \ge 0 \qquad a \in A$$

$$f_{ad}^+ \ge 0 \qquad a \in A, \ d \in D$$

$$f_{ad}^- \in [0, w_{ad}] \qquad a \in A, \ d \in D$$

$$v_{ad} \in \{0, 1\} \qquad a \in A, \ d \in D$$

$$x_{at} \in \{0, 1\} \qquad a \in A, \ t \in T$$

Linearization of Products

You still need to linearize the product terms $x_{at} y_a$ in the cash flow constraint. Because these terms are products of a bounded continuous variable and a binary variable, you can linearize them by introducing for each product another variable, z_{at}, which serves as a surrogate. In general, you know the following relationship between the original variables and their surrogates:

$$
\begin{aligned}
z_t &= x_t y & t \in T \\
\sum_{t \in T} x_t &= 1 \\
x_t &\in \{0, 1\} & t \in T \\
y &\in [0, 1]
\end{aligned}
\qquad \text{is equivalent to} \qquad
\begin{aligned}
z_t &\ge 0 & t \in T \\
z_t &\le x_t & t \in T \\
\sum_{t \in T} x_t &= 1 \\
\sum_{t \in T} z_t &= y \\
x_t &\in \{0, 1\} & t \in T \\
y &\in [0, 1]
\end{aligned}
$$

Using this relationship to replace each product form, you now can write the problem as an approximate MILP as follows:

minimize

$$\sum_{a \in A} \sum_{d \in D} \left(f_{ad}^+ + f_{ad}^- \right)$$

subject to

$$c_{ad}^x \sum_{t \in T} c_t x_{at} + c_{ad}^y y_a +$$

$$c_{ad}^z \sum_{t \in T} c_t z_{at} + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- \qquad a \in A, \ d \in D$$

$$\sum_{t \in T} x_{at} = 1 \qquad a \in A$$

$$\sum_{a \in A} \left(f_{ad}^+ - f_{ad}^- + w_{ad} \right) \leq B_d \qquad d \in D \qquad \text{(budget)}$$

$$f_{ad}^- \leq w_{ad} v_{ad} \qquad a \in A, \ d \in D$$

$$\sum_{d \in D} v_{ad} \leq K_a \qquad a \in A$$

$$z_{at} \leq x_{at} \qquad a \in A, \ t \in T$$

$$\sum_{t \in T} z_{at} = y_a \qquad a \in A$$

$$z_{at} \geq 0 \qquad a \in A, \ t \in T$$

$$y_a \in [0, 1] \qquad a \in A$$

$$u_a \geq 0 \qquad a \in A$$

$$f_{ad}^+ \geq 0 \qquad a \in A, \ d \in D$$

$$f_{ad}^- \in [0, w_{ad}] \qquad a \in A, \ d \in D$$

$$v_{ad} \in \{0, 1\} \qquad a \in A, \ d \in D$$

$$x_{at} \in \{0, 1\} \qquad a \in A, \ t \in T$$

PROC OPTMODEL Code

Because it is difficult to solve the MINLP model directly, the approximate MILP formulation is attractive. Unfortunately, the approximate MILP is much larger than the associated MINLP. Direct methods for solving this MILP do not work well. However, the problem is nicely suited for the decomposition algorithm.

When you examine the structure of the MILP model, you see clearly that the constraints can be easily decomposed by ATM. In fact, the only set of constraints that involve decision variables across ATMs is the budget constraint (budget). That is, if you relax the budget constraint, you are left with independent blocks of constraints, one for each ATM.

To show how this is done in PROC OPTMODEL, consider the following data sets, which describe an example that tracks 20 ATMs over a period of 100 days. This particular example was submitted to MIPLIB 2010, which is a collection of difficult MILPs in the public domain (Koch et al. 2011).

The first data set, budget_data, provides the cash budget on each particular day:

```
data budget_data;
   input d $ budget;
   datalines;
DATE0      70079
DATE1      66418
DATE10     52656
DATE11     50439
DATE12     58688
DATE13     45002
DATE14     52369
...
;
```

The second data set, cashout_data, provides the limit on the number of cash-outs that are allowed at each ATM:

```
data cashout_data;
   input a $ cashOutLimit;
   datalines;
ATM0            31
ATM1            24
ATM2            41
ATM3            43
ATM4            29
ATM5            24
ATM6            52
ATM7            44
ATM8            35
ATM9            48
ATM10           31
ATM11           47
ATM12           26
ATM13           34
ATM14           29
ATM15           32
ATM16           33
ATM17           32
ATM18           43
ATM19           28
;
```

The final data set, polyfit_data, provides the polynomial fit coefficients for each ATM on each date. It also provides the historical cash withdrawals.

```
data polyfit_data;
   input a $ d $ cx cy cz cu c withdrawal;
   datalines;
ATM0      DATE0       2822      1984     -1984     1045     1373      780
ATM0      DATE1       1337      2530     -2530     1510      174     2351
ATM0      DATE2       2685       -67        67      145     2820     2288
ATM0      DATE3       -595     -3135      3135      581     3319     1357
...
ATM19     DATE96      -734      3392     -3392      162     1648      914
ATM19     DATE97     -1062       969      -969      444     1746     2264
```

```
    ATM19    DATE98    7676    2308    -2308      59    1388        972
    ATM19    DATE99    3062    1308    -1308    1080     654        698
    ;
```

The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```
proc optmodel;
   set<str> DATES;
   set<str> ATMS;

   /* cash budget per date */
   num budget{DATES};

   /* maximum number of cash-outs allowed at each atm */
   num cashOutLimit{ATMS};

   /* historical withdrawal amount per atm each date */
   num withdrawal{ATMS, DATES};

   /* polynomial fit coefficients for predicted cash flow needed */
   num c {ATMS, DATES};
   num cx{ATMS, DATES};
   num cy{ATMS, DATES};
   num cz{ATMS, DATES};
   num cu{ATMS, DATES};

   /* number of points used in approximation of continuous range */
   num nSteps = 10;
   set STEPS = {0..nSteps};

   read data budget_data into DATES=[d] budget;
   read data cashout_data into ATMS=[a] cashOutLimit;
   read data polyfit_data into [a d] cx cy cz cu c withdrawal;
```

The following statements declare the variables:

```
   var x{ATMS,STEPS}                  binary;
   var v{ATMS,DATES}                  binary;
   var z{ATMS,STEPS}                  >= 0 <= 1;
   var y{ATMS}                        >= 0 <= 1;
   var u{ATMS}                        >= 0;
   var fPlus{ATMS,DATES}              >= 0;
   var fMinus{a in ATMS, d in DATES} >= 0 <= withdrawal[a,d];
```

The following statements declare the objective and the constraints:

```
   min CashFlowDiff =
      sum{a in ATMS, d in DATES} (fPlus[a,d] + fMinus[a,d]);

   con BudgetCon{d in DATES}:
      sum{a in ATMS} (fPlus[a,d] - fMinus[a,d] + withdrawal[a,d])
           <= budget[d];

   con CashFlowDefCon{a in ATMS, d in DATES}:
      cx[a,d] * sum{t in STEPS} (t/nSteps) * x[a,t] +
```

```
      cy[a,d] * y[a]                                       +
      cz[a,d] * sum{t in STEPS} (t/nSteps) * z[a,t] +
      cu[a,d] * u[a]                                       +
      c[a,d] - withdrawal[a,d] = fPlus[a,d] - fMinus[a,d];

   con PickOneStepCon{a in ATMS}:
      sum{t in STEPS} x[a,t] = 1;

   con CashOutLinkCon{a in ATMS, d in DATES}:
      fMinus[a,d] <= withdrawal[a,d] * v[a,d];

   con CashOutLimitCon{a in ATMS}:
      sum{d in DATES} v[a,d] <= cashOutLimit[a];

   con Linear1Con{a in ATMS, t in STEPS}:
      z[a,t] <= x[a,t];

   con Linear2Con{a in ATMS}:
      sum{t in STEPS} z[a,t] = y[a];
```

The following statements define the block decomposition by ATM. The **.block** suffix expects numeric indices, whereas the **SET<STR> ATMS** statement declares a set of strings. You can create a mapping from the string identifier to a numeric identifier as follows:

```
   /* create numeric block index */
   num blockIndex {ATMS};
   num index init 0;
   for{a in ATMS} do;
      blockIndex[a] = index;
      index = index + 1;
   end;
```

Then, each constraint can be added to its associated ATM block as follows:

```
   /* define blocks for each ATM */
   for{a in ATMS} do;
      PickOneStepCon[a].block  = blockIndex[a];
      CashOutLimitCon[a].block = blockIndex[a];
      Linear2Con[a].block      = blockIndex[a];
      for{d in DATES} do;
         CashFlowDefCon[a,d].block = blockIndex[a];
         CashOutLinkCon[a,d].block = blockIndex[a];
      end;
      for{t in STEPS} do;
         Linear1Con[a,t].block = blockIndex[a];
      end;
   end;
```

The budget constraint links all the ATMs, and it remains in the master problem. Finally, the following statements use DECOMP to solve the problem:

```
      /* set the number of threads and get performance details */
      performance details nthreads=4;

      /* solve with the decomposition algorithm */
      solve with milp / decomp=();
   quit;
```

The solution summary, performance information, and procedure task timing tables are displayed in Output 13.6.1.

Output 13.6.1 Performance Information, Solution Summary, and Task Timing Tables

```
                         The OPTMODEL Procedure

                        Performance Information

                 Execution Mode       Single-Machine
                 Number of Threads              4


                          Solution Summary

          Solver                                     MILP
          Algorithm                         Decomposition
          Objective Function                  CashFlowDiff
          Solution Status     Optimal within Relative Gap
          Objective Value                    2463627.0036

          Relative Gap                        0.0000311469
          Absolute Gap                        76.731987027
          Primal Infeasibility                4.329195E-10
          Bound Infeasibility                            0
          Integer Infeasibility               1.356693E-13

          Best Bound                         2463550.2716
          Nodes                                        32
          Iterations                                   40
          Presolve Time                              0.50
          Solution Time                            588.18


                      Procedure Task Timing
                                    Time
            Task                   (sec.)      % Time

            Problem Generation       0.12       0.02%
            Solver Initialization    0.08       0.01%
            Code Generation          0.00       0.00%
            Solver                 588.19      99.96%
            Solver Postprocessing    0.02       0.00%
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.6.2.

Output 13.6.2 Log

```
NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 502 variables and 345 constraints.
NOTE: The MILP presolver removed 1172 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5978 variables, 4035 constraints, and 57706
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 20 disjoint blocks.
NOTE: The decomposition subproblems cover 5978 (100.00%) variables and 3935 (97.52%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best          Master          Best         LP        IP    CPU   Real
                    Bound        Objective       Integer       Gap       Gap  Time   Time
NOTE: Starting phase 1.
         1         0.0000         1.1767              . 1.18e+00         .   136    47
         2         0.0000         0.0000              .    0.00%         .   136    47
NOTE: Starting phase 2.
         .     2.4432e+06     2.6998e+06     2.7643e+06    10.51%    13.14%   136    47
         4     2.4505e+06     2.4981e+06     2.7643e+06     1.94%    12.81%   242    80
         5     2.4631e+06     2.4656e+06     2.7643e+06     0.10%    12.23%   288    96
         6     2.4631e+06     2.4631e+06     2.7643e+06     0.00%    12.23%   331   111
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
         .     2.4631e+06     2.4631e+06     2.4648e+06     0.00%     0.07%   331   111
NOTE: Starting branch and bound.
         Node   Active   Sols         Best           Best       Gap      CPU   Real
                                   Integer         Bound              Time   Time
            0        1      2    2.4648e+06     2.4631e+06     0.07%     331   111
           10       12      2    2.4648e+06     2.4635e+06     0.05%     863   292
           11        9      3    2.4638e+06     2.4635e+06     0.01%     911   305
           20        8      3    2.4638e+06     2.4636e+06     0.01%    1285   422
           30        8      3    2.4638e+06     2.4636e+06     0.01%    1744   569
           31        5      4    2.4636e+06     2.4636e+06     0.00%    1780   580
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 580.05 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2463627.0036.
```

Example 13.7: ATM Cash Management in Distributed Mode

This section illustrates how you can use PROC OPTMODEL and the decomposition algorithm in distributed mode. The problem is the same as the one described in Example 13.6 for managing the cash flow of an ATM network. The only difference between single-machine and distributed mode is that the PERFORMANCE statement specifies the number of threads to be used in single-machine mode or the number of threads and nodes to be used in distributed mode.

The following statement changes the operating mode to distributed mode:

```
/* set the number of nodes and threads and get performance details */
performance details nodes=5 nthreads=4;
```

The solution summary, performance information, and procedure task timing tables are displayed in Output 13.7.1. When you specify NODES=5 and NTHREADS=4 in the PERFORMANCE statement in distributed mode, each grid node processes up to four threads simultaneously.

Output 13.7.1 Performance Information, Solution Summary, and Task Timing Tables

```
                        The OPTMODEL Procedure

                        Performance Information

        Host Node                            <<your grid host>>
        Execution Mode                             Distributed
        Grid Mode                                    Symmetric
        Number of Compute Nodes                              5
        Number of Threads per Node                           4



                          Solution Summary

         Solver                                         MILP
         Algorithm                             Decomposition
         Objective Function                      CashFlowDiff
         Solution Status         Optimal within Relative Gap
         Objective Value                         2463756.319

         Relative Gap                           0.0000856188
         Absolute Gap                            210.92580152
         Primal Infeasibility                    1.1759766E-9
         Bound Infeasibility                    2.220446E-16
         Integer Infeasibility                  1.325606E-13

         Best Bound                             2463545.3932
         Nodes                                            20
         Iterations                                       29
         Presolve Time                                  0.43
         Solution Time                                167.69
```

Output 13.7.1 *continued*

```
            Procedure Task Timing
                              Time
    Task                     (sec.)      % Time

    Problem Generation        0.11       0.06%
    Solver Initialization     0.07       0.04%
    Code Generation           0.00       0.00%
    Solver                  167.71      99.81%
    Solver Postprocessing     0.14       0.08%
```

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 13.7.2.

Output 13.7.2 Log

```
NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 502 variables and 345 constraints.
NOTE: The MILP presolver removed 1172 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5978 variables, 4035 constraints, and 57706
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing
      environment with 5 worker nodes.
NOTE: The DECOMP method value USER is applied.
NOTE: The decomposition subproblems consist of 20 disjoint blocks.
NOTE: The decomposition subproblems cover 5978 (100.00%) variables and 3935 (97.52%)
      constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best         Master          Best       LP       IP  Real
                   Bound      Objective       Integer      Gap      Gap  Time
NOTE: Starting phase 1.
         1       0.0000         1.1767            . 1.18e+00         .    20
         2       0.0000         0.0000            .    0.00%         .    20
NOTE: Starting phase 2.
         .    2.4432e+06     2.6947e+06     2.7543e+06   10.30%   12.73%   20
         4    2.4526e+06     2.4869e+06     2.7543e+06    1.40%   12.30%   31
         5    2.4630e+06     2.4642e+06     2.7543e+06    0.05%   11.83%   37
         6    2.4631e+06     2.4632e+06     2.7543e+06    0.00%   11.82%   43
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
         .    2.4631e+06     2.4632e+06     2.4660e+06    0.00%    0.12%   43
NOTE: Starting branch and bound.
        Node   Active   Sols         Best           Best      Gap    Real
                                  Integer          Bound              Time
           0        1      2    2.4660e+06     2.4631e+06    0.12%     43
          10       10      3    2.4654e+06     2.4635e+06    0.07%    111
          19        7      4    2.4638e+06     2.4635e+06    0.01%    163
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 163.94 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2463756.319.
```

Notice how this iteration log differs from the iteration log from single-machine mode in Example 13.6. In distributed mode, the processing is done on multiple grid machines, as opposed to being done on one client machine in single-machine mode. In this example, the grid machines and the client machine have different operating systems, and some numerical rounding off leads to different paths in the search space. When you compare two runs on different operating systems (or that use different compilers), this behavior is expected.

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993), *Network Flows: Theory, Algorithms, and Applications*, Englewood Cliffs, NJ: Prentice-Hall.

Barnhart, C., Hane, C. A., and Vance, P. H. (2000), "Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems," *Operations Research*, 48, 318–326.
URL http://www.jstor.org/stable/223148

Caprara, A., Furini, F., and Malaguti, E. (2010), *Exact Algorithms for the Temporal Knapsack Problem*, Technical Report OR-10-7, University of Bologna, Department of Electronics, Computer Science, and Systems.

Dantzig, G. B. and Wolfe, P. (1960), "Decomposition Principle for Linear Programs," *Operations Research*, 8, 101–111.
URL http://www.jstor.org/stable/167547

Galati, M. V. (2009), *Decomposition in Integer Linear Programming*, Ph.D. diss., Lehigh University.

Gamrath, G. (2010), *Generic Branch-Cut-and-Price*, diploma thesis, Technische Universität Berlin.

Koch, T., Achterberg, T., Andersen, E. D., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D. E., and Wolter, K. (2011), "MIPLIB 2010," *Mathematical Programming Computation*, 3, 103–163.
URL http://mpc.zib.de/index.php/MPC/article/view/56/28

Ralphs, T. K. and Galati, M. V. (2006), "Decomposition and Dynamic Cut Generation in Integer Linear Programming," *Mathematical Programming*, 106, 261–285.
URL http://dx.doi.org/10.1007/s10107-005-0606-3

Vanderbeck, F. and Savelsbergh, M. W. P. (2006), "A Generic View of Dantzig-Wolfe Decomposition in Mixed Integer Programming," *Operations Research Letters*, 34, 296–306.
URL http://dx.doi.org/10.1016/j.orl.2005.05.009

Chapter 14
The OPTMILP Option Tuner

Contents

Overview: The OPTMILP Option Tuner

The OPTMILP procedure provides many solver techniques and algorithms, including branch-and-bound, cutting planes, and heuristics. It also provides control options that you can adjust to improve the performance of these techniques. Although the default values of the control options have been tuned to work well for most instances, you might need to adjust one or more option values for a specific problem. The OPTMILP option tuner is a tool that enables you to explore alternative (and potentially better) option configurations for your optimization problems.

To use the tuner, you specify a single problem or set of problems to be solved and a list of options to be tuned. You can specify initial values for the options to be tuned. The tuner then uses a heuristic local search technique to generate a sequence of configurations. A *configuration* is a set of the specified options to be tuned along with a fixed value for each option. The tuner attempts to locate configurations that enable the OPTMILP procedure to process problems more quickly than the default option values or specified initial values.

The tuner option can be run in either single-machine mode or distributed mode. In single-machine mode, you can specify the number of threads to use on a single computer. In distributed mode, you can specify the number of computer nodes and the number of threads per node to use on a distributed computing environment. The tuner option only supports nondeterministic mode, so the tuning results may vary among different runs.

NOTE: Distributed mode requires SAS High-Performance Optimization software.

Getting Started: The OPTMILP Option Tuner

This example illustrates how to use the OPTMILP option tuner.

The standard set of MILP benchmark cases is called MIPLIB (Bixby et al. 1998; Achterberg, Koch, and Martin 2003) and can be found at http://miplib.zib.de/. Suppose you want to solve the problems air04 and air05 from this set. You have stored the SAS data sets *air04* and *air05*, both in MPS format, in library a. Suppose you want to tune the CUTCLIQUE=, CUTGOMORY=, and HEURISTICS options in these two problems.

The following DATA step generates the data set probs, which contains the list of problems to be solved, and the data set optvals, which contains the list of options to be tuned:

```
data probs;
   input name $1-8;
   datalines;
a.air04
a.air05
;

data optvals;
   input option $1-10;
   datalines;
cutclique
cutgomory
heuristics
;
```

The following statements call the OPTMILP procedure and enable the option tuner:

```
proc optmilp maxtime=300;
   tuner maxtime=1200 problems=probs optionvalues=optvals tunerout=out;
   performance nthreads=4;
run;
```

The MAXTIME= option in the PROC OPTMILP statement sets the maximum run time that the procedure can use to solve one problem for each option configuration. The MAXTIME= option in the TUNER statement sets a limit on the total time that the option tuner can use to solve the problems on the list by using the generated sequence of configurations. The PROBLEMS= option specifies the name of the SAS data set that contains the list of problems to be solved. The OPTIONVALUES= option specifies the name of the SAS data set that contains the list of options to be tuned. The TUNEROUT= option specifies the name of the SAS data set that contains detailed results of the tuning process. The NTHREADS= option in the PERFORMANCE

statement specifies the number of threads that the procedure can use to perform calculations. The ODS OUTPUT statement creates an output data set from the TunerResults table.

For more information about the options available in the PROC OPTMILP statement, see the section "PROC OPTMILP Statement" on page 423. For more information about the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 28.

Figure 14.1 shows a selection of tuning results that include the initial option configuration, the best option configurations, and the worst option configurations.

Figure 14.1 PROC OPTMILP Output

```
                       The OPTMILP Procedure

                     Performance Information

        Execution Mode        Single-Machine
        Number of Threads                   4


                       Tuner Information

        Target Solver                     MILP
        Number of Tuning Options             3
        Number of Tuning Instances           2
        Tuning Option Set                 USER
        Performance Goal               GEOMEAN
        Tuner Time Limit                  1200
        Tuner Configurations Limit  2147483647


                         Tuner Summary

   Actual Tuning Time                      1254.35
   Initial Run Time (geomean)                86.11
   Initial Run Time (sum)                   176.16
   Best Run Time (geomean)                   72.78
   Best Run Time (sum)                      147.99
   Number of Improved Configurations             4
   Number of Tested Configurations              24
```

Figure 14.1 *continued*

```
                              Tuner Results

                     C        C        C        C        C        C        C
                     o        o        o        o        o        o        o
                     n        n        n        n        n        n        n
                     f        f        f        f        f        f        f
                     i        i        i        i        i        i        i
                     g        g        g        g        g        g        g

                     0        1        2        3        4        5        6

cutclique           -1       -1        2       -1       -1       -1       -1
cutgomory           -1       -1        1        0        1        1        1
heuristics          -1        2        2        2        0        2        3
Mean of Run Times 86.11    72.77    76.72    79.96    82.16    89.96   108.19
Sum of Run Times 176.16   147.99   156.56   162.85   179.53   183.26   236.56
Percentage Successful 100     100      100      100      100      100      100

                              Tuner Results

                                                                        C
                              C        C        C                       o
                              o        o        o                       n
                              n        n        n                       f
                              f        f        f                       i
                              i        i        i                       g
                              g        g        g
                                                                        1
                              7        8        9                       0

        cutclique             1        0        2                      -1
        cutgomory             0        1       -1                       2
        heuristics            1        3        3                       3
        Mean of Run Times  108.79   111.71    93.49                    62.7
        Sum of Run Times   224.39   240.38   195.28                   125.5
        Percentage Successful 100     100       50                       0
```

Syntax: The OPTMILP Option Tuner

You can specify the following statements for the option tuner in the OPTMILP procedure:

PROC OPTMILP < *options* > ;
    PERFORMANCE < *performance-options* > ;
    TUNER < *tuner-options* > ;

Functional Summary

Table 14.1 summarizes the options available for the TUNER statement in the OPTMILP procedure.

Table 14.1   Options for the TUNER Statement

Option	Description
GOAL=	Specifies the goal of the tuning process
LOGFREQ=	Specifies the frequency of printing in the log
LOGLEVEL=	Specifies the detail of tuner progress printed in the log
MAXCONFIGS=	Specifies the maximum number of tuning configurations
MAXTIME=	Specifies the maximum tuning time
OPTIONMODE=	Specifies which set of options to tune
OPTIONVALUES=	Specifies the input data set that contains a list of options to be tuned
PROBLEMS=	Specifies the input data set that contains a list of tuning problems
TUNEROUT=	Specifies the output data set that contains detailed tuning results

The options available for the PROC OPTMILP statement are documented in the section "Functional Summary" on page 421 in Chapter 11, "The OPTMILP Procedure." You must specify the MAXTIME= option in the PROC OPTMILP statement so that the tuner can terminate properly.

PERFORMANCE Statement

> **PERFORMANCE** < *performance-options* > **;**

The PERFORMANCE statement specifies performance options for single-machine mode and distributed mode, passes variables that describe the distributed computing environment, and requests detailed performance results of the OPTMILP procedure.

For single-machine mode, you can use the NTHREADS= option to specify the number of threads to use on a single machine. For distributed mode, you can use the NODES= and NTHREADS= options to specify the numbers of computer nodes and threads per node to use in a distributed computing environment.

When multiple threads are specified, several MILP solvers can run concurrently on a single machine or a computer node. You might consider reducing the value of the NTHREADS= option when the MILP solver returns an out-of-memory status for some tuning problems.

The DETAILS option displays a detailed performance "Timing" table. The OPTMILP option tuner supports only the nondeterministic mode of the PARALLELMODE= option in the PERFORMANCE statement.

The PERFORMANCE statement for single-machine and distributed mode is documented in the section "PERFORMANCE Statement" on page 28 in Chapter 4, "Shared Concepts and Topics."

NOTE: Distributed mode requires SAS High-Performance Optimization software.

TUNER Statement

> **TUNER** < *tuner-options* > **;**

You can specify the following options.

GOAL=_number | string_

    specifies a goal for the option tuner. Table 14.2 describes the valid values of the GOAL= option.

Table 14.2   Values of GOAL= Option

number	string	Description
0	GEOMEAN	Minimizes the geometric mean of the solution times of the tuning problems
1	SUM	Minimizes the sum of the solution times of the tuning problems

    Every attempt to solve a tuning problem that has an option configuration is counted toward the measure that is specified by the GOAL= option.

    The default is GEOMEAN. If only one problem is used for option tuning, then GOAL=GEOMEAN and GOAL=SUM are equivalent.

LOGFREQ=_number_

    specifies how often tuning information is printed in the log. The value of *number* represents the number of problems solved by the tuner between log updates. The value of *number* can be any nonnegative integer. Specifying LOGFREQ=0 disables log updates. The default is 1.

LOGLEVEL=_number | string_

    controls the amount of information that the tuner displays in the SAS log. Table 14.3 describes the valid values of the LOGLEVEL= option.

Table 14.3   Values of LOGLEVEL= Option

number	string	Description
0	NONE	Disables tuner-related messages in the SAS log
1	BASIC	Displays a tuner summary after stopping
2	MODERATE	Prints a tuner summary and a tuning log by using the interval dictated by the LOGFREQ= option

    The default is MODERATE.

MAXCONFIGS=_number_

    specifies the maximum number of option configurations that the tuner can evaluate in each problem in the PROBLEMS= data set. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default is $2^{31} - 1$. This option is an alternative way for the tuner to control the termination.

MAXTIME=*number*

> specifies the maximum time allowed for the tuner to evaluate option configurations in tuning problems. You must specify either this option or the MAXCONFIGS= option so that the tuner can terminate properly.
>
> It is recommended that you specify a value for *number* that is large enough that the tuner can run several different option configurations. This value depends on two quantities: the number of tuning problems and the OPTMILP procedure's average run time for the tuning problems. To prevent the procedure from spending too much time running a single configuration in a single problem, you must limit the time the procedure spends solving each combination of problem and configuration. You can limit the time by specifying the MAXTIME= option in the PROC OPTMILP statement. If you prefer not to stop the option tuner as a result of elapsed time, you can specify the MAXCONFIGS= option.

OPTIONMODE=*number | string*

> specifies which set of options to tune. Table 14.4 describes the valid values of the OPTIONMODE= option.

Table 14.4 Values of OPTIONMODE= Option

number	*string*	**Description**
−1	AUTOMATIC	Uses an option set that is determined by the tuner
0	NONE	Solves the problems that are specified by the PROBLEMS= option without tuning any OPTMILP options
1	USER	Uses the option set that is specified by the OPTIONVALUES= option
2	FULL	Uses the full set of solver options that are available for tuning

> The tuner interprets the OPTIONMODE= option in accordance with the following logic:
>
> 1. If you specify neither the OPTIONMODE= nor the OPTIONVALUES= option, the tuner runs with OPTIONMODE=AUTOMATIC.
>
> 2. If you specify the OPTIONVALUES= option, you are not required to specify the OPTION-MODE= option, but you can specify OPTIONMODE=USER. Specifying any other value for the OPTIONMODE= option causes the tuner to terminate with an error.
>
> 3. If you do not specify the OPTIONVALUES= option, you can specify either OPTION-MODE=FULL or OPTIONMODE=AUTOMATIC. Specifying OPTIONMODE=USER causes the tuner to terminate with an error.
>
> When OPTIONMODE=NONE, the options PRIMALOUT= and DUALOUT= of PROC OPTMILP can be used to output primal and dual solutions of all the problems that are listed in the PROBLEMS= data set. All primal or dual solutions are appended to the PRIMALOUT= or DUAL= data set. An additional variable _PROBLEM_ is created for each data set to store problem names.

OPTIONVALUES=*SAS-data-set*

OPTVALS=*SAS-data-set*

> specifies an input data set that contains a list of options to be tuned and ranges of values over which each option should be tuned. You can specify an initial tuning value for each option in the list. If you do not specify a range for a tuning option, the tuner uses the default range of that option. If you do not specify an initial value for a tuning option, the tuner uses the default value of that option. If the option's default value is not in the specified tuning range, the tuner uses the first (smallest) value in the tuning range. If the data set that is specified by the OPTIONVALUES= option is not found, a default set of options is used. For more information, see the section "Variables in the OPTIONVALUES= Data Set" on page 587 and the section "Default Set of Tuning Options" on page 588.

> **NOTE:** An option value that you specify in the PROC OPTMILP statement is applied to all tuning problems unless you specify that option in the OPTIONVALUES= data set. In that case, the value that you specify in the PROC OPTMILP statement is ignored.

PROBLEMS=*SAS-data-set*

PROBS=*SAS-data-set*

> specifies the input data set that contains a list of MILP problems to be used for option tuning. This list includes the name of each problem, its library location, and (optionally) its objective sense. For more information, see the section "Variables in the PROBLEMS= Data Set" on page 587. The tuning problems should be stored in MPS-format SAS data sets. To perform option tuning on a single problem, you can omit the PROBLEMS= option in the TUNER statement and specify the DATA= option in the PROC OPTMILP statement. For more information about this option, see the section "Data Set Options" on page 423.

TUNEROUT=*SAS-data-set*

TOUT=*SAS-data-set*

> specifies the output data set to contain detailed results for each tuning problem over all the option configurations that are evaluated. This data set helps you rank the performance with your own rules, especially when some problems are not optimal but the integer solutions are acceptable for some configurations. For more information, see the section "Variables in the TUNEROUT= Data Set" on page 587.

Details: The OPTMILP Option Tuner

Data Input and Output

This subsection describes the input data sets that are specified by the PROBLEMS= and OPTIONVALUES= options and the output data set that is specified by the TUNEROUT= option.

When you specify OPTIONMODE=NONE, you can specify the PRIMALOUT= and DUALOUT= options in the PROC OPTMILP statement to output primal and dual solutions of all the problems listed in the PROBLEMS= data set. All primal or dual solutions are appended to the PRIMALOUT= or DUALOUT= data set, respectively. An additional variable, _PROBLEM_, is created for each data set; this variable stores problem names.

Variables in the PROBLEMS= Data Set

The PROBLEMS= data set contains the following variables:

NAME

specifies a list of names of MPS-format data sets. Each data set contains a MILP problem to be used in option tuning. The format of each name must be *libref.filename*. If no libref is specified, the tuner searches for the file in WORK.

OBJSENSE

specifies whether the objective sense for a tuning problem is MIN or MAX. The values of this variable provide or overwrite the objective sense for the corresponding SAS data set. This variable is optional.

Variables in the OPTIONVALUES= Data Set

The OPTIONVALUES= data set contains the following variables:

INITIAL

specifies an initial value for each option to be tuned. This variable is optional. If this variable is missing, the tuner uses the default value of the option as the initial value. If the default value of the option is not in the list specified by the VALUES variable, the tuner uses the first entry in the VALUES list for that option.

OPTION

specifies a list of control options to be tuned by solving the problems specified in the PROBLEMS= data set.

VALUES

specifies a comma-delimited list of values for each control option that the tuner can use to generate configurations. If you do not specify a list of values for an option, the tuner uses all valid values of that option. This variable is optional. If you do not provide discrete values, such as (1, 2, 3, 100), for the options that have unlimited number of possible values, then those options are not tuned. An exception is the STRONGITER= option, where the default tuning values are (–1, 100, 10000, 2147483647).

Variables in the TUNEROUT= Data Set

The TUNEROUT= data set contains the following variables:

OBJSENSE

specifies the objective sense used for each tuning problem. The value is either MIN or MAX.

PROBLEM

specifies a list of data set names. Each named data set contains one of the problems that are used by the tuner.

RANK

specifies the rank of each option configuration, based on the criteria specified by the GOAL= option. The row that has a rank value of 0 contains the solution information of either the initial values that you provide or the solver default values for the options. When a solution status is not optimal, feasible, or bounded, a penalty is applied to the solution time in the ranking. If you want to put other factors

(such as the relative gap) into the ranking, then you can define your own rules by using the solution information in this data set.

The tuner's ranking is based on solver's running time, which has nondeterministic nature, so the tuning results may vary among different runs.

Option Configurations

name each variable for a tuning option and contain the option value that is used for the current option configuration.

Solution Information

specifies solution information for each option configuration that the tuner evaluates. This information includes the status, solution status, objective value, relative gap, absolute gap, nodes, and solution time. For more information about these terms, see the section "Macro Variable _OROPTMILP_" on page 445. When the tuner's time limit is reached, any unfinished runs will have the solution status TUNER_TIME_LIM.

Default Set of Tuning Options

Table 14.5 lists the options and values that the tuner uses when OPTIONMODE=AUTOMATIC.

Table 14.5   Default Set of Tuning Options

Option	Values
CONFLICTSEARCH=	−1, 0
CUTGOMORY=	−1, 0
CUTMILIFTED=	−1, 0
CUTSTRATEGY=	−1, 0, 1, 2
CUTZEROHALF=	−1, 0
HEURISTICS=	−1, 0, 1, 2, 3
NODESEL=	−1, 0, 1, 2
PRESOLVER=	−1, 0, 1, 2, 3
PROBE=	−1, 0
VARSEL=	−1, 3

For more information about these options, see the section "Functional Summary" on page 421 in Chapter 11, "The OPTMILP Procedure."

Full Set of Tuning Options

When OPTIONMODE=FULL, the tuner tunes the set of options listed in Table 14.6 over an automatically determined range.

Table 14.6 Full Set of Tuning Options

ALLCUTS=	CUTMIR=
CONFLICTSEARCH=	CUTSFACTOR=
CUTCLIQUE=	CUTSTRATEGY=
CUTFLOWCOVER=	CUTZEROHALF=
CUTFLOWPATH=	HEURISTICS=
CUTGOMORY=	NODESEL=
CUTGUB=	PRESOLVER=
CUTIMPLIED=	PROBE=
CUTKNAPSACK=	SCALE=
CUTLAP=	STRONGITER=
CUTMILIFTED=	VARSEL=

For more information about these options, see the section "Functional Summary" on page 421 in Chapter 11, "The OPTMILP Procedure." You can also tune other performance related OPTMILP options that are not listed here by using the OPTIONVALUES= option.

Tuner Log

The following information about the option tuner is printed in the tuner log:

SolveCalls
: indicates the number of problems that the tuner has completed.

Configurations
: indicates the number of configurations that the tuner has completed.

BestTime
: indicates the geometric mean or sum of the solve times (over all tuning problems) of the current best option configuration. When one of the solves comes from an unsuccessful run, an asterisk (*) is placed next to the time.

Time
: indicates the time (in seconds) that is used by the tuner.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the tuner log. Figure 14.2 shows a sample tuner log.

Figure 14.2 Sample Option Tuner Log

```
NOTE: The Option Tuning algorithm (the Tuner) is enabled.
NOTE: The non-deterministic parallel mode is enabled.
NOTE: The Tuner is using up to 4 threads.
      SolveCalls  Configurations      BestTime         Time
              10               5         86.11       296.43
              20              10         79.96       543.70
              30              14         76.72       797.27
              40              20         76.72      1074.21
NOTE: The tuning time is 1254.35 seconds.
NOTE: The data set WORK.OUT has 48 observations and 13 variables.
```

ODS Tables

The tuner creates several Output Delivery System (ODS) tables by default unless you specify a value other than 1 for the PRINTLEVEL= option in the PROC OPTMILP statement. The names of these tables are listed in Table 14.7. The TunerInfo and TunerSummary tables contain the tuner's input summary and results summary, respectively. The TunerResults table contains the option values, geometric mean and summation of the solution times, success rate for the initial option configuration, and a selection of best option configurations and worst option configurations.They are sorted according to the success rate and performance measure specified by the GOAL= option. The *Config 0* column contains the solution information of either the initial values that you provided or the solver default values for the options.

The "Performance Information" table is produced by default. It displays information about the execution mode. For single-machine mode, the table displays the number of threads used. For distributed mode, the table displays the grid mode (symmetric or asymmetric), the number of compute nodes, and the number of threads per node.

If you specify the DETAILS option in the PERFORMANCE statement, the procedure also produces a "Timing" table in which the accumulated elapsed times (absolute and relative) for the main tasks of the procedure are displayed.

You can create output data sets from these tables by specifying the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Table 14.7 ODS Tables Produced by the OPTMILP Option Tuner

ODS Table Name	Description
PerformanceInfo	Performance information
Timing	Timing report
TunerInfo	Summary of option tuning input
TunerResults	Option tuning results
TunerSummary	Summary of option tuning results

Figure 14.3 shows an example PerformanceInfo table for single-machine mode.

Figure 14.3 Example Tuner Output: PerformanceInfo

```
                    The OPTMILP Procedure


                 Performance Information


        Execution Mode        Single-Machine
        Number of Threads                  4
```

Figure 14.4 shows an example Timing table.

Figure 14.4 Example Tuner Output: Timing

```
                   Procedure Task Timing
                            Time
         Task               (sec.)      % Time

         Data Loading         0.53       0.01%
         Data Transfer        0.00       0.00%
         Tuner                0.76       0.02%
         Solver            4894.80      99.95%
         Idle                 1.40       0.03%
```

Figure 14.5 shows an example TunerInfo table.

Figure 14.5 Example Tuner Output: TunerInfo

```
                   Tuner Information

         Target Solver                     MILP
         Number of Tuning Options             3
         Number of Tuning Instances           2
         Tuning Option Set                 USER
         Performance Goal               GEOMEAN
         Tuner Time Limit                  1200
         Tuner Configurations Limit  2147483647
```

Figure 14.6 shows an example TunerResults table.

Figure 14.6 Example Tuner Output: TunerResults

```
                           Tuner Results

                  C         C         C         C         C         C         C
                  o         o         o         o         o         o         o
                  n         n         n         n         n         n         n
                  f         f         f         f         f         f         f
                  i         i         i         i         i         i         i
                  g         g         g         g         g         g         g

                  0         1         2         3         4         5         6

cutclique            -1        -1         2        -1        -1        -1        -1
cutgomory            -1        -1         1         0         1         1         1
heuristics           -1         2         2         2         0         2         3
Mean of Run Times  86.11     72.77     76.72     79.96     82.16     89.96    108.19
Sum of Run Times  176.16    147.99    156.56    162.85    179.53    183.26    236.56
Percentage Successful 100      100       100       100       100       100       100

                           Tuner Results

                                                                      C
                                                                      o
                            C         C         C                     n
                            o         o         o                     f
                            n         n         n                     i
                            f         f         f                     g
                            i         i         i
                            g         g         g                     1
                                                                      0
                            7         8         9

        cutclique           1         0         2        -1
        cutgomory           0         1        -1         2
        heuristics          1         3         3         3
        Mean of Run Times 108.79    111.71     93.49     62.7
        Sum of Run Times  224.39    240.38    195.28    125.5
        Percentage Successful 100      100        50         0
```

Figure 14.7 shows an example TunerSummary table.

Figure 14.7 Example Tuner Output: TunerSummary

```
                        Tuner Summary

        Actual Tuning Time                        1254.35
        Initial Run Time (geomean)                  86.11
        Initial Run Time (sum)                     176.16
        Best Run Time (geomean)                     72.78
        Best Run Time (sum)                        147.99
        Number of Improved Configurations               4
        Number of Tested Configurations                24
```

Examples: The OPTMILP Option Tuner

Example 14.1: Tuning the Default Set of Options for a Single Problem

This example demonstrates how to tune the default set of tuning options for a single problem. The problem is the air05 problem from the MIPLIB 2003 problem set, which is introduced in the section "Getting Started: The OPTMILP Option Tuner" on page 580. The SAS data set that defines the problem (in MPS format) is named *air05*.

Because you are using only one problem to perform option tuning, you do not need to create a PROBLEMS= data set. Because you are tuning the default set of options, you do not need to create an OPTIONVALUES= data set. The following statements call the OPTMILP option tuner and determine the stopping criterion by specifying the MAXCONFIGS= option instead of the MAXTIME= option:

```
proc optmilp data=a.air05 maxtime=300;
   tuner maxconfigs=20 printfreq=2 tunerout=out;
   performance nthreads=4;
run;

title "Tuner Output";
proc print data=out(obs=10);
run;
```

The output data set is shown in Figure 14.1.1.

Output 14.1.1 Single Problem with Default Tuning Options: Output

```
                                    Tuner Output
```

Obs	RANK	PROBLEM	OBJSENSE	PRESOLVER	PROBE	CONFLICTSEARCH	NODESEL	VARSEL	HEURISTICS	CUTSTRATEGY	CUTGOMORY	CUTMILIFTED	CUTZEROHALF	STATUS	SOLUTION_STATUS	OBJECTIVE	RELATIVE_GAP	ABSOLUTE_GAP	NODES	SOLUTION_TIME
1	0	AIR05	MIN	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	OK	OPTIMAL	26374	0	0	190	61.17
2	1	AIR05	MIN	3	-1	-1	2	-1	2	2	-1	0	0	OK	OPTIMAL	26374	0	0	175	51.97
3	2	AIR05	MIN	3	-1	-1	2	-1	2	1	-1	-1	0	OK	OPTIMAL	26374	0	0	203	52.92
4	3	AIR05	MIN	3	-1	-1	2	-1	2	1	-1	0	0	OK	OPTIMAL	26374	0	0	203	54.79
5	4	AIR05	MIN	3	-1	0	2	-1	2	1	-1	0	0	OK	OPTIMAL	26374	0	0	307	55.18
6	5	AIR05	MIN	3	0	-1	2	-1	2	1	-1	0	0	OK	OPTIMAL	26374	0	0	203	55.38
7	6	AIR05	MIN	3	-1	-1	2	-1	3	1	-1	0	0	OK	OPTIMAL	26374	0	0	189	58.10
8	7	AIR05	MIN	3	-1	-1	-1	-1	2	1	-1	0	0	OK	OPTIMAL	26374	0	0	227	60.34
9	8	AIR05	MIN	3	-1	-1	1	-1	2	1	-1	0	0	OK	OPTIMAL	26374	0	0	275	69.41
10	9	AIR05	MIN	0	-1	-1	0	-1	0	-1	-1	-1	0	OK	OPTIMAL	26374	0	0	105	70.12

Example 14.2: Tuning a Defined Set of Options for Multiple Problems

This example demonstrates how to specify a set of tuning options and tune them for multiple problems.

The following DATA step creates a PROBLEMS= data set named probs that contains the list of tuning problems. This data set is the same as in the section "Getting Started: The OPTMILP Option Tuner" on page 580.

```
data probs;
   input name $1-8;
   datalines;
a.air04
a.air05
;
```

The following DATA step creates an OPTIONVALUES= data set named optvals that is different from the default set, which is described in the section "Default Set of Tuning Options" on page 588:

```
data optvals;
   input option $1-10 values $12-28 initial $30-32;
   datalines;
cutclique   -1, 0, 2          -1
cutgomory                      1
heuristics
;
```

The optvals data set contains a nondefault list of tuning values for the CUTCLIQUE= option in addition to initial values for the CUTCLIQUE= and CUTGOMORY= options. The options for which sets of tuning values are not specified (in this case, the CUTGOMORY= and HEURISTICS= options) are tuned for all available values. The options for which initial values are not specified (in this case, the HEURISTICS= option) are tuned by using the default initial value.

The following statements call the OPTMILP option tuner and then print the ODS table TunerResults and the TUNEROUT= data set:

```
proc optmilp maxtime=300;
   tuner problems=probs  optionvalues=optvals  optionmode=user
         maxtime=1200 tunerout=out;
   performance nthreads=4;
run;

title "Tuner Output";
proc print data=out(obs=20);
run;
```

The output is shown in Figure 14.2.1.

Output 14.2.1 Multiple Problems with Specified Tuning Options: Output

```
                                    Tuner Output

                               The OPTMILP Procedure

                                   Tuner Results
```

	C o n f i g 0	C o n f i g 1	C o n f i g 2	C o n f i g 3	C o n f i g 4	C o n f i g 5	C o n f i g 6
cutclique	-1	2	-1	2	2	-1	-1
cutgomory	1	0	2	2	0	2	2
heuristics	-1	-1	-1	2	2	2	3
Mean of Run Times	76.3	73.36	73.41	73.8	74.48	75.39	109.51
Sum of Run Times	155	149.4	149.49	150.26	151.61	153.79	239.94
Percentage Successful	100	100	100	100	100	100	100

```
                                   Tuner Results
```

	C o n f i g 7	C o n f i g 8	C o n f i g 9	C o n f i g 1 0
cutclique	2	0	-1	-1
cutgomory	1	0	2	0
heuristics	3	3	0	2
Mean of Run Times	109.57	110.98	24.31	43.09
Sum of Run Times	240.42	236.78	48.63	98.61
Percentage Successful	100	100	0	0

Output 14.2.1 *continued*

```
                              Tuner Output
```

Obs	RANK	PROBLEM	OBJSENSE	cutclique	cutgomory	heuristics	STATUS	SOLUTION_STATUS	OBJECTIVE	RELATIVE_GAP	ABSOLUTE_GAP	NODES	SOLUTION_TIME
1	0	air04	MIN	-1	1	-1	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	91.08
2	0	air05	MIN	-1	1	-1	OK	OPTIMAL	26374	.000000000	0.00000	190	63.91
3	1	air04	MIN	2	0	-1	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	88.77
4	1	air05	MIN	2	0	-1	OK	OPTIMAL	26374	.000000000	0.00000	192	60.63
5	2	air04	MIN	-1	2	-1	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	88.80
6	2	air05	MIN	-1	2	-1	OK	OPTIMAL	26374	.000000000	0.00000	192	60.69
7	3	air04	MIN	2	2	2	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	89.16
8	3	air05	MIN	2	2	2	OK	OPTIMAL	26374	.000000000	0.00000	192	61.09
9	4	air04	MIN	2	0	2	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	89.91
10	4	air05	MIN	2	0	2	OK	OPTIMAL	26374	.000000000	0.00000	192	61.70
11	5	air04	MIN	-1	2	2	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	92.00
12	5	air05	MIN	-1	2	2	OK	OPTIMAL	26374	.000000000	0.00000	192	61.79
13	6	air04	MIN	-1	-1	2	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	92.88
14	6	air05	MIN	-1	-1	2	OK	OPTIMAL	26374	.000000000	0.00000	190	62.45
15	7	air04	MIN	-1	1	2	OK	OPTIMAL_RGAP	56137	.000053444	3.00000	96	95.99
16	7	air05	MIN	-1	1	2	OK	OPTIMAL	26374	.000000000	0.00000	190	65.45
17	8	air04	MIN	2	-1	1	OK	OPTIMAL_RGAP	56137	.000067097	3.76640	159	126.33
18	8	air05	MIN	2	-1	1	OK	OPTIMAL	26374	.000000000	0.00000	192	53.99
19	9	air04	MIN	2	0	0	OK	OPTIMAL_RGAP	56137	.000067097	3.76640	162	129.10
20	9	air05	MIN	2	0	0	OK	OPTIMAL	26374	.000000000	0.00000	192	54.11

Example 14.3: Tuning a Defined Set of Options for Multiple Problems in Distributed Mode

This example demonstrates how to run the tuner in distributed mode. The example is similar to Example 14.2. The only difference between single-machine and distributed mode is that the PERFORMANCE statement specifies the number of threads and nodes to be used. The following statement changes the operating mode to distributed:

```
/* set the numbers of nodes and threads and get performance details */
performance nodes=5 nthreads=4 details;
```

The performance information and procedure task timing tables are displayed in Figure 14.3.1. The NODES=5 and NTHREADS=4 options in the PERFORMANCE statement cause the tuner to run in distributed mode, where each computer node processes up to four threads simultaneously.

Output 14.3.1 Performance Information in Distributed Mode: Output

```
                        Tuner Output

                   The OPTMILP Procedure

                  Performance Information

Host Node                      < your grid host >
Execution Mode                 Distributed
Grid Mode                      Symmetric
Number of Compute Nodes        5
Number of Threads per Node     4


                  Procedure Task Timing
                            Time
            Task            (sec.)      % Time

            Data Loading      0.83       0.02%
            Data Transfer     0.08       0.00%
            Tuner             0.27       0.01%
            Solver         3994.81      99.95%
            Idle              0.96       0.02%
```

References

Achterberg, T., Koch, T., and Martin, A. (2003), "MIPLIB 2003," http://miplib.zib.de/.

Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. (1998), "An Updated Mixed Integer Programming Library: MIPLIB 3.0," *Optima*, 58, 12–15.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stuetzle, T. (2009), "ParamILS: An Automatic Algorithm Configuration Framework," *Journal of Artificial Intelligence Research*, 36, 267–306.

Chapter 15
The MPS-Format SAS Data Set

Contents

Overview: MPS-Format SAS Data Set

The MPS file format is a format commonly used in industry for describing linear programming (LP) and integer programming (IP) problems (Murtagh 1981; IBM 1988). It can be extended to the QPS format (Maros and Mészáros 1999), which describes quadratic programming (QP) problems. MPS-format and QPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format SAS data set corresponds closely to the format of an MPS-format or QPS-format file and is used to describe linear programming, mixed integer programming, and quadratic programming problems for SAS/OR.

Observations

An MPS-format data set contains six variables: field1, field2, field3, field4, field5, and field6. The variables field4 and field6 are numeric type; the others are character type. Among the character variables, only the value of field1 is case-insensitive and leading blanks are ignored. Values of field2, field3, and field5 are case-sensitive and leading blanks are NOT ignored. Not all variables are used in a particular observation.

Observations in an MPS-format SAS data set are grouped into sections. Each section starts with an *indicator record*, followed by associated *data records*. Indicator records specify the names of sections and the format of the following data records. Data records contain the actual data values for a section.

Order of Sections

Sections of an MPS-format SAS data set must be specified in a **fixed** order.

Sections of linear programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- ENDATA

Sections of quadratic programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- QSECTION
- ENDATA

Sections of mixed integer programming problems are listed in the following order:

- NAME

- ROWS

- COLUMNS

- RHS (optional)

- RANGES (optional)

- BOUNDS (optional)

- BRANCH (optional)

- ENDATA

Sections Format: MPS-Format SAS Data Set

The following subsections describe the format of the records for each section of the MPS-format data set. Note that each section contains two types of records: an indicator record and multiple data records. The following subsections of this documentation describe the two different types of records for each section of the MPS data set.

NAME Section

The NAME section contains only a single record identifying the name of the problem.

Field1	Field2	Field3	Field4	Field5	Field6
NAME	Blank	Input model name	.	Blank	.

ROWS Section

The ROWS section contains the name and type of the rows (linear constraints or objectives). The type of each row is specified by the indicator code in field1 as follows:

- **MIN**: minimization objective

- **MAX**: maximization objective

- **N**: objective

- **G**: $\geq$ constraint

- **L**: $\leq$ constraint

- **E**: $=$ constraint

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
ROWS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Indicator code	Row name	Blank	.	Blank	.

Notes:

1. At least one objective row should be specified in the ROWS section. It is possible to specify multiple objective rows. However, among all the data records indicating the objective, only the first one is regarded as the objective row, while the rest are ignored. If a type-N row is taken as the objective row, minimization is assumed.

2. Duplicate entries of field2 in the ROWS section are not allowed. In other words, row name is unique. The variable field2 in the ROWS section cannot take a missing value.

COLUMNS Section

The COLUMNS section defines the column (i.e., variable or decision variable) names of the problem. It also specifies the coefficients of the columns for each row.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
COLUMNS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Column name (e.g., col)	Row name (e.g., rowi)	Matrix element in row rowi, column col	Row name (e.g., rowj)	Matrix element in row rowj, column col

Notes:

1. All elements belonging to one column must be grouped together.

2. A missing coefficient value is ignored. A data record with missing values in both field4 and field6 is ignored.

3. Duplicate entries in each pair of column and row are not allowed.

4. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. The value in field2 of the first data record in the section cannot be missing.

Mixed Integer Programs

Mixed integer programming (MIP) problems require you to specify which variables are constrained to be integers. Integer variables can be specified in the COLUMNS section with the use of special marker records in the following form:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Marker name	'MARKER' (including the quotation marks)	.	'INTORG' or 'INTEND' (including the quotation marks)	.

A marker record with field5 that contains the value 'INTORG' indicates the start of integer variables. In the marker record that indicates the end of integer variables, field5 must be 'INTEND'. An alternative way to specify integer variables without using the marker records is described in the section "BOUNDS Section (Optional)" on page 605.

Notes:

1. INTORG and INTEND markers must appear in pairs in the COLUMNS section. The marker pairs can appear any number of times.

2. The marker name in field2 should be different from the preceding and following column names.

3. All variables between the INTORG and INTEND markers are assumed to be binary unless you specify a different lower bound and/or upper bound in the BOUNDS section.

RHS Section (Optional)

The RHS section specifies the right-hand-side value for the rows. Any row unspecified in this section is considered to have an RHS value of 0. Missing the entire RHS section implies that all RHS values are 0.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
RHS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	RHS name	Row name (e.g., rowi)	RHS value for row rowi	Row name (e.g., rowj)	RHS value for row rowj

Notes:

1. The rows that have an RHS element defined in this section need not be specified in the same order in which the rows were specified in the ROWS section. However, a row in the RHS section should be defined in the ROWS section.

2. It is possible to specify multiple RHS vectors, which are labeled by different RHS names. Normally, the first RHS vector encountered in the RHS section is used, and all other RHS vectors are discarded. All the elements of the selected RHS vector must be specified before other RHS vectors are introduced. Within a specific RHS vector, for a given row, duplicate assignments of RHS values are not allowed.

3. An RHS value assigned to the objective row is ignored by PROC OPTLP and PROC OPTMILP, while it is taken as a constant term of the objective function by PROC OPTQP.

4. A missing value in field4 or field6 is ignored. A data record with missing values in both field4 and field6 is ignored.

5. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

RANGES Section (Optional)

The RANGES section specifies the range of the RHS value for the constraint rows. With range specification, a row can be constrained from above and below.

For a constraint row c, if b is the RHS value and r is the range for this row, then the equivalent constraints are given in Table 15.1, depending on the type of row and the sign of r.

Table 15.1   Range Effect

Type of Row	Sign of r	Equivalent Constraints
G	$\pm$	$b \leq c \leq b + |r|$
L	$\pm$	$b - |r| \leq c \leq b$
E	$+$	$b \leq c \leq b + r$
E	$-$	$b + r \leq c \leq b$

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
RANGES	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Range name	Row name (e.g., rowi)	Range for RHS of row rowi	Row name (e.g., rowj)	Range for RHS of row rowj

Notes:

1. Range assignment for an objective row (i.e., **MAX**, **MIN**, or **N** row) is not allowed.

2. The rows that have a range element defined in this section need not be specified in the same order in which the rows were specified in the ROWS or RHS section. However, a row in the RANGES section should be defined in the ROWS section.

3. It is possible to specify multiple range vectors, which are labeled by different range names. Normally, the first range vector encountered in the RANGES section is used, and all other range vectors are discarded. All the elements in a range vector must be specified before other range vectors are introduced. Within the specific range vector, for a given range, duplicate assignments of range values are not allowed.

4. A missing value in field4 or field6 is ignored. A data record with missing values in both field4 and field6 is ignored.

5. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

BOUNDS Section (Optional)

The BOUNDS section specifies bounds for the columns.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
BOUNDS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Bound type	Bound name	Column name	Bound for the column	Blank	Blank

Notes:

1. If you do not specify any bound for a column, then the upper bound is $+\infty$ for a continuous variable, and 1 for an integer variable, that is specified in the COLUMNS section. The lower bound is 0 by default.

2. General bound types include **LO**, **UP**, **FX**, **FR**, **MI**, and **PL**. Suppose the bound for a column identified in field3 is specified as b in field4. Table 15.2 explains the effects of different bound types.

Table 15.2 Bound Type Rules

Bound Type	Ignore b	Resultant Lower Bound	Resultant Upper Bound
LO	No	b	*unspecified*
UP	No	*unspecified*	b
FX	No	b	b
FR	Yes	$-\infty$	$+\infty$
MI	Yes	$-\infty$	*unspecified*
PL	Yes	*unspecified*	$+\infty$

If a bound (lower or upper) is not explicitly specified, then it takes the default values according to Note 1. There is one exception: if the upper bound is specified as a negative value ($b < 0$) and the lower bound is unspecified, then the lower bound is set to $-\infty$.

Mixed integer programming problems can specify integer variables in the BOUNDS section. Table 15.3 shows bound types defined for MIP.

Table 15.3 Bound Type Rules

Bound Type	Ignore b	Variable Type	Value
BV	Yes	binary	0 or 1
LI	No	integer	$[b, \infty)$
UI	No	integer	$(-\infty, b]$

3. The columns that have bounds do not need to be specified in the same order in which the columns were specified in the COLUMNS section. However, all columns in the BOUNDS section should be defined in the COLUMNS section.

4. It is possible to specify multiple bound vectors, which are labeled by different bound names. Normally, the first bound vector encountered in the BOUNDS section is used, and all other bound vectors are discarded. All the elements of the selected bound vector must be specified before other bound vectors are introduced.

5. When data records in a sequence have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

6. Within a particular BOUNDS vector, for a given column, if a bound (lower or upper) is explicitly specified by the bound type rules listed in Table 15.2, any other specification is considered to be an error.

7. If the value in field1 is **LO**, **UP**, **FX**, **LI**, or **UI**, then a data record with a missing value in field4 is ignored.

BRANCH Section (Optional)

Sometimes you want to specify branching priorities or directions for integer variables to improve performance. Variables with higher priorities are branched on before variables with lower priorities. The branch direction is used to decide which branch to take first at each node. For more information, see the section "Branching Priorities" on page 438.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
BRANCH	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Branch direction	Blank	First column name	First column priority	Second column name	Second column priority

Notes:

1. Valid directions include **UP** (up branch), **DN** (down branch), **RD** (rounding) and **CB** (closest bound). If field1 is blank, the solver automatically decides the direction.

2. If field4 is missing, then the name defined in field3 is ignored. Similarly, if field6 is missing, then the name defined in field5 is ignored.

3. The priority value in field4 and field6 must be nonnegative. Zero is the lowest priority and is also the default.

QSECTION Section

The QSECTION section is needed only to describe quadratic programming problems. It specifies the coefficients of the quadratic terms in the objective function.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
QSECTION or QUADOBJ	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Column name	Column name	Coefficient in objective function	Blank	.

Notes:

1. The QSECTION section is required for PROC OPTQP and should not appear for PROC OPTLP. For PROC OPTQP, there should be at least one valid data record in the QSECTION section. For PROC OPTLP, an error is reported when the submitted data set contains a QSECTION section.

2. The variables field2 and field3 contain the names of the columns that form a quadratic term in the objective function. They must have been defined in the COLUMNS section. They need not refer to the same column. Zero entries should not be specified.

3. Duplicate entries of a quadratic term are not allowed. This means the combination of (field2, field3) must be unique, where (k, j) and (j, k) are considered to be the same combination.

4. If field4 of one data record is missing or takes a value of zero, then this data record is ignored.

ENDATA Section

The ENDATA section simply declares the end of all records. It contains only one indicator record, where field1 takes the value ENDATA and the values of the remaining variables are blank or missing.

Details: MPS-Format SAS Data Set

Converting an MPS/QPS-Format File: %MPS2SASD

As described in the section "Overview: MPS-Format SAS Data Set" on page 599, the MPS or QPS format is a standard file format for describing linear, integer, and quadratic programming problems. The MPS/QPS format is defined for plain text files, whereas in the SAS System it is more convenient to read data from SAS data sets. Therefore, a facility is required to convert MPS/QPS-format text files to MPS-format SAS data sets. The SAS macro %MPS2SASD serves this purpose.

In the MPS/QPS-format text file, a record refers to a single line of text that is divided into six fields. MPS/QPS files can be read and printed in both *fixed* and *free* format. In fixed MPS/QPS format, each field of a data record must occur in specific columns:

Field	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Columns	2–3	5–12	15–22	25–36	40–47	50–61

In free format, fields of a record are separated by a space. Both fixed and free format have limitations. If users need to use row names or column names longer than 8 characters, then there is not enough space to hold them in fixed format. If users use a space as a part of a row name or column name, such as "ROW NAME", then free-format MPS format interprets this symbol as two fields, "ROW" and "NAME".

You can insert a comment record, denoted by an asterisk (*) in column 1, anywhere in an MPS/QPS file. Also, if a dollar sign ($) is the first character in field 3 or field 5 of any record, the information from that point to the end of the record is treated as a comment. All comments are ignored by the %MPS2SASD macro, described as follows.

%MPS2SASD Macro Parameters

%MPS2SASD (MPSFILE='infilename', OUTDATA=mpsdata, MAXLEN=n, FORMAT=FIXED/FREE);

MPSFILE=*'infilename'*
> specifies the path and name of the input MPS-format file. The input file is a plain text file, normally with either an ".mps" extension for linear programming problems or a ".qps" extension for quadratic programming problems. This parameter is required; there is no default value.

OUTDATA=*mpsdata*
> specifies the name of the output MPS-format SAS data set. This parameter is optional; the default value is mpsdata.

MAXLEN=*n*
> specifies length of the variables field2, field3, and field5 in the output MPS-format SAS data set. This parameter is optional; the default value is 8.

FORMAT=*FIXED/FREE*
> specifies the format of the input MPS file. Valid values can be either FIXED or FREE. This parameter is optional; the default value is the one, if any, specified by the flat file and FIXED otherwise.

Length of Variables

In an MPS-format SAS data set, normally the variables field2, field3, and field5 hold the names of the rows and columns. The length of these character variables is limited to the maximum size of a SAS character variable. This enables you to use sufficiently long names for the rows and columns in your model.

In a SAS data set generated by the %MPS2SASD macro, the length of the variables field2, field3, and field5 is fixed to be 8 ASCII characters by default. This length fits the fixed-format MPS/QPS file well since field 2, field 3, and field 5 are fixed at 8 characters. However, the free-format MPS/QPS files might have longer row names or longer column names. The %MPS2SASD macro provides a parameter **MAXLEN = n**. Using this parameter, you can set the variables field2, field3, and field5 to have a length of n characters in the output SAS data set.

The parameter MAXLEN works only when the given MPS file is in free format. For a fixed-format MPS file, this parameter is ignored and the length of field2, field3, and field5 is 8 characters by default.

Examples: MPS-Format SAS Data Set

Example 15.1: MPS-Format Data Set for a Product Mix Problem

Consider a simple product mix problem where a furniture company tries to find an optimal product mix of four items: a desk (x_1), a chair (x_2), a cabinet (x_3), and a bookcase (x_4). 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.

This problem can be expressed as the following linear program:

$$
\begin{aligned}
\max \quad & 95x_1 + 41x_2 + 84x_3 + 76x_4 \\
\text{subject to} \quad & 3x_1 + 1.5x_2 + 2x_3 + 2x_4 \leq 800 \quad \text{(STAMP)} \\
& 10x_1 + 6x_2 + 8x_3 + 7x_4 \leq 1200 \quad \text{(ASSEMB)} \\
& 10x_1 + 6x_2 + 8x_3 + 7x_4 \geq 300 \quad \text{(ASSEMB)} \\
& 10x_1 + 8x_2 + 8x_3 + 7x_4 \leq 800 \quad \text{(FINISH)} \\
& x_2 \leq 75 \\
& x_4 \geq 50 \\
& x_i \geq 0 \quad i = 1, 2, 3
\end{aligned}
$$

The following DATA step saves the problem specification as an MPS-format SAS data set:

```
data prodmix;
   infile datalines;
   input field1 $ field2 $ field3$ field4 field5 $ field6;
   datalines;
NAME                .          PROD_MIX         .          .          .
ROWS                .          .                .          .          .
MAX             PROFIT         .                .          .          .
L               STAMP          .                .          .          .
L               ASSEMB         .                .          .          .
L               FINISH         .                .          .          .
COLUMNS             .          .                .          .          .
   .            DESK           STAMP            3.0        ASSEMB     10
   .            DESK           FINISH           10.0       PROFIT     95
   .            CHAIR          STAMP            1.5        ASSEMB     6
   .            CHAIR          FINISH           8.0        PROFIT     41
   .            CABINET        STAMP            2.0        ASSEMB     8
   .            CABINET        FINISH           8.0        PROFIT     84
   .            BOOKCSE        STAMP            2.0        ASSEMB     7
   .            BOOKCSE        FINISH           7.0        PROFIT     76
RHS                 .          .                .          .          .
   .            TIME           STAMP            800.0      ASSEMB     1200
   .            TIME           FINISH           800.0      .          .
RANGES              .          .                .          .          .
   .            T1             ASSEMB           900.0      .          .
BOUNDS              .          .                .          .          .
UP              BND            CHAIR            75.0       .          .
LO              BND            BOOKCSE          50.0       .          .
ENDATA              .          .                .          .          .
;
```

Example 15.2: Fixed-MPS-Format File

The following file, example_fix.mps, contains the data from Example 15.1 in the form of a fixed-MPS-format file. The indicator codes MAX and MIN are not available for objective rows in fixed MPS format, so the PROFIT row is specified as type N. Minimization is assumed for type-N rows; for a maximization objective, the objective coefficients must be replaced with values of the opposite sign.

```
* THIS IS AN EXAMPLE FOR FIXED MPS FORMAT.
NAME            PROD_MIX
ROWS
 N  PROFIT
 L  STAMP
 L  ASSEMB
 L  FINISH
COLUMNS
    DESK        STAMP              3.00000    ASSEMB          10.00000
    DESK        FINISH            10.00000    PROFIT         -95.00000
    CHAIR       STAMP              1.50000    ASSEMB           6.00000
    CHAIR       FINISH             8.00000    PROFIT         -41.00000
    CABINET     STAMP              2.00000    ASSEMB           8.00000
    CABINET     FINISH             8.00000    PROFIT         -84.00000
    BOOKCSE     STAMP              2.00000    ASSEMB           7.00000
    BOOKCSE     FINISH             7.00000    PROFIT         -76.00000
RHS
    TIME        STAMP            800.00000    ASSEMB        1200.0000
    TIME        FINISH           800.00000
RANGES
    T1          ASSEMB           900.00000
BOUNDS
 UP BND         CHAIR             75.00000
 LO BND         BOOKCSE           50.00000
ENDATA
```

Example 15.3: Free-MPS-Format File

In free format, fields in data records other than the first record have no predefined positions. They can be written anywhere except column 1, with each field separated from the next by one or more blanks (a tab cannot be used as a field separator). However, the fields must appear in the same sequence as in the fixed format. The following file, example_free.mps, is an example. It describes the same problem as in Example 15.2.

```
* THIS IS AN EXAMPLE FOR FREE MPS FORMAT.
NAME            PROD_MIX  FREE
ROWS
 N  PROFIT
    L  STAMP
    L  ASSEMB
    L  FINISH
```

```
COLUMNS
 DESK        STAMP     3.00000    ASSEMB      10.00000
 DESK        FINISH   10.00000     PROFIT    -95.00000
    CHAIR    STAMP     1.50000    ASSEMB       6.00000
    CHAIR    FINISH    8.00000     PROFIT    -41.00000
 CABINET     STAMP     2.00000    ASSEMB       8.00000
 CABINET     FINISH    8.00000     PROFIT    -84.00000
    BOOKCSE STAMP      2.00000    ASSEMB       7.00000
    BOOKCSE FINISH     7.00000     PROFIT    -76.00000
 RHS
    TIME STAMP     800.00000   ASSEMB 1200.0000
    TIME FINISH    800.00000
 RANGES
    T1    ASSEMB     900.00000
 BOUNDS
 UP BND       CHAIR 75.00000
 LO BND     BOOKCSE 50.00000
 ENDATA
```

Example 15.4: Using the %MPS2SASD Macro

We illustrate the use of the %MPS2SASD macro in this example, assuming the files example_fix.mps and example_free.mps are in your current SAS working directory.

The MPS2SASD macro function has one required parameter, MPSFILE= 'infilename', which specifies the path and name of the MPS/QPS-format file. With this single parameter, the macro reads the file, converts the records, and saves the conversion to the default MPS-format SAS data set MPSDATA.

Running the following statements converts the fixed-format MPS file shown in Example 15.2 to the MPS-format SAS data set MPSDATA:

```
%mps2sasd(mpsfile='example_fix.mps');
proc print data=mpsdata;
run;
```

Output 15.4.1 displays the MPS-format SAS data set MPSDATA.

Output 15.4.1 The MPS-Format SAS Data Set MPSDATA

Obs	field1	field2	field3	field4	field5	field6
1	NAME		PROD_MIX	.		.
2	ROWS			.		.
3	N	PROFIT		.		.
4	L	STAMP		.		.
5	L	ASSEMB		.		.
6	L	FINISH		.		.
7	COLUMNS			.		.
8		DESK	STAMP	3.0	ASSEMB	10
9		DESK	FINISH	10.0	PROFIT	−95
10		CHAIR	STAMP	1.5	ASSEMB	6
11		CHAIR	FINISH	8.0	PROFIT	−41
12		CABINET	STAMP	2.0	ASSEMB	8
13		CABINET	FINISH	8.0	PROFIT	−84
14		BOOKCSE	STAMP	2.0	ASSEMB	7
15		BOOKCSE	FINISH	7.0	PROFIT	−76
16	RHS			.		.
17		TIME	STAMP	800.0	ASSEMB	1200
18		TIME	FINISH	800.0		.
19	RANGES			.		.
20		T1	ASSEMB	900.0		.
21	BOUNDS			.		.
22	UP	BND	CHAIR	75.0		.
23	LO	BND	BOOKCSE	50.0		.
24	ENDATA			.		.

Running the following statement converts the free-format MPS file shown in Example 15.3 to the MPS-format SAS data set MPSDATA:

```
%mps2sasd(mpsfile='example_free.mps');
```

The data set is identical to the one shown in Output 15.4.1.

In the following statement, when the free-format MPS file is converted, the length of the variables field2, field3, and field5 in the SAS data set MPSDATA is explicitly set to 10 characters:

```
%mps2sasd(mpsfile='example_free.mps', maxlen=10, format=free);
```

If you want to save the converted data to a SAS data set other than the default data set MPSDATA, you can use the parameter OUTDATA= mpsdata. The following statement reads data from the file example_fix.mps and writes the converted data to the data set PRODMIX:

```
%mps2sasd(mpsfile='example_fix.mps', outdata=PRODMIX);
```

References

IBM (1988), *Mathematical Programming System Extended/370 (MPSX/370) Version 2 Program Reference Manual*, volume SH19-6553-0, Armonk, NY: IBM.

Maros, I. and Mészáros, C. (1999), "A Repository of Convex Quadratic Programming Problems," *Optimization Methods and Software*, 11–12, 671–681.

Murtagh, B. A. (1981), *Advanced Linear Programming: Computation and Practice*, New York: McGraw-Hill.

Subject Index

Syntax Index

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

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