SAS/OR® 14.1 User's Guide
Local Search Optimization

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *SAS/OR® 14.1 User's Guide: Local Search Optimization*. Cary, NC: SAS Institute Inc.

SAS/OR® 14.1 User's Guide: Local Search Optimization

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

July 2015

Contents

Credits

Documentation

Writing	Steven Gardner, Joshua Griffin, Joe Hutchinson
Editing	Anne Baxter, Ed Huddleston
Documentation Support	Tim Arnold, Remya Chandran, Melanie Gratton, Michelle Opp, Daniel Underwood
Technical Review	Feng Chen, Gehan A. Corea, Tao Huang, Radhika V. Kulkarni

Software

PROC GA	Joe Hutchinson
PROC OPTLSO	Steven Gardner, Joshua Griffin
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	Tim Carter, Feng Chen, Girija Gavankar, Dright Ho, Tao Huang, Cheryl LeSaint, Jim McKenzie, Jim Metcalf, Bengt Pederson, Yong Wang, Lois Zhu
Technical Support	Tonya Chapman

Chapter 1
What's New in SAS/OR 14.1

Contents

Overview

SAS/OR 14.1 adds a number of new optimization capabilities that shorten optimization time, increase diagnostic capabilities, and make the software easier to use. Highlights include the following:

- Several solvers improve their performance.
- The concurrent FOR loop (the COFOR loop) in PROC OPTMODEL can run in distributed mode. **NOTE:** Distributed mode requires SAS High-Performance Optimization.
- PROC OPTMODEL adds a profiler that tracks the amount of time spent in problem generation, presolve, and various stages of the solution process.
- PROC OPTNET enables parallel computing, provides faster graph data input, and adds enhancements to three of its algorithms.
- The quadratic and nonlinear solvers add irreducible infeasible set (IIS) diagnostics.
- The decomposition algorithm expands the range of constraint matrix structures that it can detect automatically.
- The CLP procedure adds more variable selection strategies.

SAS Simulation Studio 14.1, a component of SAS/OR 14.1 for Windows environments, adds features that improve the accuracy of your models and give you additional controls on model execution. Highlights include the following:

- controls on the order in which dynamically created data input and output ports on blocks execute during the run of a model

- centralized controls on the ranking of blocks in a model, which determines the order of execution for events that are scheduled for the same simulation clock time
- expanded and improved controls on the allocation of resource units among resource entities when there is a scheduled adjustment
- automatic launching of the SAS server on your local PC

Mathematical Optimization Updates

Solver Performance Improvements

Several optimization solvers have improved their performance over that of SAS/OR 13.2 as follows:

- The interior point linear programming (LP) solver is 30% faster.
- The branch-and-cut mixed integer linear programming (MILP) solver is 35% faster.
- When METHOD=AUTO (the default setting), the decomposition (DECOMP) algorithm is 130% faster in finding an optimal MILP solution and 240% faster in finding a solution with a 5% optimality gap.
- The active set algorithm in the nonlinear programming (NLP) solver is 4% faster for the entire test suite and 10% faster for the more difficult problems in the test suite.

PROC OPTMODEL Improvements

The OPTMODEL procedure makes three major changes:

- The COFOR statement (which specifies a concurrent FOR loop) can distribute processing of solver invocations in the loop across multiple computational nodes, not simply among multiple computational threads on the same node. You request processing across multiple nodes by specifying a value greater than 1 in the NODES= option in the PERFORMANCE statement. **NOTE:** Distributed mode requires SAS High-Performance Optimization.
- CLP solver invocation via the SOLVE WITH CLP statement is promoted to production status.
- The PROFILE statement is added. This statement invokes and configures the PROC OPTMODEL profiler to track and report the time spent processing declarations, the time spent executing statements, and the number of times statements are executed. The profiler can help you identify elements of problem generation, presolve, and solution that require large amounts of time; it can be a useful aid in error detection and performance improvement. The PROFILE statement includes two optional arguments, *mode* and *options*. The *mode* argument enables or disables the profiler, and can also request that accumulated profiler data be printed. The *options* argument controls how PROC OPTMODEL collects, retains, and displays profiler information.

Other Optimization and Related Improvements

PROC CLP adds the new conflict-directed variable selection strategies, VARSELECT=WDEG and VARSELECT=DOMWDEG; adds a new dynamic variable section strategy, VARSELECT=DOMDDEG; and promotes the PACK and LEXICO constraint classes to production status.

PROC OPTNET enables faster graph data input. The STANDARDIZED_LABELS option in the PROC OPTNET statement enables you to read graph data that contain only numeric node identifiers more quickly. More generally, PROC OPTNET supports parallel computing, including parallel graph data input. This is especially helpful in shortening the time that is needed to input large-scale graph data. You can use the PERFORMANCE statement and its NTHREADS= option to request multithreaded computing.

PROC OPTNET adds enhancements to three of the algorithms that it uses. The TSP (traveling salesman problem) algorithm can solve asymmetric problems, which are defined on directed graphs; the shortest path algorithm can accept negative link weights; and the default connected components algorithm for undirected graphs is the more efficient union-find algorithm.

PROC OPTNET produces ODS tables as output. You can use the DETAILS option in the PERFORMANCE statement to request that PROC OPTNET produce the Timing ODS table, which reports the amount of time that each step of the procedure uses.

The NLP solver and the quadratic programming (QP) solver each add the IIS= option, which directs the solver to identify an irreducible infeasible set among the linear constraints and decision variable bounds for a problem that is found to be infeasible. Identification of irreducible infeasible sets provides valuable guidance in restoring infeasible problems to feasibility.

The MILP solver now runs in parallel (threaded or distributed) mode by default for improved performance. **NOTE:** Distributed mode requires SAS High-Performance Optimization.

The DECOMP algorithm for the LP and MILP solvers adds the SET value for the METHOD= option. This value directs the DECOMP algorithm to find a set-partitioning or set-covering structure in the constraint matrix. If such constraints are detected, they serve as the linking constraints, and the remaining weakly connected components of the constraint matrix define blocks of constraints for use in the DECOMP algorithm.

Discrete-Event Simulation Updates

SAS Simulation Studio 14.1, which provides a graphical environment for building and working with discrete-event simulation models, makes four major improvements:

- Three blocks have added new controls on the order in which dynamically created data input or output ports (or both) execute during the run of a model. Correct ordering of port execution can be critical to ensuring that your model functions properly. In previous releases, only the Formula block provided this feature, and the dynamic ports on other blocks executed in the order in which they were created. In SAS Simulation Studio 14.1, the Modifier, Extractor, and Gate blocks also include buttons, labeled **Move Up** and **Move Down**, that enable you to increase or decrease, respectively, the priority of any dynamic port in the order of execution.

- Centralized controls on the ranking of blocks in a model enable you to determine the order of execution for events that are scheduled for the same simulation clock time. These controls are similar in nature to the controls for port execution order, but on a larger scale. The model-wide Block Ranking dialog box enables you to specify tie-breaking priority rules to determine which block's events execute first. You open this dialog box by right-clicking the name of a model and selecting **Block Ranking**. The Block Ranking dialog box displays in a hierarchical check-box tree all blocks that have an adjustable rank in the selected model. A horizontal bar represents the current rank for each such block. Selecting the check box for a block enables you to drag its bar to adjust its rank. A longer horizontal bar corresponds to a higher rank. All ranks correspond to specific numeric values (and can also be specified numerically), but the rank values are evaluated only relatively for practical purposes. For example, if the rank of block A is higher than the rank of block B, the precise values involved are irrelevant; if block A and block B have events scheduled for the same simulation clock time, then the block A event will execute first.

- Controls on the allocation of resource units among resource entities have been expanded and improved. The Resource Scheduler block dialog box adds the **Adjust Units By** field, enabling you to choose the policy that is used to increase or decrease the units of resource entities when there is a scheduled adjustment. You can choose from among the following allocation policies:

  - **FairIntegerBased:** Any adjustments to the units of a resource entity are made on an integer basis, and any fractional resource units are ignored. If resource units increase or decrease, then those units are distributed integrally and as evenly as possible among all potential resource entities. This is the new default allocation policy.
  - **IntegerBased:** For a scheduled decrease in resource units, the units of the first targeted resource entity are integrally decreased as much as possible. If necessary, other resource entities are targeted if the first entity does not have enough units to cover the scheduled decrease. For an increase in resource units, this policy is the same as the **FairIntegerBased** policy.
  - **FractionBased:** For a scheduled increase in resource units, the additional units are distributed evenly among all targeted resource entities, potentially resulting in fractional units for some resource entities. For a scheduled decrease in resource units, the behavior is the same as for the **IntegerBased** policy, except that fractional units are also included. This was the default allocation policy in releases prior to SAS Simulation Studio 14.1.

- The SAS server on the local PC on which SAS Simulation Studio is installed is automatically launched on an as-needed basis. In previous releases, you were required to launch the local SAS server before reading a SAS data set, writing simulated data to a SAS data set, or running a SAS program. This enhancement ensures that the local SAS server will be available whenever it is required. However, if you specify that any block in your model should use a remote SAS workspace server, you are still required to start that server by using the Remote SAS Workspace Server Login dialog box, which appears during model execution.

Chapter 2
Using This Book

Contents

Purpose

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

This chapter describes the organization of this book and the conventions that are used in the text and example code. To gain full benefit from using this book, you should familiarize yourself with the information presented in this section and refer to it when needed. The section "Additional Documentation for SAS/OR Software" on page 8 refers to other documents that contain related information.

Organization

Chapter 3 describes the OPTLSO procedure. Chapter 4 describes the GA procedure. Each procedure description is self-contained; you need to be familiar with only the basic features of the SAS System and with SAS terminology to use it. The following list summarizes the types of information provided for each procedure:

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

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

Syntax constitutes the major reference section for the syntax of the procedure. First, the statement syntax is summarized. Next, a functional summary table lists all the statements and options in the procedure, classified by function. The PROC statement and other available statements are then described.

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

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

References lists references that are relevant to the chapter.

Typographical Conventions

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

roman	is the standard type style used for most text.

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

Conventions for Examples

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

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

Accessing the SAS/OR Sample Library

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

Online Documentation

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

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

Additional Documentation for SAS/OR Software

In addition to *SAS/OR User's Guide: Local Search Optimization*, you might find the following documents helpful when using SAS/OR software:

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

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

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

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

SAS/OR User's Guide: 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, the earned value management macros, the Microsoft Project conversion macros, and the PROJMAN application.

SAS Simulation Studio: User's Guide

provides documentation for 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
The OPTLSO Procedure

Contents

Overview: OPTLSO Procedure

The OPTLSO procedure performs optimization of general nonlinear functions that are defined by the FCMP procedure in Base SAS over both continuous and integer variables. These functions do not need to be expressed in analytic closed form, and they can be non-smooth, discontinuous, and computationally expensive to evaluate. Problem types can be single-objective or multiobjective. PROC OPTLSO runs in either single-machine mode or distributed mode.

NOTE: Distributed mode requires SAS High-Performance Optimization.

The general problem formulation is given by

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & x_\ell \le x \le x_u \\ & b_\ell \le Ax \le b_u \\ & c_\ell \le c(x) \le c_u \\ & x_i \in \mathbb{Z},\ i \in \mathcal{I} \end{aligned}$$

where $x \in \mathbb{R}^n$ is the vector of the decision variables; $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function; A is an $m \times n$ linear coefficient matrix; $c(x) : \mathbb{R}^n \to \mathbb{R}^p$ is the vector of general nonlinear constraint functions—that is, $c = (c_1, \ldots, c_p)$; x_ℓ and x_u are the vectors of the lower and upper bounds, respectively, on the decision variables; b_ℓ and b_u are the vectors of the lower and upper bounds, respectively, on the linear constraints; and c_ℓ and c_u are the vectors of the lower and upper bounds, respectively, on the nonlinear constraint functions. Equality constraints can be represented by equating the lower and upper bounds of the desired variable or constraint.

Because of the limited assumptions that are made on the objective function $f(x)$ and constraint functions $c(x)$, the OPTLSO procedure uses a parallel hybrid derivative-free approach similar to approaches that are used in Taddy et al. (2009); Plantenga (2009); Gray, Fowler, and Griffin (2010); Griffin and Kolda (2010a). Derivative-free methods are effective whether or not derivatives are available, provided that the dimension of x is not too large (Gray and Fowler 2011). As a rule of thumb, derivative-free algorithms are rarely applied to black-box optimization problems that have more than 100 variables. The term *black box* emphasizes that the function is used only as a mapping operator and makes no implicit assumption about or requirement on the structure of the functions themselves. In contrast, derivative-based algorithms commonly require the nonlinear objectives and constraints to be continuous and smooth and to have an exploitable analytic representation.

The OPTLSO procedure solves general nonlinear problems by simultaneously applying multiple instances of global and local search algorithms in parallel. This streamlines the process of needing to first apply a global algorithm in order to determine a good starting point to initialize a local algorithm. For example, if the problem is convex, a local algorithm should be sufficient, and the application of the global algorithm would create unnecessary overhead. If the problem instead has many local minima, failing to run a global search algorithm first could result in an inferior solution. Rather than attempting to guess which paradigm

is best, PROC OPTLSO simultaneously performs global and local searches while continuously sharing computational resources and function evaluations. The resulting run time and solution quality should be similar to having automatically selected the best global and local search combination, given a suitable number of threads and processors. Moreover, because information is shared, the robustness of the hybrid approach can be increased over hybrid combinations that simply use the output of one algorithm to hot-start the second algorithm. In this chapter, the term *solver* refers to an implementation of one or more algorithms that can be used to solve a problem.

The OPTLSO procedure uses different strategies to handle different types of constraints. Linear constraints are handled by using both linear programming and strategies similar to those in Griffin, Kolda, and Lewis (2008), where tangent directions to nearby constraints are constructed and used as search directions. Nonlinear constraints are handled by using smooth merit functions (Griffin and Kolda 2010b). Integer and categorical variables are handled by using strategies and concepts similar to those in Griffin et al. (2011). This approach can be viewed as a genetic algorithm that includes an additional "growth" step, in which selected points from the population are allotted a small fraction of the total evaluation budget to improve their fitness score (that is, the objective function value) by using local optimization over the continuous variables.

Because the OPTLSO procedure is a high-performance analytical procedure, it also does the following:

- enables you to run in distributed mode on a cluster of machines that distribute the data and the computations
- enables you to run in single-machine mode on the server where SAS is installed
- exploits all the available cores and concurrent threads, regardless of execution mode

For more information, see Chapter 4, "Shared Concepts and Topics" (*SAS/OR User's Guide: Mathematical Programming*), and Chapter 3, "Shared Concepts and Topics" (*Base SAS Procedures Guide: High-Performance Procedures*), for more information about the options available for the PERFORMANCE statement.

Getting Started: OPTLSO Procedure

All nonlinear objective and constraint functions should be defined by using the FCMP procedure. In PROC FCMP, you specify the objective and constraint functions by using SAS statements that are similar to syntax that is used in the SAS DATA step; these functions are then compiled into function libraries for subsequent use. The SAS CMPLIB= system option specifies where to look for previously compiled functions and subroutines. All procedures (including PROC FCMP) that support the use of FCMP functions and subroutines use this system option. After your objective and constraint functions have been specified in a library, PROC OPTLSO requires the names and context of the functions within this library that are relevant to the current optimization problem. You can provide this information to PROC OPTLSO by using SAS data sets. You use additional data sets to describe variables and linear constraints in either a sparse or a dense format.

Introductory Examples

The following introductory examples illustrate how to get started using the OPTLSO procedure.

A Bound-Constrained Problem

Consider the simple example of minimizing the Branin function,

$$f(x) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10$$

subject to $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$ (Jones, Perttunen, and Stuckman 1993). The minimum function value is $f(x^*) = 0.397887$ at $x^* = (-\pi, 12.275), (\pi, 2.275), (9.42478, 2.475)$. You can use the following statements to solve this problem:

```
data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
x1 -5 10
x2  0 15
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function branin(x1, x2);
      pi = constant('PI');
      y1 = (x2-(5.1/(4*pi**2))*x1*x1+5*x1/pi-6)**2;
      y2 = 10*(1-1/(8*pi))*cos(x1);
      return (y1+y2+10);
   endsub;
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f branin min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
   performance nthreads=4;
run;

proc print data=solution;
run;
```

The OBJECTIVE= option in the PROC OPTLSO statement refers to the OBJDATA data set, which identifies BRANIN as the name of the objective function that is defined in the FCMP library sasuser.myfuncs and specifies that BRANIN should be minimized. The VARIABLES= option in the PROC OPTLSO statement

names the decision variables x1 and x2 and specifies lower and upper bounds. The PERFORMANCE statement specifies the number of threads that PROC OPTLSO can use.

Figure 3.1 shows the ODS tables that the OPTLSO procedure produces by default.

Figure 3.1 Bound-Constrained Problem: Output

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	2
Integer Variables	0
Continuous Variables	2
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	0.3978873689
Infeasibility	0
Iterations	32
Evaluations	1805
Cached Evaluations	36
Global Searches	1
Population Size	80
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	0.39789
2	0	_inf_	0.00000
3	0	x1	9.42482
4	0	x2	2.47508
5	0	f	0.39789

Adding a Linear Constraint

You can use a LINCON= option to specify general linear equality or inequality constraints of the following form:

$$\sum_{j=1}^{n} a_{ij} x_j \{\leq | = | \geq\} b_i \quad \text{for } i = 1, \ldots, m$$

For example, suppose that in addition to the bound constraints on the decision variables, you need to guarantee that the sum $x_1 + x_2$ is less than or equal to 0.6. To guarantee this, you can add a LINCON= option to the previous data set definitions, as in the following statements:

```
data lindata;
   input _id_ $ _lb_ x1 x2 _ub_;
   datalines;
a1  . 1 1  0.6
;

proc optlso
   variables = vardata
   objective = objdata
   lincon    = lindata;
run;
```

Here the symbol A1 denotes the name of the given linear constraints that are specified in the _ID_ column. The corresponding lower and upper bounds are specified in the _LB_ and _UB_ columns, respectively.

Nonlinear Constraints on the Decision Variables

You can specify general nonlinear equality or inequality constraints by using the NLINCON= option and adding its definition to the existing PROC FCMP library. Consider the previous problem with the following additional constraint:

$$x_1^2 - 2x_2 \geq 14$$

You can specify this constraint by adding a new FCMP function library and providing it a corresponding name and bounds in the NLINCON= option, as in the following statements:

```
data condata;
   input _id_ $ _lb_ _ub_;
   datalines;
c1  14 .
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function c1(x1, x2);
      return (x1**2 - 2*x2);
   endsub;
run;

proc optlso
   variables = vardata
   objective = objdata
   lincon    = lindata
   nlincon   = condata;
run;
```

By calling PROC FCMP a second time, you can append the definition of C1 to your existing user library. That is, you do not need to redefine BRANIN after it has been added.

A Simple Maximum Likelihood Example

The following is a very simple example of a maximum likelihood estimation problem that uses the log-likelihood function:

$$l(\mu,\sigma) = -\log(\sigma) - \frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

The maximum likelihood estimates of the parameters μ and σ form the solution to

$$\max_{\mu,\sigma} \sum_i l_i(\mu,\sigma)$$

where $\sigma > 0$ and

$$l_i(\mu,\sigma) = -\log(\sigma) - \frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2$$

The following sets of statements demonstrate two ways to formulate this example problem:

```
data lkhvar;
   input _id_ $ _lb_;
   datalines;
mu    .
sigma 0
;

data lkhobj1;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f loglkh1 max
;
```

In the following statements, the FCMP function is “stand-alone” because all the necessary data are defined within the function itself:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function loglkh1(mu, sigma);
      array x[5] / nosym (1 3 4 5 7);
      s=0;
      do j=1 to 5;
         s = s - log(sigma) - 0.5*((x[j]-mu)/sigma)**2;
      end;
      return (s);
   endsub;
run;

proc optlso
   variables = lkhvar
   objective = lkhobj1;
run;
```

Alternatively, you can use an external data set to store the necessary observations and run PROC OPTLSO to feed each observation to an FCMP function that processes a single line of data. In this case, PROC OPTLSO sums the results for you. This mode is particularly useful when the data set is large and possibly distributed over a set of nodes, as shown in "Example 3.7: Using External Data Sets" on page 57.

The following statements demonstrate how to store the necessary observations in an external data set for use with PROC FCMP:

```
data logdata;
   input x @@;
   datalines;
1 3 4 5 7
;

data lkhobj2;
   input _id_ $ _function_ $ _sense_ $ _dataset_ $;
   datalines;
f loglkh2 max logdata
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function loglkh2(mu, sigma, x);
      return (-log(sigma) -0.5*((x-mu)/sigma)**2);
   endsub;
run;

proc optlso
   variables = lkhvar
   objective = lkhobj2;
run;
```

In this case, for each line of data in the data set logdata, the FCMP function LOGLKH2 is called. It is important that the non-variable arguments of LOGLKH2 coincide with a subset of the column names in logdata, in this case X. However, the order in which the variables and data column names appear is not important. The following definition would work as well:

```
data lkhobj3;
   input _id_ $ _function_ $ _sense_ $ _dataset_ $;
   datalines;
f loglkh3 max logdata
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function loglkh3(x, sigma, mu);
      return (- log(sigma) - 0.5*((x-mu)/sigma)**2);
   endsub;
run;

proc optlso
   variables = lkhvar
   objective = lkhobj3;
run;
```

Syntax: OPTLSO Procedure

The following statements are available in the OPTLSO procedure:

PROC OPTLSO *<options>* **;**
READARRAY *SAS-data-set-1 <SAS-data-set-2 . . . SAS-data-set-k>* **;**
PERFORMANCE *<options>* **;**

PROC OPTLSO Statement

PROC OPTLSO *<options>* **;**

The PROC OPTLSO statement invokes the OPTLSO procedure.

Functional Summary

Table 3.1 outlines the *options* available in the PROC OPTLSO statement.

Table 3.1 Summary of PROC OPTLSO Options

Description	*option*
Input Data Set Options	
Specifies the input cache file	CACHEIN=
Specifies the initial genetic algorithm population	FIRSTGEN=
Describes the linear constraints	LINCON=
Indicates that the description uses the mathematical programming system (MPS) data format	MPSDATA=
Describes the nonlinear constraints	NLINCON=
Describes the objective function	OBJECTIVE=
Specifies the initial trial points	PRIMALIN=
Indicates that the description uses the quadratic programming system (QPS) data format	QPSDATA=
Describes the variables	VARIABLES=
Output Data Set Options	
Specifies the output cache file	CACHEOUT=
Specifies the local solutions and best feasible solution	PRIMALOUT=
Specifies the members of the genetic algorithm population on exit	LASTGEN=
Stopping Condition Options	
Specifies the absolute function convergence criterion	ABSFCONV=
Specifies the maximum number of function evaluations	MAXFUNC=
Specifies the maximum number of genetic algorithm iterations	MAXGEN=
Specifies the upper limit on real time	MAXTIME=

Description	*option*
Optimization Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the number of global searches	NGLOBAL=
Specifies the number of local searches	NLOCAL=
Genetic algorithm population size	POPSIZE=
Technical Options	
Specifies the cache tolerance	CACHETOL=
Specifies the maximum cache size	CACHEMAX=
Specifies the maximum size of the Pareto-optimal set	PARETOMAX=
Specifies how frequently to print the solution progress	LOGFREQ=
Specifies how much detail of solution progress to print in the log	LOGLEVEL=
Enables or disables the printing summary	PRINTLEVEL=
Initializes the seed for sampling routines	SEED=

Input Data Set Options

You can specify the following *options* that are related to input data sets:

CACHEIN=*SAS-data-set*
names a previously computed sample set. Using a previously computed sample set enables PROC OPTLSO to warm-start. It is crucial that the nonlinear objective and function values be identical to those that were used when the cache data set was generated. For more information, see the section "Specifying and Returning Trial Points" on page 36.

FIRSTGEN=*SAS-data-set*
specifies an initial sample set that defines a subset of the initial population. The columns of this data set should coincide with the same format that is used by the PRIMALIN= data set. If the population size p is smaller than the size of this set, only the first p points of this set are used. For more information, see the section "Specifying and Returning Trial Points" on page 36.

LINCON=*SAS-data-set*
uses a dense format to describe the optimization problem's linear constraints. The corresponding data set should have columns _LB_ and _UB_ to describe lower and upper bounds, respectively. The column _ID_ is reserved for the corresponding constraint name. The remaining columns must correspond to the linear coefficients of the variables that are listed in the VARIABLES= option. For more information, see the section "Describing Linear Constraints" on page 31.

MPSDATA=*SAS-data-set*
specifies a data set that can be used as a sparse alternative to the LINCON= option, which uses a dense format to define variables. Mathematical programming system (MPS) is a common file format for representing linear and mixed-integer mathematical programs. For an example of using the OPTMODEL procedure to create the corresponding MPS file, see "Example 3.2: Using MPS Format" on page 44. Internally, binary variables are converted into integer variables with lower and upper bounds of 0 and 1, respectively. For more information, see the section "Describing Linear Constraints" on page 31.

NLINCON=*SAS-data-set*
names the FCMP functions to be used from the current library as nonlinear constraints, along with respective bounds. This data set should contain three columns: _ID_ to specify the corresponding FCMP function names and _LB_ and _UB_ to specify the lower and upper bounds for the corresponding constraints, respectively. For more information, see the section "Describing Nonlinear Constraints" on page 32.

OBJECTIVE=*SAS-data-set*
names the FCMP functions to be used from the current library to form the objective. At a minimum, this data set should have three columns: _ID_ to specify the function name to be used internally by the solver, _FUNCTION_ to specify the corresponding FCMP function, and _SENSE_ to specify whether the objective is to be minimized or maximized. PROC OPTLSO enables you to implicitly define your objective function by using an external data set and an intermediate FCMP function definition that can be used as placeholders to store temporary terms with respect to the external data set. For more information, see the section "Describing the Objective Function" on page 26.

PRIMALIN=*SAS-data-set*
specifies an initial sample set to be evaluated. Initial data sets might be useful over a sequence of runs when you want to ensure that PROC OPTLSO generates points that are at least as good as your current best solution. This option is more general than the FIRSTGEN= option because points that are defined in this data set might or might not be used to define the initial population for the genetic algorithm (GA). For more information, see the section "Specifying and Returning Trial Points" on page 36.

QPSDATA=*SAS-data-set*
specifies a data set that can be used as a sparse alternative to the LINCON= option, which uses a dense format to define variables. Quadratic programming system (QPS) is a common file format for representing linear and mixed-integer mathematical programs that have quadratic terms in the objective function. This option differs from the MPSDATA= option in that any quadratic terms in the objective can be included in the data set. Do not use this option if the problem does not have quadratic terms. For an example of using PROC OPTMODEL to create the corresponding QPS file, see "Example 3.3: Using QPS Format" on page 47. Internally, binary variables are converted into integer variables with lower and upper bounds of 0 and 1, respectively.

VARIABLES=*SAS-data-set*
stores the variable names, bounds, type, and scale. These names must match corresponding names, FCMP functions, and related data sets. For more information, see the section "The Variable Data Set" on page 25.

Output Data Set Options

You can specify the following *options* that are related to output data sets:

CACHEOUT=*SAS-data-set*
specifies the data set to which all completed function evaluations are output. For more information, see the section "Specifying and Returning Trial Points" on page 36.

LASTGEN=*SAS-data-set*
specifies the data set to which the members of the current genetic algorithm population are returned on exit. If more than one genetic algorithm is used, the data set combines the members from each population into a single data set.

PRIMALOUT=*SAS-data-set*
specifies the output solution set. You can use this data set in future solves as the input set for the PRIMALIN= option. For more information, see the section "Specifying and Returning Trial Points" on page 36.

Stopping Condition Options

You can specify the following *options* that determine when to stop optimization:

ABSFCONV=*r*[*n*]
specifies an absolute function convergence criterion. PROC OPTLSO stops when the changes in the objective function and constraint violation values in successive iterations meet the criterion

$$|f(x^{(k-1)}) - f(x^{(k)})| + |\theta(x^{(k-1)}) - \theta(x^{(k)})| \leq r$$

where $\theta(x)$ denotes the maximum constraint violation at point x. The optional integer value *n* specifies the number of successive iterations for which the criterion must be satisfied before the process is terminated. The default is *r*=1E–6 and *n*=10. To cause an early exit, you must specify a value for *n* that is less than the value of the MAXGEN= option.

MAXFUNC=*i*
specifies the maximum number of function calls in the optimization process. The actual number of function evaluations can exceed this number in order to ensure deterministic results.

MAXGEN=*i*
specifies the maximum number of genetic algorithm iterations. The default is 500.

MAXTIME=*r*
specifies an upper limit in seconds on the real time used in the optimization process. The actual running time of PROC OPTLSO optimization might be longer because the actual running time includes the remaining time needed to finish current function evaluations, the time for the output of the (temporary) results, and (if required) the time for saving the results to appropriate data sets. By default, the MAXTIME= option is not used.

Optimization Control Options

You can specify the following *options* to tailor the solver to your specific optimization problem:

FEASTOL=*r*
specifies a feasibility tolerance for provided constraints. Specify *r*≥1E–9. The default is *r*=1E–3.

NGLOBAL=*i*
specifies the number of genetic algorithms to create, with each algorithm working on a separate population of the size specified by the POPSIZE= option. Specify *i* as an integer greater than 0.

NLOCAL=*i*
specifies the number of local solvers to create. Specify *i* as an integer greater than 0. The default is twice the number of variables in the problem.

POPSIZE=*i*

specifies the population size for the genetic algorithm to use. The default is $40 \times \text{ceil}(\log(n) + 1)$, where n denotes the number of variables.

Technical Options

You can specify the following technical *options*:

CACHEMAX=*i*

specifies the maximum number of points that can be cached. By default, PROC OPTLSO automatically calculates the maximum number of points.

PARETOMAX=*i*

specifies the maximum number of points in the Pareto-optimal set. The default is 5,000.

CACHETOL=*r*

specifies the cache tolerance to be used for caching and referencing previously evaluated points. For more information about this tolerance, see the section "Function Value Caching" on page 37. The value of *r* can be any number in the interval [0, 1]. The default is 1E–9.

LOGFREQ=*i*

requests that the solution progress be printed to the iteration log after every *i* iterations if the value of the LOGLEVEL= option is greater than or equal to 0. The value *i*=0 disables the printing of the solution progress. The final iteration is always printed if $i \geq 1$ and LOGLEVEL is nonzero. The default is 1.

LOGLEVEL=0 | 1

controls how much information is printed to the log file. If LOGLEVEL=0, nothing is printed. If LOGLEVEL=1, a short summary of the problem description and final solution status is printed. If LOGLEVEL=0, this option overrides the LOGFREQ= option. By default, LOGLEVEL=1.

PRINTLEVEL=0 | 1

specifies whether to print a summary of the problem and solution. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=0, then no ODS tables are produced. By default, PRINTLEVEL=1.

For more information about the ODS tables that are created by PROC OPTLSO, see the section "ODS Tables" on page 39.

SEED=*i*

specifies a nonnegative integer as a seed value for the pseudorandom number generator. Pseudorandom numbers are used within the genetic algorithm.

PERFORMANCE Statement

PERFORMANCE *<options>* **;**

The PERFORMANCE statement defines performance parameters for multithreaded and distributed computing, passes variables that describe the distributed computing environment, and requests detailed results

about the performance characteristics of a SAS high-performance analytics procedure. For more information about the options available for the PERFORMANCE statement, see Chapter 4, "Shared Concepts and Topics" (*SAS/OR User's Guide: Mathematical Programming*), and Chapter 3, "Shared Concepts and Topics" (*Base SAS Procedures Guide: High-Performance Procedures*).

Note that the SAS High-Performance Optimization license is required to invoke PROC OPTLSO in distributed mode. For examples of running in distributed mode see the third program in "Example 3.7: Using External Data Sets" on page 57 and "Example 3.8: Johnson's Systems of Distributions" on page 63.

READARRAY Statement

READARRAY *SAS-data-set-1 <SAS-data-set-2 . . . SAS-data-set-k>* **;**

PROC FCMP (see "The FCMP Procedure" on page 25) provides the READ_ARRAY function to read data from a SAS data set into array variables. In order to ensure that the referenced data sets are available, PROC OPTLSO also requires that the names of these data sets be provided as a list of names in the READARRAY statement. For an example, see the second program in "Example 3.7: Using External Data Sets" on page 57. The following example creates and reads a SAS data set into an FCMP array variable:

```
data barddata;
   input y @@;
   datalines;
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
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3);
      array y[15] / nosym;
         rc = read_array('barddata', y);
         fx = 0;
         do k=1 to 15;
            dk  = (16-k)*x2 + min(k,16-k)*x3;
            fxk = y[k] - (x1 + k/dk);
            fx = fx + fxk**2;
         end;
      return (0.5*fx);
   endsub;
run;

options cmplib = sasuser.myfuncs;
data _null_;
   bval = bard(1,2,3);
   put bval=;
run;
```

Here the call to READ_ARRAY in the PROC FCMP function definition of Bard populates the array Y with the rows of the BardData data set. If the Bard function were subsequently used in a problem definition for PROC OPTLSO, the BardData data set should be listed in the corresponding READARRAY statement of PROC OPTLSO as demonstrated in the second program in "Example 3.7: Using External Data Sets" on page 57.

Details: OPTLSO Procedure

The OPTLSO procedure uses a hybrid combination of genetic algorithms (Goldberg 1989; Holland 1975), which optimize integer and continuous variables, and generating set search (Kolda, Lewis, and Torczon 2003), which performs local search on the continuous variables to improve the robustness of both algorithms. Both genetic algorithms (GAs) and the generating set search (GSS) have proven to be effective algorithms for many classes of derivative-free optimization problems. When only continuous variables are present, a GA usually requires more function evaluations than a GSS to converge to a minimum. This is partly due to the GA's need to simultaneously perform a global search of the solution space. In a hybrid setting, the requirement for the GA to find accurate local minima can be relaxed, and internal parameters can be tuned toward finding promising starting points for the GSS.

The FCMP Procedure

The FCMP procedure is part of Base SAS software and is the primary mechanism in SAS for creating user-defined functions to be used in a DATA step and many SAS procedures. You can use most of the SAS programming statements and SAS functions that you can use in a DATA step to define FCMP functions and subroutines. The OPTLSO procedure also uses PROC FCMP to provide a general gateway for you to describe objective and constraint functions. For more information, see the sections "Describing the Objective Function" on page 26 and "Describing Nonlinear Constraints" on page 32), respectively.

However, there are a few differences between the capabilities of the DATA step and the FCMP procedure. For more information, see the documentation about the FCMP procedure in *Base SAS Procedures Guide*. Further, not all PROC FCMP functionality is currently compatible with PROC OPTLSO; in particular, the following FCMP functions are not supported and should not be called within your FCMP function definitions: WRITE_ARRAY, RUN_MACRO, and RUN_SASFILE. The READ_ARRAY function is supported; however, the corresponding data sets that are used must be listed in a separate statement of PROC OPTLSO (see the section "READARRAY Statement" on page 24 and the second program in "Example 3.7: Using External Data Sets" on page 57). After you define your objective and constraint functions, you must specify the libraries by using the CMPLIB= system option in the OPTIONS statement. For more information about the OPTIONS statement, see *SAS Statements: Reference*. For more information about the CMPLIB= system option, see *SAS System Options: Reference*.

The Variable Data Set

The variable data set can have up to five columns. The variable names are described in the _ID_ column. These names are used to map variable values to identically named FCMP function variable arguments. Lower and upper bounds on the variables are defined in columns _LB_ and _UB_, respectively. You can manually enter scaling for each variable by using the _SCALE_ column. Derivative-free optimization performance can often be greatly improved by scaling the variables appropriately. By default, the scale vector is defined in a manner similar to that described in Griffin, Kolda, and Lewis (2008). If integer variables are present, the _TYPE_ column value signifies that a given variable is either `C` for continuous or `I` for integer. By default, all variables are assumed to be continuous.

You can use the VARIABLES= option to specify the SAS data set that describes the variables to be optimized. For example, suppose you want to specify the following set of variables, where x_1 and x_2 are continuous and x_3 is an integer variable:

$$\begin{aligned} 0 &\le x_1 \le 1000 \\ 0 &\le x_2 \le 0.001 \\ 0 &\le x_3 \le 4 \end{aligned}$$

You can specify this set of variables by using the following DATA step:

```
data vardata;
   input _id_ $ _lb_ _ub_ _type_ $;
   datalines;
x1  0 1000  C
x2  0 0.001 C
x3  0 4     I
;
```

By default, variables are automatically scaled. PROC OPTLSO uses this scaling along with the cache tolerance when PROC OPTLSO builds the function value cache. For more information about scaling, see the section "Function Value Caching" on page 37. You can provide your own scaling by setting a _SCALE_ column in the variable data set as follows:

```
data vardata;
   input _id_ $ _lb_ _ub_ _type_ $ _scale_;
   datalines;
x1  0 1000  C  1000
x2  0 0.001 C  0.5
x3  0 4     I  2
;
```

When derivative-free optimization is performed, proper scaling of variables can dramatically improve performance. Default scaling takes into account the lower and upper bounds, so you are encouraged to provide the best estimates possible.

Describing the Objective Function

PROC OPTLSO enables you to define the function explicitly by using PROC FCMP. The following statements describe a PROC FCMP objective function that is used in "Example 3.5: Linear Constraints and a Nonlinear Objective" on page 52:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function sixhump(x1,x2);
      return ((4 - 2.1*x1**2 + x1**4/3)*x1**2
              + x1*x2 + (-4 + 4*x2**2)*x2**2);
   endsub;
run;
```

Because PROC FCMP writes to an external library that might contain a large number of functions, PROC OPTLSO needs to know which objective function in the FCMP library to use and whether to minimize or maximize the function. You provide this information to PROC OPTLSO by specifying an objective

description data set in the OBJECTIVE= option. To minimize the function sixhump, you could specify the OBJECTIVE= objdata option and define the following objective description data set:

```
data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   sixhump     min
;
```

In the preceding DATA step, the _ID_ column specifies the function name to be used internally by the solver, the _FUNCTION_ column specifies the corresponding FCMP function name, and the _SENSE_ column specifies whether the objective is to be minimized or maximized.

Intermediate Functions

You can use intermediate functions to simplify the objective function definition and to improve computational efficiency. You specify intermediate functions by using a missing value entry in the _SENSE_ column to denote that the new function is not an objective. The _ID_ column entries for intermediate functions can then be used as arguments for the objective function. The following set of programming statements demonstrates how to create an equivalent objective definition for "Example 3.5: Linear Constraints and a Nonlinear Objective" on page 52 by using intermediate functions.

```
data objdata;
   length _function_ $10;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f1   sixhump1       .
f2   sixhump2       .
f3   sixhumpNew     min
;
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function sixhump1(x1,x2);
      return (4 - 2.1*x1**2 + x1**4/3);
   endsub;
   function sixhump2(x1,x2);
      return (-4 + 4*x2**2);
   endsub;
   function sixhumpNew(x1,x2,f1,f2);
      return (f1*x1**2 + x1*x2 + f2*x2**2);
   endsub;
run;
```

In this case, PROC OPTLSO first computes the values for sixhump1 and sixhump2, internally assigning the output to f1 and f2, respectively. The _ID_ column entries for intermediate functions can then be used as arguments for the objective function f3. Because the intermediate functions are evaluated first, before the objective function is evaluated, intermediate functions should never depend on output from the objective function.

Incorporating MPS and QPS Objective Functions

If you use the MPSDATA= or QPSDATA= options to define linear constraints, an objective function $m(x)$ is necessarily defined (see "Describing Linear Constraints" on page 31). If you do not specify the OBJECTIVE= option, PROC OPTLSO optimizes $m(x)$. However, if you specify the OBJECTIVE= option, the objective

function $m(x)$ is ignored unless you explicitly include the corresponding objective function name and specify whether the function is to be used as an intermediate or objective function. (See "Example 3.4: Combining MPS and FCMP Function Definitions" on page 49.) If the objective name also matches a name in the FCMP library, the FCMP function definition takes precedence.

Using Large Data Sets

When your objective function includes the sum of a family of functions that are parameterized by the rows of a given data set, PROC OPTLSO enables you to include in your objective function definition a single external data set that is specified in the _DATASET_ column of the OBJECTIVE= data set. Consider the following unconstrained optimization problem, where k is a very large integer (for example, 10^6), $\mathbf{A}$ denotes a $k \times 5$ matrix, and $\mathbf{b}$ denotes the corresponding right-hand-side target vector:

$$\min_{x \in \mathbb{R}^5} f(x) = \|Ax - b\|_2 + \|x\|_1$$

To evaluate the objective function, the following operations must be performed:

$$\begin{aligned} f_1(x) &= \sum_{i=1}^{k} d_i \\ f_2(x) &= \sum_{j=1}^{5} |x_i| \\ f(x) &= \sqrt{f_1(x)} + f_2(x) \end{aligned}$$

where

$$d_i = \left(-b_i + \sum_{j=1}^{5} a_{ij} x_j \right)^2$$

and a_{ij} denotes the jth entry of row i of the matrix $\mathbf{A}$. Assume that there is an existing SAS data set that stores numerical entries for $\mathbf{A}$ and $\mathbf{b}$. The following DATA step shows an example data set, where $k = 3$:

```
data Abdata;
   input _id_ $ a1 a2 a3 a4 a5 b;
   datalines;
row1  1  2  3  4  5  6
row2  7  8  9  10 11 12
row3  13 14 15 16 17 18
;
```

The following statements pass this information to PROC OPTLSO by adding the corresponding functions to the FCMP function library Sasuser.Myfuncs.Mypkg:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function axbi(x1,x2,x3,x4,x5,a1,a2,a3,a4,a5,b);
      array x[5];
      array a[5];
      di = -b;
      do j=1 to 5;
```

```
         di = di + a[j]*x[j];
      end;
      return (di*di);
   endsub;

   function onenorm(x1,x2,x3,x4,x5);
      array x[5];
      f2 = 0;
      do j=1 to 5;
         f2 = f2 + abs(x[j]);
      end;
      return (f2);
   endsub;

   function combine(f1, f2);
      return (sqrt(f1)+f2);
   endsub;
run;
```

The next DATA step then defines the objective name with a given target:

```
data lsqobj1;
   input _id_ $ _function_$ _sense_ $ _dataset_ $;
   datalines;
f1        axbi          .      Abdata
f2        onenorm       .      .
f         combine       min    .
;
```

The following DATA step declares the variables:

```
data xvar;
   input _id_ $ @@;
   datalines;
   x1 x2 x3 x4 x5
   ;
```

The following statements call the OPTLSO procedure:

```
options cmplib=sasuser.myfuncs;
proc optlso
   variables = xvar
   objective = lsqobj1;
run;
```

The contents of the OBJECTIVE= data set (lsqobj1) direct PROC OPTLSO to search for the three FCMP functions AXBI, ONENORM, and COMBINE in the library that is specified by the CMPLIB= option. The missing values in the _SENSE_ column indicate that AXBI and ONENORM are intermediate functions to be used as arguments of the objective function COMBINE, which is of type MIN. Of the three FCMP functions, only F1 has requested data. The entry Abdata specifies that the FCMP function AXBI should be called on each row of the data set Abdata and that the results should be summed. This value is then specified as the first argument to the FCMP function COMBINE.

In this example, Abdata is a data set that comes from the Work library. However, Abdata could just as easily come from a data set in a user-defined library or even a data set that had been previously distributed

(for example, to a Teradata or Greenplum database). If the data set Abdata is stored in a different library, replace **Abdata** with **libref.Abdata** in the data set lsqob1. The source of the data set is irrelevant to PROC OPTLSO.

You can omit F2 if you want to form the one-norm directly in the aggregate function. Thus, an equivalent formulation would be as follows:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function combine2(x1,x2,x3,x4,x5, f1);
      array x[5];
      f2 = 0;
      do j=1 to 5;
         f2 = f2 + abs(x[j]);
      end;
      return (sqrt(f1)+f2);
   endsub;
run;
```

In this case, you define the objective name with a given target in a data set:

```
data lsqobj2;
   input _id_ $ _function_$ _sense_ $ _dataset_ $;
   datalines;
f1        axbi          .          Abdata
f         combine2     min         .
;

options cmplib=sasuser.myfuncs;
proc optlso
   variables = xvar
   objective = lsqobj2;
run;
```

Thus, any of the intermediate functions that are used within the OBJECTIVE= data set (objdata) are permitted to have arguments that form a subset of the variables listed in the VARIABLES= data set (xvar) and the numerical columns from the data set that is specified in the _DATASET_ column of the OBJECTIVE= data set. Only numerical values are supported from external data sets. Only one function can be of type MIN or MAX. This function can take as arguments any of the variables in the VARIABLES= data set, any of the numerical columns from an external data set for the objective (if specified), and any implicit variables that are listed in the _ID_ column of the OBJECTIVE= data set.

The following rules for the objective data set are used during parsing:

- Only one data set can be used for a given problem definition.
- The objective function can take a data set as input only if no intermediate functions are being used. Otherwise, only the intermediate functions can be linked to the corresponding data set.

The data set is used in a distributed format if either the NODES= option is specified in the PERFORMANCE statement or the data set is a distributed library.

Describing Linear Constraints

The preferred method for describing linear constraints is to use the LINCON=, MPSDATA=, or QPSDATA= option. You should not describe linear constraints by using the NLINCON= option because they are treated as black-box constraints and can degrade performance.

If you have only a few variables and linear constraints, you might prefer to use the dense format to describe your linear constraints by specifying both the LINCON= option and the VARIABLES= option. Suppose a problem has the following linear constraints:

$$\begin{aligned} -1 \leq \quad & 2x_1 - 4x_3 \quad && \leq 15 \\ 1 \leq \quad & -3x_1 + 7x_2 \quad && \leq 13 \\ & x_1 + x_2 + x_3 \quad && = 11 \end{aligned}$$

The following DATA step formulates this information as an input data set for PROC OPTLSO:

```
data lcondata;
   input _id_ $ _lb_ x1-x3 _ub_;
   datalines;
a1 -1    2 0 -4 15
a2  1   -3 7  0 13
a3  11   1 1  1 11
;
```

Linear constraints are handled by using tangent search directions that are defined with respect to nearby constraints. The metric that is used to determine when a constraint is sufficiently near might cause the algorithm to temporarily treat range constraints as equality constraints. For this reason, it is preferable that you not separate a range constraint into two inequality constraints. For example, although both of the following range constraint formulations are equivalent, the former is preferred:

$$-1 \leq \quad 2x_1 - 4x_3 \quad \leq 15$$

and

$$\begin{aligned} & 2x_1 - 4x_3 \quad && \leq 15 \\ -1 \leq \quad & 2x_1 - 4x_3 \end{aligned}$$

Even for a moderate number of variables and constraints, using the dense format to describe the linear constraints can be burdensome. As an alternative, you can construct a sparse formulation by using MPS-format (or QPS-format) SAS data sets and specifying the MPSDATA= (or QPSDATA=) option in PROC OPTLSO. For more information about these data sets, see Chapter 17, "The MPS-Format SAS Data Set" (*SAS/OR User's Guide: Mathematical Programming*). You can easily create these data sets by using the OPTMODEL procedure, as demonstrated in "Example 3.2: Using MPS Format" on page 44 and "Example 3.3: Using QPS Format" on page 47. For more information about using PROC OPTMODEL to create MPS data sets, see Chapter 5, "The OPTMODEL Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

Both MPS and QPS data sets require that you define an objective function. Although the QPS standard supports a constant term, the MPS standard does not; thus any constant term that is defined in PROC OPTMODEL is naturally dropped and not included in the corresponding data set. In particular, if the MPSDATA= option is used, the corresponding objective will have the form

$$m(x) = c^T x$$

However, if the QPSDATA= option is used, the corresponding objective will have the form

$$m(x) = c^T x + \frac{1}{2} x^T Q x + K$$

where c, Q, and the constant term K are defined within the provided data sets. Although multiple objectives might be listed, PROC OPTLSO uses only the first objective from the MPSDATA= or QPSDATA= option. How the corresponding objective function is used is determined by the contents of the OBJECTIVE= data set. For more information, see "Incorporating MPS and QPS Objective Functions" on page 27.

Describing Nonlinear Constraints

Nonlinear constraints are treated as black-box functions and are described by using the FCMP procedure. You should use the NLINCON= option to provide PROC OPTLSO with the corresponding FCMP function names and lower and upper bounds. Suppose a problem has the following nonlinear constraints:

$$\begin{aligned} -1 \leq \quad & x_1 x_2 x_3 + \sin(x_2) && \leq 1 \\ & x_1 x_2 + x_1 x_3 + x_2 x_3 && = 1 \end{aligned}$$

The following statements pass this information to PROC OPTLSO by adding two corresponding functions to the FCMP function library Sasuser.Myfuncs.Mypkg:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function con1(x1, x2, x3);
      return (x1*x2*x3 + sin(x2));
   endsub;
   function con2(x1, x2, x3);
      return (x1*x2 + x1*x3 + x3*x3);
   endsub;
run;
```

Next, the following DATA step defines nonlinear constraint names and provides their corresponding bounds in the data set condata:

```
data condata;
   input _id_ $ _lb_ _ub_;
   datalines;
con1 -1 1
con2  1 1
;
```

Finally, you can call PROC OPTLSO and specify NLINCON=CONDATA. For another example with nonlinear constraints, see "Example 3.6: Using Nonlinear Constraints" on page 54.

The OPTLSO Algorithm

The OPTLSO algorithm is based on a genetic algorithm (GA). GAs are a family of local search algorithms that seek optimal solutions to problems by applying the principles of natural selection and evolution. Genetic algorithms can be applied to almost any optimization problem and are especially useful for problems for which other calculus-based techniques do not work, such as when the objective function has many local

optima, when the objective function is not differentiable or continuous, or when solution elements are constrained to be integers or sequences. In most cases, genetic algorithms require more computation than specialized techniques that take advantage of specific problem structures or characteristics. However, for optimization problems for which no such techniques are available, genetic algorithms provide a robust general method of solution.

In general, genetic algorithms use some variation of the following process to search for an optimal solution:

initialization: An initial population of solutions is randomly generated, and the objective function is evaluated for each member of this initial iteration.

selection: Individual members of the current iteration are chosen stochastically either to parent the next iteration or to be passed on to it such that the members that are the fittest are more likely to be selected. A solution's fitness is based on its objective value, with better objective values reflecting greater fitness.

crossover: Some of the selected solutions are passed to a crossover operator. The crossover operator combines two or more parents to produce new offspring solutions for the next iteration. The crossover operator tends to produce new offspring that retain the common characteristics of the parent solutions while combining the other traits in new ways. In this way, new areas of the search space are explored while attempting to retain optimal solution characteristics.

mutation: Some of the next-iteration solutions are passed to a mutation operator, which introduces random variations in the solutions. The purpose of the mutation operator is to ensure that the solution space is adequately searched to prevent premature convergence to a local optimum.

repeat: The current iteration of solutions is replaced by the new iteration. If the stopping criterion is not satisfied, the process returns to the selection phase.

The crossover and mutation operators are commonly called *genetic operators*. Selection and crossover distinguish genetic algorithms from a purely random search and direct the algorithm toward finding an optimum.

There are many ways to implement the general strategy just outlined, and it is also possible to combine the genetic algorithm approach with other heuristic solution improvement techniques. In the traditional genetic algorithm, the solution space is composed of bit strings, which are mapped to an objective function, and the genetic operators are modeled after biological processes. Although there is a theoretical foundation for the convergence of genetic algorithms that are formulated in this way, in practice most problems do not fit naturally into this paradigm. Modern research has shown that optimizations can be set up by using the natural solution domain (for example, a real vector or integer sequence) and by applying crossover and mutation operators that are analogous to the traditional genetic operators but are more appropriate to the natural formulation of the problem.

Although genetic algorithms have been demonstrated to work well for a variety of problems, there is no guarantee of convergence to a global optimum. Also, the convergence of genetic algorithms can be

sensitive to the choice of genetic operators, mutation probability, and selection criteria, so that some initial experimentation and fine-tuning of these parameters are often required.

The OPTLSO procedure seeks to automate this process by using hybrid optimization in which several genetic algorithms can run in parallel and each algorithm is initialized with different default option settings. PROC OPTLSO also adds a step to the GA; this step permits generating set search (GSS) algorithms to be used on a selected subset of the current population. GSS algorithms are designed for problems that have continuous variables and have the advantage that, in practice, they often require significantly fewer evaluations than GAs to converge. Furthermore, GSS can provide a measure of local optimality that is very useful in performing multimodal optimization.

The following additional "growth steps" are used whenever continuous variables are present:

local search selection: A small subset of points are selected based on their fitness score and distance to (1) other points and (2) pre-existing locally optimal points.

local search optimization: Local search optimization begins concurrently for each selected point. The only modification made to the original optimization problem is that the variables' lower and upper bounds are modified to temporarily fix integer variables to their current setting.

These additional growth steps are performed for each iteration; they permit selected members of the population (based on diversity and fitness) to benefit from local optimization over the continuous variables. If only integer variables are present, this step is not used.

Multiobjective Optimization

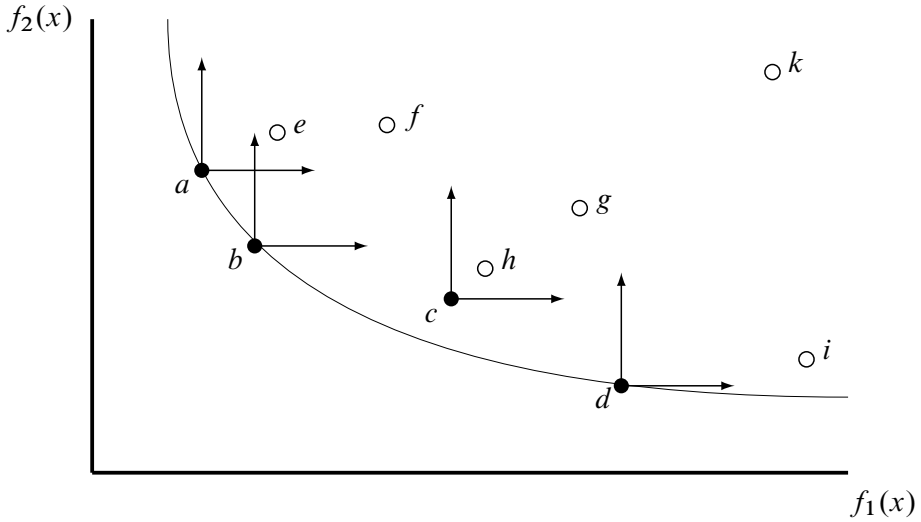
Many practical optimization problems involve more than one objective criterion, so the decision maker needs to examine trade-offs between conflicting objectives. For example, for a particular financial model you might want to maximize profit while minimizing risk, or for a structural model you might want to minimize weight while maximizing strength. The desired result for such problems is usually not a single solution, but rather a range of solutions that can be used to select an acceptable compromise. Ideally each point represents a necessary compromise in the sense that no single objective can be improved without worsening at least one remaining objective. The goal of PROC OPTLSO in the multiobjective case is thus to return to the decision maker a set of points that represent the continuum of best-case scenarios. Multiobjective optimization is performed in PROC OPTLSO whenever more than one objective function of type MIN or MAX exists. For an example, see "Example 3.10: Multiobjective Optimization" on page 70.

Mathematically, multiobjective optimization can be defined in terms of *dominance* and *Pareto optimality*. For a k-objective minimizing optimization problem, a point x is *dominated* by a point y if $f_i(x) \geq f_i(y)$ for all $i = 1,\ldots,k$ and $f_j(x) > f_j(y)$ for some $j = 1,\ldots,k$.

A Pareto-optimal set contains only nondominated solutions. In Figure 3.2, a Pareto-optimal frontier is plotted with respect to minimization objectives $f_1(x)$ and $f_2(x)$ along with a corresponding population of 10 points that are plotted in the objective space. In this example, point a dominates $\{e, f, k\}$, b dominates $\{e, f, g, k\}$, c

dominates $\{g, h, k\}$, and d dominates $\{k, i\}$. Although c is not dominated by any other point in the population, it has not yet converged to the true Pareto-optimal frontier. Thus there exist points in a neighborhood of c that have smaller values of f_1 and f_2.

Figure 3.2 Pareto-Optimal Set



In the constrained case, a point x is dominated by a point y if $\theta(x) > \epsilon$ and $\theta(y) < \theta(x)$, where $\theta(x)$ denotes the maximum constraint violation at point x and FEASTOL=ϵ; thus feasibility takes precedence over objective function values.

Genetic algorithms enable you to attack multiobjective problems directly in order to evolve a set of Pareto-optimal solutions in one run of the optimization process instead of solving multiple separate problems. In addition, local searches in neighborhoods around nondominated points can be conducted to improve objective function values and reduce crowding. Because the number of nondominated points that are encountered might be greater than the total population size, PROC OPTLSO stores nondominated points in an archived set $\mathcal{N}$; you can specify the PARETOMAX= option to control the size of this set.

Although it is difficult to verify directly that a point lies on the true Pareto-optimal frontier without using derivatives, convergence can indirectly be measured by monitoring movement of the population with respect to $\mathcal{N}$, the current set of nondominated points. A number of metrics for measuring convergence in multiobjective evolutionary algorithms have been suggested, such as the generational distance by Van Veldhuizen (1999), the inverted generational distance by Coello Coello and Cruz Cortes (2005), and the averaged Hausdorff distance by Schütze et al. (2012). PROC OPTLSO uses a variation of the averaged Hausdorff distance that is extended for general constraints.

Distance between sets is computed in terms of the distance between a point and a set, which in turn is defined in terms of the distance between two points. The distance measure used by PROC OPTLSO for two points x and y is calculated as

$$d(x, y) = |\theta(x) - \theta(y)| + \sum_{i=1}^{k} |f_i(x) - f_i(y)|$$

where $\theta(x)$ denotes the maximum constraint violation at point x. Then the distance between a point x and a set $\mathcal{A}$ is defined as

$$d(x, \mathcal{A}) = \min_{y \in \mathcal{A}} d(x, y)$$

Let $\mathcal{F}_j$ denote the set of all nondominated points within the current population at the start of generation j. Let $\mathcal{N}_j$ denote the set of all nondominated points at the start of generation j. At the beginning of each generation, $\mathcal{F}_j \subseteq \mathcal{N}_j$ is always satisfied. Then progress made during iteration j+1 is defined as

$$\text{Progress}(j+1) = \frac{1}{|\mathcal{F}_j|} \sum_{x \in \mathcal{F}_j} d(x, \mathcal{N}_{j+1})$$

Because $d(x, \mathcal{N}_{j+1}) = 0$ whenever $x \in \mathcal{F}_j \cap \mathcal{F}_{j+1}$, the preceding sum is over the set of points in the population that move from a status of nondominated to dominated. In this case, the progress made is measured as the distance to the nearest dominating point.

Specifying and Returning Trial Points

You can use the following options to initialize PROC OPTLSO with user-specified points: CACHEIN=, FIRSTGEN=, and PRIMALIN=. You can use the following options to have trial points returned to you: CACHEOUT=, LASTGEN=, and PRIMALOUT=.

Both input and output point data sets have the following columns:

SOL
specifies the point's unique solution tag.

_ID_
specifies the variable and (function) ID name.

VALUE
specifies the variable (function) value.

Input Data Sets

The following DATA step generates 30 random points for the initial population if the variables $x \in \mathbb{R}^5$ have bounds $-5 \leq x_i \leq 10$:

```
data popin;
   low = -5.0;
   upp = 10.0;
   numpoints = 30;
   dim = 5;
   do _sol_=1 to numpoints;
      do i=1 to dim;
         _id_= compress("x" || put(i, 4.0));
         _value_ = low + (upp-low)*ranuni(2);
         output;
      end;
   end;
   keep _sol_ _id_ _value_;
run;
```

You can then use this data set as input for the OPTLSO procedure by using the FIRSTGEN= option.

Output Data Sets

PROC OPTLSO dynamically creates and reports on two metadata functions for you: the true objective (which is a combination of the FCMP objective and the linear and quadratic terms) and the maximum constraint violation. These functions are assigned the following function ID names (therefore, these names should not be used as variable, constraint, or function names):

OBJ
specifies the point's objective. **NOTE:** This value is omitted when solving a multiobjective problem.

INF
specifies the point's maximum constraint violation.

Output data sets have additional rows that correspond to the problem's FCMP functions. These rows must exist for the data set specified in the CACHEIN= option, but they should be omitted for the data sets specified in the FIRSTGEN= and PRIMALIN= options.

When you observe the solution output, it might be easier to compare points if they are listed as rows rather than as columns. SAS provides a variety of ways to transform the results for your purposes. For example, if you prefer that the rows of the data sets correspond to individual points, you can use the following statements to transpose the returned data set, where popout denotes the data set that is returned by PROC OPTLSO:

```
proc transpose data=popout out=poprow (drop=_label_ _name_);
   by _sol_;
   var _value_;
   id _id_;
run;
```

Function Value Caching

To improve performance, the OPTLSO procedure implements a caching system. This caching system helps reduce the overall workload of the solver by eliminating duplicate function evaluations. As the individual optimizers (citizens) submit new trial points for evaluation, the points are saved in the cache along with their function evaluation values. Then, before beginning the potentially expensive function evaluations for a new trial point, PROC OPTLSO determines whether a similar point already exists within the cache. If a similar point is found in the cache, its function values are immediately returned for the newly submitted point. Returning function values from the cache instead of duplicating the effort of computing new function values can lead to significant performance improvements for the overall procedure.

The cache is implemented as a splay tree, similar to that used in Gray and Kolda (2006); Hough, Kolda, and Patrick (2000). The splay tree rebalances itself as new points are added to the cache. This rebalancing ensures that recently added or accessed points are maintained at the root of the tree and are therefore available for quick retrieval. This splay tree data structure lends itself nicely to the needs of a caching system.

When determining whether a newly submitted trial point has a similar point already in the cache, PROC OPTLSO compares the new trial point to all previous trial points by using the value of the CACHETOL= option in the PROC OPTLSO statement. This value specifies a tolerance to use in comparing two points and determining whether the two points are similar enough. In addition to the CACHETOL= value, the cache comparison also takes into account the _SCALE_ values that are specified as part of the variable data set that is specified in the VARIABLES= option.

Two points x and y are considered *cache equivalent* if

$$|x_i - y_i| \leq s_i \epsilon, \text{ for } i = 1, \ldots, n$$

where s denotes the corresponding scaling vector. Thus, if a point x has been evaluated (or is being evaluated) and a point y is submitted for evaluation and satisfies the preceding criteria, y is "cache evaluated" instead of being evaluated directly. In this case, y is associated with $f(x)$ and $c(x)$. Although the splay tree is very efficient, the cache lookup time for quick evaluation might eventually dominate the evaluation process time. You can use the CACHEMAX= option to limit the size of the cache.

Even when the FCMP functions are not defined, the cache system helps to avoid redundant computations when the linear constraints are explicitly handled.

Iteration Log

The iteration log provides detailed information about the progress of the OPTLSO procedure. The log appears by default or if LOGLEVEL=1. The iteration log provides the following information:

Iteration	Displays the number of completed genetic algorithm iterations.
Best Objective	Displays the best objective found at each iteration with respect to the current setting of the FEASTOL= option. **NOTE:** This column is included only for single-objective optimization.
Nondom	Displays the number of nondominated solutions in the current Pareto-optimal set. **NOTE:** This column is included only for multiobjective optimization.
Progress	Displays a numerical indication of the progress being made from one iteration to the next. For a description of how this value is computed, see the section "Multiobjective Optimization" on page 34. **NOTE:** This column is included only for multiobjective optimization.
Infeasibility	Displays the aggregate constraint violation of the current best point.
Evals	Displays the cumulative number of function evaluations.
Time	Displays the time needed to achieve current best solution.

Procedure Termination Messages

After PROC OPTLSO terminates, one of the following messages is displayed:

User interrupted.
The procedure was interrupted by the user.

Function convergence criteria reached.
The best objective value and feasibility have not sufficiently improved for the specified number of iterations. This criterion is a typical exit state if the global optimum is reached early in the algorithm.

Maximum function evaluations reached.
PROC OPTLSO has attempted to exceed the maximum number of function evaluations.

Generations complete.
The maximum number of GA generations (iterations) has been reached. At this point, no new points are generated, and the algorithm exits.

Maximum time reached.
PROC OPTLSO has exceeded the maximum allotted time.

Problem may be unbounded.
The objective value appears to take on arbitrarily large values at points that satisfy the feasibility tolerance.

Infeasible.
The problem is infeasible.

Failed.
The algorithm exited for any reason other than the preceding reasons. For example, this message is displayed if the provided FCMP function does not exist or cannot be evaluated.

ODS Tables

PROC OPTLSO creates three Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input 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 the DETAILS option in the PERFORMANCE statement, then the Timing table is also produced.

Table 3.4 lists all the ODS tables that can be produced by the OPTLSO procedure, along with the statement and option specifications that are required to produce each table.

Table 3.4 ODS Tables Produced by PROC OPTLSO

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input optimization problem	PROC OPTLSO	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTLSO	PRINTLEVEL=1 (default)
PerformanceInfo	List of performance options and their values	PROC OPTLSO	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

Macro Variable _OROPTLSO_

The OPTLSO procedure defines a macro variable named _OROPTLSO_. This variable contains a character string that indicates the status of the procedure upon termination. The contents of the macro variable are interpreted as follows.

STATUS
indicates the procedure's status at termination. It can take one of the following values:

OK	The procedure 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.

SOLUTION_STATUS
indicates the status of the solution at termination. It can take one of the following values:

ABORTED	The user signaled that the procedure should terminate.
ABSFCONV	The procedure terminated on the absolute function convergence criterion.
INFEASIBLE	Problem is globally infeasible. Only returned if all constraints are linear.
MAXFUNC	The procedure terminated on the maximum number of function evaluations.
MAXGEN	The maximum number of genetic algorithm iterations was completed.
MAXTIME	The maximum time limit was reached.
UNBOUNDED	The problem might be unbounded.
FAILED	The status cannot be classified into any of the preceding categories.

OBJECTIVE
indicates the objective value that is obtained by the procedure at termination. **NOTE:** This value is included only for single-objective optimization.

NONDOMINATED
indicates the number of nondominated solutions in the Pareto-optimal set. **NOTE:** This value is included only for multiobjective optimization.

PROGRESS
indicates the progress made during the last iteration. For a description of how this value is computed, see the section "Multiobjective Optimization" on page 34. **NOTE:** This value is included only for multiobjective optimization.

INFEASIBILITY
indicates the level of infeasibility of the constraints at the solution.

EVALUATIONS
indicates the total number of evaluations that were not cached.

ITERATIONS
indicates the number of iterations that were required to solve the problem.

SETUP_TIME
indicates the real time (in seconds) that is taken to set up the optimization problem and distribute data.

SOLUTION_COUNT
indicates number of solutions that are returned in the PRIMALIN= data set if that option was specified.

SOLUTION_TIME
indicates the real time (in seconds) that is taken by the procedure to perform iterations for solving the problem.

Examples: OPTLSO Procedure

Example 3.1: Using Dense Format

This example uses data from Floudas and Pardalos (1992) and illustrates how to use the VARIABLES= and LINCON= options in conjunction with the OBJECTIVE= option. The problem has nine linear constraints and a quadratic objective function. Suppose you want to minimize

$$f(x) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$$

subject to

$$\begin{aligned}
&2x_1 + 2x_2 + x_{10} + x_{11} \le 10\\
&2x_1 + 2x_3 + x_{10} + x_{12} \le 10\\
&2x_1 + 2x_3 + x_{11} + x_{12} \le 10\\
&-8x_1 + x_{10} \le 0\\
&-8x_2 + x_{11} \le 0\\
&-8x_3 + x_{12} \le 0\\
&-2x_4 - x_5 + x_{10} \le 0\\
&-2x_6 - x_7 + x_{11} \le 0\\
&-2x_8 - x_9 + x_{12} \le 0
\end{aligned}$$

and

$$\begin{aligned}
&0 \le x_i \le 1, && i = 1, 2, \ldots, 9\\
&0 \le x_i \le 100, && i = 10, 11, 12\\
&0 \le x_{13} \le 1
\end{aligned}$$

The following statements use the dense format to define the linear constraints:

```
data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
 x1 0   1
 x2 0   1
 x3 0   1
 x4 0   1
 x5 0   1
 x6 0   1
 x7 0   1
 x8 0   1
 x9 0   1
x10 0 100
x11 0 100
x12 0 100
x13 0   1
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function quadobj(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13);
      sum1 = 5*(x1 + x2 + x3 + x4);
      sum2 = 5*(x1**2 + x2**2 + x3**2 + x4**2);
      sum3 = (x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13);
      return (sum1 - sum2 - sum3);
   endsub;
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   quadobj   min
;

data lindata;
   input _id_ $ _lb_ x1-x13 _ub_;
   datalines;
a1 .  2  2  0  0  0  0  0  0  0 1 1 0 0 10
a2 .  2  0  2  0  0  0  0  0  0 1 0 1 0 10
a3 .  2  0  2  0  0  0  0  0  0 0 1 1 0 10
a4 . -8  0  0  0  0  0  0  0  0 1 0 0 0  0
a5 .  0 -8  0  0  0  0  0  0  0 0 1 0 0  0
a6 .  0  0 -8  0  0  0  0  0  0 0 0 1 0  0
a7 .  0  0  0 -2 -1  0  0  0  0 1 0 0 0  0
a8 .  0  0  0  0  0 -2 -1  0  0 0 1 0 0  0
a9 .  0  0  0  0  0  0  0 -2 -1 0 0 1 0  0
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   objective = objdata
   variables = vardata
```

```
   lincon    = lindata;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

The VARIABLES=VARDATA option in the PROC OPTLSO statement specifies the variables and their respective bounds. The objective function is defined by using PROC FCMP and the objective function name QUADOBJ. Other properties are described in the SAS data set objdata. The linear constraints are specified by using the SAS data set lindata, in which each row stores the (zero and nonzero) coefficients of the corresponding linear constraint along with their respective lower and upper bounds. The problem description is then passed to the OPTLSO procedure by using the options VARIABLES=VARDATA, OBJECTIVE=OBJDATA, and LINCON=LINDATA. The PERFORMANCE statement specifies the number of threads that PROC OPTLSO can use.

Output 3.1.1 shows the output from running these statements.

Output 3.1.1 Using Dense Format

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Problem Summary	
Problem Type	NLP
Linear Constraints	LINDATA
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	13
Integer Variables	0
Continuous Variables	13
Number of Constraints	9
Linear Constraints	9
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Output 3.1.1 *continued*

Solution Summary	
Solution Status	Function convergence
Objective	-15.00099016
Infeasibility	0.0009901585
Iterations	33
Evaluations	3956
Cached Evaluations	476
Global Searches	1
Population Size	160
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-15.0010
2	0	_inf_	0.0010
3	0	x1	1.0000
4	0	x2	1.0000
5	0	x3	1.0000
6	0	x4	1.0000
7	0	x5	1.0000
8	0	x6	1.0000
9	0	x7	1.0000
10	0	x8	1.0000
11	0	x9	1.0000
12	0	x10	3.0000
13	0	x11	3.0000
14	0	x12	3.0010
15	0	x13	1.0000
16	0	f	-15.0010

Example 3.2: Using MPS Format

In this example, the linear component of the same problem definition as in Example 3.1 is described by using the OPTMODEL procedure and is saved as an MPS data set. The quadratic objective is then defined by using the FCMP function QUADOBJ.

Because PROC OPTMODEL outputs an MPS data set that uses array notation, you can use the following macro definition to strip brackets from the resulting data set:

```
%macro lsompsmod(setold,setnew);
   data &setnew(drop=i);
      set &setold;
      array FC{*} _CHARACTER_;
      do i=1 to dim(FC);
         FC[i] = compress(FC[i], "[]");
      end;
   run;
%mend;
```

For more complicated array structures, take care to ensure that the resulting transformation is well defined. Next, you run a PROC OPTMODEL step that outputs a MPS data set, followed by the newly defined %LSOMPSMOD macro to strip brackets, followed by a PROC OPTLSO step that takes as input the MPS data set that was output by PROC OPTMODEL and transformed by the %LSOMPSMOD macro.

```
proc optmodel;
   var x{1..13} >= 0 <= 1;
   for {i in 10..12} x[i].ub = 100;
   min z = 0;
   con a1: 2*x[1] + 2*x[2] + x[10] + x[11] <= 10;
   con a2: 2*x[1] + 2*x[3] + x[10] + x[12] <= 10;
   con a3: 2*x[1] + 2*x[3] + x[11] + x[12] <= 10;
   con a4: -8*x[1] + x[10] <= 0;
   con a5: -8*x[2] + x[11] <= 0;
   con a6: -8*x[3] + x[12] <= 0;
   con a7: -2*x[4] - x[5] + x[10] <= 0;
   con a8: -2*x[6] - x[7] + x[11] <= 0;
   con a9: -2*x[8] - x[9] + x[12] <= 0;
   save mps lindataOld;
quit;

%lsompsmod(lindataOld, lindata);

proc optlso
   primalout = solution
   mpsdata   = lindata
   objective = objdata;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Output 3.2.1 shows the output from running these steps.

Output 3.2.1 Using MPS Format

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Output 3.2.1 *continued*

Problem Summary	
Problem Type	NLP
MPS Data Set	LINDATA
Objective Definition Set	OBJDATA
Number of Variables	13
Integer Variables	0
Continuous Variables	13
Number of Constraints	9
Linear Constraints	9
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	-15.00099016
Infeasibility	0.0009901585
Iterations	33
Evaluations	3956
Cached Evaluations	476
Global Searches	1
Population Size	160
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-15.0010
2	0	_inf_	0.0010
3	0	x1	1.0000
4	0	x2	1.0000
5	0	x3	1.0000
6	0	x4	1.0000
7	0	x5	1.0000
8	0	x6	1.0000
9	0	x7	1.0000
10	0	x8	1.0000
11	0	x9	1.0000
12	0	x10	3.0000
13	0	x11	3.0000
14	0	x12	3.0010
15	0	x13	1.0000
16	0	f	-15.0010
17	0	z	0.0000

Example 3.3: Using QPS Format

In this example, the entire problem definition is first described in the following PROC OPTMODEL step and is then saved as a QPS data set in the subsequent PROC OPTLSO step. In this case, no FCMP function needs to be defined.

```
proc optmodel;
   var x{1..13} >= 0 <= 1;
   for {i in 10..12} x[i].ub = 100;
   min z = 5*sum{i in 1..4} x[i]
      - 5*sum{i in 1..4} x[i]**2 - sum{i in 5..13} x[i];
   con a1: 2*x[1] + 2*x[2] + x[10] + x[11] <= 10;
   con a2: 2*x[1] + 2*x[3] + x[10] + x[12] <= 10;
   con a3: 2*x[1] + 2*x[3] + x[11] + x[12] <= 10;
   con a4: -8*x[1] + x[10] <= 0;
   con a5: -8*x[2] + x[11] <= 0;
   con a6: -8*x[3] + x[12] <= 0;
   con a7: -2*x[4] - x[5] + x[10] <= 0;
   con a8: -2*x[6] - x[7] + x[11] <= 0;
   con a9: -2*x[8] - x[9] + x[12] <= 0;
   save qps qpdata;
quit;

proc optlso
   primalout = solution
   qpsdata   = qpdata;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Note that in this case the objective definition is taken directly from the QPS data set qpdata. Output 3.3.1 shows the output from running these steps.

Output 3.3.1 Using QPS Format

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Problem Summary	
Problem Type	QP
QPS Data Set	QPDATA
Number of Variables	13
Integer Variables	0
Continuous Variables	13
Number of Constraints	9
Linear Constraints	9
Nonlinear Constraints	0
Objective Definition Source	QPDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	-15.00099016
Infeasibility	0.0009901585
Iterations	33
Evaluations	3956
Cached Evaluations	476
Global Searches	1
Population Size	160
Seed	1

Output 3.3.1 *continued*

Obs	_sol_	_id_	_value_
1	0	_obj_	-15.0010
2	0	_inf_	0.0010
3	0	x[1]	1.0000
4	0	x[2]	1.0000
5	0	x[3]	1.0000
6	0	x[4]	1.0000
7	0	x[5]	1.0000
8	0	x[6]	1.0000
9	0	x[7]	1.0000
10	0	x[8]	1.0000
11	0	x[9]	1.0000
12	0	x[10]	3.0000
13	0	x[11]	3.0000
14	0	x[12]	3.0010
15	0	x[13]	1.0000
16	0	z	-15.0010

Example 3.4: Combining MPS and FCMP Function Definitions

In this example, the linear component of the same problem definition as in Example 3.1 is described by using the OPTMODEL procedure and is saved as an MPS data set. The quadratic component of the objective is then defined by using the FCMP function QUADOBJ.

As in "Example 3.2: Using MPS Format" on page 44, you can use the macro definition lsompsmod to strip brackets from the resulting data set:

```
%macro lsompsmod(setold,setnew);
   data &setnew(drop=i);
      set &setold;
      array FC{*} _CHARACTER_;
      do i=1 to dim(FC);
         FC[i] = compress(FC[i], "[]");
      end;
   run;
%mend;
```

For more complicated array structures, take care to ensure that the resulting transformation is well defined. Next you run a PROC OPTMODEL step that outputs a MPS data set, followed by the %LSOMPSMOD macro, followed by the PROC FCMP step that defines the FCMP function QUADOBJ, followed by a PROC OPTLSO step that takes as input both the MPS data set that was output by PROC OPTMODEL and transformed by the %LSOMPSMOD macro and the quadratic component of the objective that was defined by PROC FCMP.

```
proc optmodel;
   var x{1..13} >= 0 <= 1;
   for {i in 10..12} x[i].ub = 100;
   min linobj = 5*sum{i in 1..4} x[i] - sum{i in 5..13} x[i];
   con a1: 2*x[1] + 2*x[2] + x[10] + x[11] <= 10;
   con a2: 2*x[1] + 2*x[3] + x[10] + x[12] <= 10;
   con a3: 2*x[1] + 2*x[3] + x[11] + x[12] <= 10;
   con a4: -8*x[1] + x[10] <= 0;
   con a5: -8*x[2] + x[11] <= 0;
   con a6: -8*x[3] + x[12] <= 0;
   con a7: -2*x[4] - x[5] + x[10] <= 0;
   con a8: -2*x[6] - x[7] + x[11] <= 0;
   con a9: -2*x[8] - x[9] + x[12] <= 0;
   save mps lindataOld;
quit;
%lsompsmod(lindataOld, lindata);

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function quadobj(x1,x2,x3,x4,f1);
      return (f1 - 5*(x1**2 + x2**2 + x3**2 + x4**2));
   endsub;
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f1  linobj    .
f   quadobj   min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   mpsdata   = lindata
   objective = objdata;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Output 3.4.1 shows the output from running these steps.

Output 3.4.1 Using MPS Format

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Output 3.4.1 *continued*

Problem Summary	
Problem Type	NLP
MPS Data Set	LINDATA
Objective Definition Set	OBJDATA
Number of Variables	13
Integer Variables	0
Continuous Variables	13
Number of Constraints	9
Linear Constraints	9
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize
Objective Intermediate Functions	1

Solution Summary	
Solution Status	Function convergence
Objective	-15.00099016
Infeasibility	0.0009901585
Iterations	33
Evaluations	3956
Cached Evaluations	476
Global Searches	1
Population Size	160
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-15.0010
2	0	_inf_	0.0010
3	0	x1	1.0000
4	0	x2	1.0000
5	0	x3	1.0000
6	0	x4	1.0000
7	0	x5	1.0000
8	0	x6	1.0000
9	0	x7	1.0000
10	0	x8	1.0000
11	0	x9	1.0000
12	0	x10	3.0000
13	0	x11	3.0000
14	0	x12	3.0010
15	0	x13	1.0000
16	0	f	-15.0010
17	0	f1	4.9990
18	0	linobj	4.9990

Example 3.5: Linear Constraints and a Nonlinear Objective

The problem in this example is to minimize the six-hump camel-back function (Michalewicz 1996, Appendix B). Minimize

$$f(x) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1 x_2 + \left(-4 + 4x_2^2\right) x_2^2$$

subject to

$$\begin{aligned} 2x_1 + x_2 &\leq 2 \\ x_1 - x_2 &\geq -2 \\ x_1 + 2x_2 &\geq -2 \end{aligned}$$

Providing derivative-free algorithms with good estimates for lower and upper bounds often greatly improves performance because it prevents the algorithm from unnecessarily sampling in regions that you do not want to explore. For this problem, the following statements add the explicit variable bounds $-2 \leq x_1 \leq 2$ and $-2 \leq x_2 \leq 2$:

```
data xbnds;
   input _id_ $ _lb_ _ub_;
   datalines;
x1 -2 2
x2 -2 2
;

data lindata;
   input _id_ $ _lb_ x1 x2 _ub_;
   datalines;
a1   . 2  1 2
a2  -2 1 -1 .
a3  -2 1  2 .
;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   sixhump     min
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function sixhump(x1,x2);
      return ((4 - 2.1*x1**2 + x1**4/3)*x1**2 + x1*x2 + (-4 + 4*x2**2)*x2**2);
   endsub;
run;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = xbnds
   objective = objdata
   lincon    = lindata;
```

```
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Output 3.5.1 shows the output from running these steps.

Output 3.5.1 Linear Constraints and a Nonlinear Objective

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Problem Summary	
Problem Type	NLP
Linear Constraints	LINDATA
Objective Definition Set	OBJDATA
Variables	XBNDS
Number of Variables	2
Integer Variables	0
Continuous Variables	2
Number of Constraints	3
Linear Constraints	3
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	-1.031628453
Infeasibility	0
Iterations	35
Evaluations	2002
Cached Evaluations	24
Global Searches	1
Population Size	80
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-1.03163
2	0	_inf_	0.00000
3	0	x1	-0.08984
4	0	x2	0.71266
5	0	f	-1.03163

Example 3.6: Using Nonlinear Constraints

The following optimization problem is discussed in Haverly (1978) and Liebman et al. (1986). This example illustrates how to use PROC FCMP to define nonlinear constraints and use an MPS data set to define linear constraints. Maximize

$$f(x) = 9x_1 + 15x_2 - 6x_3 - 16x_4 - 10x_5$$

subject to

$$\begin{aligned}
x_3 + x_4 &= x_6 + x_7 \\
x_6 + x_8 &= x_1 \\
x_7 + x_9 &= x_2 \\
x_8 + x_9 &= x_5 \\
2.5x_1 - x_{10}x_6 - 2x_8 &\geq 0 \\
1.5x_2 - x_{10}x_7 - 2x_9 &\geq 0 \\
3x_3 + x_4 - x_{10}(x_3 + x_4) &= 0
\end{aligned}$$

and

$$\begin{aligned}
0 &\leq x_1 \leq 100 \\
0 &\leq x_2 \leq 200 \\
1 &\leq x_{10} \leq 3 \\
0 &\leq x_i, \text{ for } i = 3, \ldots, 9
\end{aligned}$$

In the following steps, the linear component of the problem definition is first described in PROC OPTMODEL and then saved as an MPS data set. Because the objective is linear, no FCMP objective function needs to be used. In the second section of steps, the nonlinear constraints are defined by using FCMP functions, and their corresponding names and lower and upper bounds are stored in the data set condata. The OPTLSO procedure is then called with the options NLINCON=CONDATA and MPSDATA=NLCEX.

As in "Example 3.2: Using MPS Format" on page 44, you can use the macro definition lsompsmod to strip brackets from the resulting data set:

```
%macro lsompsmod(setold,setnew);
   data &setnew(drop=i);
      set &setold;
      array FC{*} _CHARACTER_;
      do i=1 to dim(FC);
         FC[i] = compress(FC[i], "[]");
      end;
   run;
%mend;

proc optmodel;
   var x{1..10} >= 0;
   x[10].lb = 1;
   x[10].ub = 3;
   x[1].ub = 100;
   x[2].ub = 200;
   con x[3] + x[4] = x[6] + x[7],
   x[6] + x[8] = x[1],
```

```
   x[7] + x[9] = x[2],
   x[8] + x[9] = x[5];
   max f = 9*x[1] + 15*x[2] - 6*x[3] - 16*x[4] - 10*x[5];
   save mps nlcexOld;
quit;

%lsompsmod(nlcexOld, nlcex);

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function nlc1(x1,x6,x8,x10);
      return (2.5*x1 - x10*x6 - 2*x8);
   endsub;
   function nlc2(x2,x7,x9,x10);
      return (1.5*x2 - x10*x7 - 2*x9);
   endsub;
   function nlc3(x3,x4,x10);
      return (3*x3 + x4 - x10*(x3 + x4));
   endsub;
run;

data condata;
   input _id_ $ _lb_ _ub_;
   datalines;
nlc1 0 .
nlc2 0 .
nlc3 0 0
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   mpsdata   = nlcex
   nlincon   = condata
   logfreq   = 10;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Output 3.6.1 shows the ODS tables that are produced from running these steps.

Output 3.6.1 Using Nonlinear Constraints: ODS Tables

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Output 3.6.1 *continued*

Problem Summary	
Problem Type	NLP
MPS Data Set	NLCEX
Nonlinear Constraints	CONDATA
Number of Variables	10
Integer Variables	0
Continuous Variables	10
Number of Constraints	7
Linear Constraints	4
Nonlinear Constraints	3
Objective Definition Source	NLCEX
Objective Sense	Maximize

Solution Summary	
Solution Status	Function convergence
Objective	400.01457808
Infeasibility	0.000935117
Iterations	41
Evaluations	4828
Cached Evaluations	530
Global Searches	1
Population Size	160
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	400.015
2	0	_inf_	0.001
3	0	x1	0.000
4	0	x2	200.000
5	0	x3	0.000
6	0	x4	99.999
7	0	x5	100.000
8	0	x6	0.000
9	0	x7	99.999
10	0	x8	0.000
11	0	x9	100.001
12	0	x10	1.000
13	0	f	400.015
14	0	nlc1	-0.000
15	0	nlc2	-0.001
16	0	nlc3	0.000

Output 3.6.2 shows the iteration log from running these steps.

Output 3.6.2 Using Nonlinear Constraints: Log

```
NOTE: The OPTLSO procedure is executing in single-machine mode.
NOTE: The OPTLSO algorithm is using up to 2 threads.
NOTE: The problem has 10 variables (0 integer, 10 continuous).
NOTE: The problem has 7 constraints (4 linear, 3 nonlinear).
NOTE: The problem has 3 FCMP function definitions.
NOTE: The deterministic parallel mode is enabled.
                        Best
        Iteration       Objective    Infeasibility    Evals     Time
                1               0                0      170        0
               11 399.967674198371               0     1274        0
               21 400.003665875725  0.0006109165499     2468        0
               31 400.014578082483 0.00093511701363     3663        1
               41 400.014578082483 0.00093511701363     4828        1
NOTE: Function convergence criteria reached.
NOTE: There were 3 observations read from the data set WORK.CONDATA.
NOTE: The data set WORK.SOLUTION has 16 observations and 3 variables.
```

Example 3.7: Using External Data Sets

This example illustrates the use of external data sets that are specified in the OBJECTIVE= option. The Bard function (Moré, Garbow, and Hillstrom 1981) is a least squares problem that has $n = 3$ parameters and $m = 15$ functions f_k,

$$f(x) = \frac{1}{2}\sum_{k=1}^{15} f_k^2(x), \quad x = (x_1, x_2, x_3)$$

where

$$f_k(x) = y_k - \left(x_1 + \frac{k}{v_k x_2 + w_k x_3}\right)$$

with $v_k = 16 - k$, $w_k = \min(u_k, v_k)$, 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 $f(x^*) = 4.107\text{E}{-3}$ occurs at the point $(0.08, 1.13, 2.34)$. In this example, the additional variable bounds $-1000 \le x_i \le 1000$ for $i = 1, 2, 3$ are added.

There are three approaches to specifying the objective function. The first approach assumes that the necessary data are stored within the FCMP function. In this case, you can specify the objective function without using an external data set, as follows:

```
data vardata;
   input _id_ $ _lb_ _ub_ ;
   datalines;
x1  -1000 1000
x2  -1000 1000
x3  -1000 1000
;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   bard     min
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3);
      array y[15] /nosym (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);
      fx = 0;
      do k=1 to 15;
         vk = 16 - k;
         wk = min(k,vk);
         fxk = y[k] - (x1 + k/(vk*x2 + wk*x3));
         fx = fx + fxk**2;
      end;
      return (0.5*fx);
   endsub;
run;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
   performance nthreads=2;
run;

proc print data=solution;
run;
```

Output 3.7.1 shows the output from running these steps.

Output 3.7.1 Using External Data Sets

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	2

Output 3.7.1 *continued*

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	3
Integer Variables	0
Continuous Variables	3
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	0.0041074417
Infeasibility	0
Iterations	73
Evaluations	6261
Cached Evaluations	91
Global Searches	1
Population Size	120
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	0.00411
2	0	_inf_	0.00000
3	0	x1	0.08239
4	0	x2	1.13238
5	0	x3	2.34437
6	0	f	0.00411

This approach is cumbersome if the size of the required data increases. A second approach for medium-sized data sets is to use the READ_ARRAY statement within the FCMP function definition. Because the environment might be distributed, PROC OPTLSO requires a list of all data sets that are used in FCMP function definitions to ensure that the corresponding data sets are available. This list should be specified by using the READARRAY statement.

```
data barddata;
   input y @@;
   datalines;
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
;

data vardata;
   input _id_ $ _lb_ _ub_ ;
   datalines;
x1  -1000 1000
x2  -1000 1000
x3  -1000 1000
;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f   bard     min
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3);
      array y[15] /nosym;
      rc = read_array('barddata', y);
      fx = 0;
      do k=1 to 15;
         dk  = (16-k)*x2 + min(k,16-k)*x3;
         fxk = y[k] - (x1 + k/dk);
         fx = fx + fxk**2;
      end;
      return (0.5*fx);
   endsub;
run;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
   readarray barddata;
run;

proc print data=solution;
run;
```

Output 3.7.2 shows the output from running these statements.

Output 3.7.2 Using External Data Sets (II)

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Data Access Information			
Data	**Engine**	**Role**	**Path**
WORK.BARDDATA	V9	Input	On Client

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	3
Integer Variables	0
Continuous Variables	3
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	0.0041074417
Infeasibility	0
Iterations	73
Evaluations	6261
Cached Evaluations	91
Global Searches	1
Population Size	120
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	0.00411
2	0	_inf_	0.00000
3	0	x1	0.08239
4	0	x2	1.13238
5	0	x3	2.34437
6	0	f	0.00411

The preceding approach can be prohibitive if the size of the data set is large. As a third approach to specifying the objective function, PROC OPTLSO provides an alternate data input gateway that is described in the OBJECTIVE= data set, as shown in the following statements:

```
data vardata;
   input _id_ $ _lb_ _ub_ ;
   datalines;
x1  -1000 1000
x2  -1000 1000
x3  -1000 1000
;

data barddata;
   k = _n_;
   input y @@;
   datalines;
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
;

data objdata;
   input _id_ $ _function_ $ _sense_ $ _dataset_ $;
   datalines;
fx   bard     min  barddata
;

proc fcmp outlib=sasuser.myfuncs.mypkg;
   function bard(x1, x2, x3, k, y);
      vk = 16 - k;
      wk = min(k,vk);
      fxk = y - (x1 + k/(vk*x2 + wk*x3));
      return (0.5*fxk**2);
   endsub;
run;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata;
   performance nodes=2 nthreads=8;
run;


proc print data=solution;
run;
```

Output 3.7.3 shows the output from running these statements.

Output 3.7.3 Using External Data Sets (III)

The OPTLSO Procedure

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	3
Integer Variables	0
Continuous Variables	3
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize
Objective Data Set	barddata

Solution Summary	
Solution Status	Function convergence
Objective	0.0041074417
Infeasibility	0
Iterations	73
Evaluations	6261
Cached Evaluations	91
Global Searches	1
Population Size	120
Seed	1

Performance Information	
Host Node	<< your grid host >>
Execution Mode	Distributed
Number of Compute Nodes	2
Number of Threads per Node	8

Example 3.8: Johnson's Systems of Distributions

This example further illustrates the use of external data sets that are specified in the OBJECTIVE= option. For this example, a data set that contains $n = 20,000$ randomly generated observations is used to estimate the parameters for the Johnson S_U family of distributions (Bowman and Shenton 1983). The objective is the log likelihood for the family, which involves four variables, $x = (x_1, x_2, x_3, x_4)$:

$$f(x) = n\log(x_4) - n\log(x_2) - \frac{1}{2}\sum_{k=1}^{n}\left((x_3 + x_4\log(z_k))^2 + \log(1 + y_k^2)\right)$$

where

$$z_k = y_k + \sqrt{1 + y_k^2} \text{ with } y_k = \frac{d_k - x_1}{x_2}$$

Here, d_k denotes the value of d in the kth observation of the data set that is generated by the following DATA step.

```
data sudata;
   n=20000;
   theta=-1;
   sigma=1;
   delta=3;
   gamma=5;
   rngSeed=123;
   do i = 1 to n;
      z = rannor(rngSeed);
      a = exp( (z - gamma)/delta );
      d = sigma * ( (a**2 - 1)/(2*a) ) + theta;
      output;
   end;
   keep d;
run;
```

This generates a data set called sudata that contains $n = 20,000$ observations. You can modify n to increase or decrease the computational work per function evaluation. The following call to PROC FCMP defines the corresponding FCMP function definition:

```
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function jsu(x4,x2,f1);
      return (20000*(log(x4) - log(x2)) + f1);
   endsub;

   function jsu1(x1,x2,x3,x4,d);
      yk = (d - x1)/x2;
      zk = yk + sqrt(1 + yk**2);
      return (-0.5*(x3 + x4*log(zk))**2 -0.5*log(1 + yk**2));
   endsub;
run;
options cmplib = sasuser.myfuncs;
```

In the following steps, the assumption for the definition of jsu and jsu1 is that jsu1 is called once for each line of data (in this case 20,000 times) and cumulatively summed. The resulting value is then provided to the function jsu for a final calculation, which is called only once per evaluation of $f(x)$.

```
data objdata;
   input _id_ $ _function_ $ _sense_ $ _dataset_ $;
   datalines;
f1 jsu1 .   sudata
f  jsu  max .
;

data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
```

```
x1  .     .
x2  1e-12 .
x3  .     .
x4  1e-12 .
;

proc optlso
   primalout = solution
   objective = objdata
   variables = vardata
   logfreq   = 100
   maxgen    = 1000;
   performance nodes=4 nthreads=8;
run;
```

Output 3.8.1 shows the output from running these steps.

Output 3.8.1 Estimation for Johnson S_U Family of Distributions

The OPTLSO Procedure

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	4
Integer Variables	0
Continuous Variables	4
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Maximize
Objective Intermediate Functions	1
Objective Data Set	sudata

Solution Summary	
Solution Status	Function convergence
Objective	-8414.726059
Infeasibility	0
Iterations	494
Evaluations	43171
Cached Evaluations	147
Global Searches	1
Population Size	120
Seed	1

Output 3.8.1 *continued*

Performance Information	
Host Node	<< your grid host >>
Execution Mode	Distributed
Number of Compute Nodes	4
Number of Threads per Node	8

Example 3.9: Discontinuous Function with a Lookup Table

This example illustrates the ability of PROC OPTLSO to optimize a discontinuous function. The example generates a data set of discrete values that approximate a given smooth nonlinear function. The function being optimized is simply using that data set as a lookup table to find the appropriate discretized value.

```
%let N = 100;
%let L = 0;
%let U = 10;

options cmplib = sasuser.myfuncs;
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function SmoothFunc(x);
      y = x*sin(x) + x*x*cos(x);
      return (y);
   endsub;

   function DiscretizedFunc(x);
      array lookup[&N, 2] / nosymbols;
      rc = read_array('f_discrete', lookup);
      do i = 1 to %eval(&N-1);
         if x >= lookup[i,1] and x < lookup[i+1,1] then
         do;
            /* lookup value at nearest smaller discretized point */
            y = lookup[i,2];
            i = %eval(&N-1);
         end;
      end;
      return (y);
   endsub;
run;
```

The previous statements define PROC FCMP functions for both the smooth and discretized versions of the objective function. The smooth version is used as follows to generate the discrete points in the lookup table data set f_discrete for x values at 100 (N) points between 0 (L) and 10 (U). The values in the following data set created are used in the discretized version of the function that will be used in PROC OPTLSO for optimization. That discretized version of the function performs a simple lookup of the point data that are contained in the f_discrete data set. For a specified x value, the function finds the two discrete values in the data set that are closest to x. Then the smaller of the two nearest points is returned as the function value.

```
data f_discrete;
   a=&L; b=&U; n=&N;
   drop i a b n;
   do i = 1 to n;
      x = a + (i/n)*(b-a);
      y = SmoothFunc(x);
      output;
   end;
run;

data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
x   0   10
;

/* Use the discretized function as the objective */
data objdata;
   length _function_ $16;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f    DiscretizedFunc    min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primaluut = finalsol
   variables = vardata
   objective = objdata;
   readarray f_discrete;
run;

proc print data=finalsol;
run;
```

Output 3.9.1 shows the ODS tables that are produced.

Output 3.9.1 Discontinuous Function: ODS Tables

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Data Access Information			
Data	**Engine**	**Role**	**Path**
WORK.F_DISCRETE	V9	Input	On Client

Output 3.9.1 *continued*

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	1
Integer Variables	0
Continuous Variables	1
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Solution Summary	
Solution Status	Function convergence
Objective	-93.18498962
Infeasibility	0
Iterations	12
Evaluations	306
Cached Evaluations	57
Global Searches	1
Population Size	40
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-93.1850
2	0	_inf_	0.0000
3	0	x	9.7087
4	0	f	-93.1850

The following code optimizes the smooth version of the same function to demonstrate that virtually the same result is achieved in both cases:

```
%let N = 100;
%let L = 0;
%let U = 10;

options cmplib = sasuser.myfuncs;
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function SmoothFunc(x);
      y = x*sin(x) + x*x*cos(x);
      return (y);
   endsub;
run;
```

```
data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
x   0   10
;

/* Use the smooth function as the objective */
data objdata;
   length _function_ $16;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f    SmoothFunc    min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primaluut = finalsol
   variables = vardata
   objective = objdata;
run;

proc print data=finalsol;
run;
```

Output 3.9.2 shows the ODS tables that are produced.

Output 3.9.2 Smooth Function: ODS Tables

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	1
Integer Variables	0
Continuous Variables	1
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Output 3.9.2 *continued*

Solution Summary	
Solution Status	Function convergence
Objective	-93.22152602
Infeasibility	0
Iterations	19
Evaluations	475
Cached Evaluations	80
Global Searches	1
Population Size	40
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	-93.2215
2	0	_inf_	0.0000
3	0	x	9.7269
4	0	f	-93.2215

Example 3.10: Multiobjective Optimization

The following optimization problem is discussed in Huband et al. (2006); Custódio et al. (2011). This example illustrates how to use PROC FCMP to define multiple nonlinear objectives. This problem minimizes

$$f_1(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 \text{ and } f_2(x) = (x_1 - x_2)^2 + (x_2 - 3)^2$$

subject to $0 \leq x_1, x_2 \leq 5$. The VARIABLES=VARDATA option in the PROC OPTLSO statement specifies the variables and their respective bounds. The objective functions are defined by using PROC FCMP, and the objective function names and other properties are described in the SAS data set objdata. The problem description is then passed to the OPTLSO procedure by using the options VARIABLES=VARDATA and OBJECTIVE=OBJDATA.

```
data vardata;
   input _id_ $ _lb_ _ub_;
   datalines;
x1 0 5
x2 0 5
;
proc fcmp outlib=sasuser.myfuncs.mypkg;
   function fdef1(x1, x2);
      return ((x1-1)**2 + (x1-x2)**2);
   endsub;

   function fdef2(x1, x2);
      return ((x1-x2)**2 + (x2-3)**2);
   endsub;
run;

data objdata;
   input _id_ $ _function_ $ _sense_ $;
   datalines;
f1 fdef1 min
```

```
f2 fdef2 min
;

options cmplib = sasuser.myfuncs;
proc optlso
   primalout = solution
   variables = vardata
   objective = objdata
   logfreq = 50
;
run;

proc transpose data=solution out=pareto label=_sol_ name=_sol_;
   by _sol_;
   var _value_;
   id _id_;
run;

proc gplot data=pareto;
   plot f2*f1;
run;

quit;
```

Output 3.10.1 shows the ODS tables that are produced.

Output 3.10.1 Multiobjective: ODS Tables

The OPTLSO Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Problem Summary	
Problem Type	NLP
Objective Definition Set	OBJDATA
Variables	VARDATA
Number of Variables	2
Integer Variables	0
Continuous Variables	2
Number of Constraints	0
Linear Constraints	0
Nonlinear Constraints	0
Objective Definition Source	OBJDATA
Objective Sense	Minimize

Output 3.10.1 *continued*

Solution Summary	
Solution Status	Function convergence
Nondominated	5000
Progress	6.9766249E-7
Infeasibility	0
Iterations	444
Evaluations	23647
Cached Evaluations	103
Global Searches	1
Population Size	80
Seed	1

Output 3.10.2 shows the iteration log.

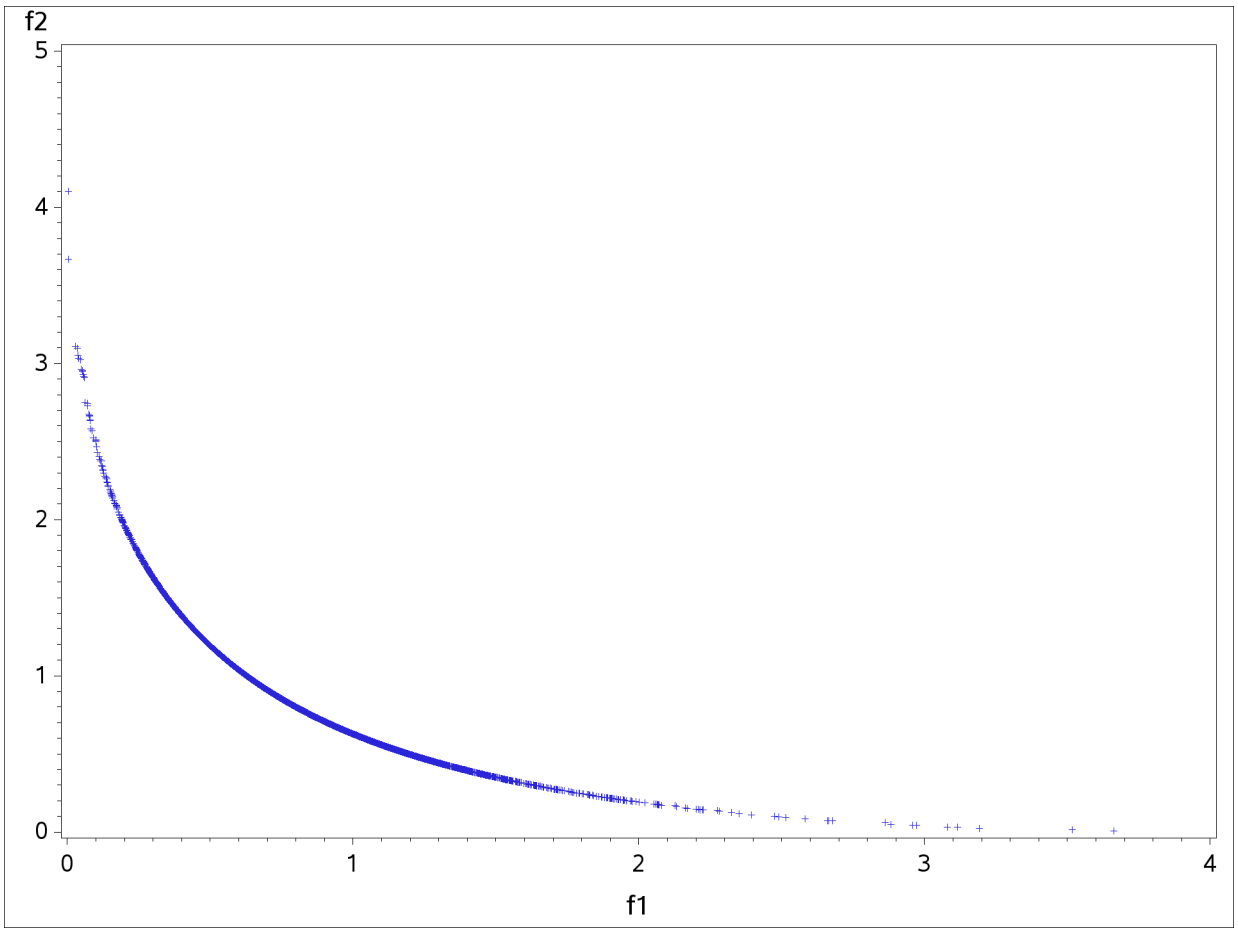
Output 3.10.2 Multiobjective: Log

```
NOTE: The OPTLSO procedure is executing in single-machine mode.
NOTE: The OPTLSO algorithm is using up to 4 threads.
NOTE: The problem has 2 variables (0 integer, 2 continuous).
NOTE: The problem has 0 constraints (0 linear, 0 nonlinear).
NOTE: The problem has 2 FCMP function definitions.
NOTE: The deterministic parallel mode is enabled.
        Iteration    Nondom        Progress    Infeasibility    Evals     Time
                1         7               .                0       84        0
               51       962     0.000070835                0     2885        0
              101      1689     0.000017074                0     5613        1
              151      2289     0.000009575                0     8249        2
              201      2933     0.000004561                0    10906        2
              251      3429     0.000051118                0    13573        4
              301      3876     0.000001452                0    16163        5
              351      4386     0.000000716                0    18757        6
              401      4796     0.000002728                0    21375        7
              444      5000     0.000000698                0    23647        9
NOTE: Function convergence criteria reached.
NOTE: There were 2 observations read from the data set WORK.VARDATA.
NOTE: There were 2 observations read from the data set WORK.OBJDATA.
NOTE: The data set WORK.SOLUTION has 25000 observations and 3 variables.
```

When solving a multiobjective problem with 2 objectives, it can be useful to create a plot with PROC GPLOT of the Pareto-optimal set returned by PROC OPTLSO.

Output 3.10.3 shows a plot of the Pareto-optimal set found by PROC OPTLSO.

Output 3.10.3 Multiobjective: Plot of Pareto-optimal Set



References

Bowman, K. O., and Shenton, L. R. (1983). "Johnson's System of Distributions." In *Encyclopedia of Statistical Sciences*, vol. 4, edited by S. Kotz, N. L. Johnson, and C. B. Read. New York: John Wiley & Sons.

Coello Coello, C. A., and Cruz Cortes, N. (2005). "Solving Multiobjective Optimization Problems Using an Artificial Immune System." *Genetic Programming and Evolvable Machines* 6:163–190.

Custódio, A. L., Madeira, J. F. A., Vaz, A. I. F., and Vicente, L. N. (2011). "Direct Multisearch for Multiobjective Optimization." *SIAM Journal on Optimization* 21:1109–1140.

Floudas, C. A., and Pardalos, P. M. (1992). *Recent Advances in Global Optimization*. Princeton, NJ: Princeton University Press.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Gray, G. A., and Fowler, K. R. (2011). "The Effectiveness of Derivative-Free Hybrid Methods for Black-Box Optimization." *International Journal of Mathematical Modeling and Numerical Optimization* 2:112–133.

Gray, G. A., Fowler, K. R., and Griffin, J. D. (2010). "Hybrid Optimization Schemes for Simulation-Based Problems." *Procedia Computer Science* 1:1349–1357.

Gray, G. A., and Kolda, T. G. (2006). "Algorithm 856: APPSPACK 4.0—Asynchronous Parallel Pattern Search for Derivative-Free Optimization." *ACM Transactions on Mathematical Software* 32:485–507.

Griffin, J. D., Fowler, K. R., Gray, G. A., and Hemker, T. (2011). "Derivative-Free Optimization via Evolutionary Algorithms Guiding Local Search (EAGLS) for MINLP." *Pacific Journal of Optimization* 7:425–443.

Griffin, J. D., and Kolda, T. G. (2010a). "Asynchronous Parallel Hybrid Optimization Combining DIRECT and GSS." *Optimization Methods and Software* 25:797–817.

Griffin, J. D., and Kolda, T. G. (2010b). "Nonlinearly Constrained Optimization Using Heuristic Penalty Methods and Asynchronous Parallel Generating Set Search." *Applied Mathematics Research Express* 2010:36–62.

Griffin, J. D., Kolda, T. G., and Lewis, R. M. (2008). "Asynchronous Parallel Generating Set Search for Linearly Constrained Optimization." *SIAM Journal on Scientific Computing* 30:1892–1924.

Haverly, C. A. (1978). "Studies of the Behavior of Recursion for the Pooling Problem." *SIGMAP Bulletin, Association for Computing Machinery* 25:19–28.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor: University of Michigan Press.

Hough, P. D., Kolda, T. G., and Patrick, H. A. (2000). *Usage Manual for APPSPACK 2.0*. Technical Report SAND2000-8843, Sandia National Laboratories, Albuquerque, NM and Livermore, CA.

Huband, S., Hingston, P., Barone, L., and While, L. (2006). "A Review of Multiobjective Test Problems and a Scalable Test Problem Toolkit." *IEEE Transactions on Evolutionary Computation* 10:477–506.

Jones, D. R., Perttunen, C. D., and Stuckman, B. E. (1993). "Lipschitzian Optimization without the Lipschitz Constant." *Journal of Optimization Theory and Applications* 79:157–181.

Kolda, T. G., Lewis, R. M., and Torczon, V. (2003). "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods." *SIAM Review* 45:385–482.

Liebman, J., Lasdon, L., Schrage, L., and Waren, A. (1986). *Modeling and Optimization with GINO*. Redwood City, CA: Scientific Press.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. New York: Springer-Verlag.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1981). "Testing Unconstrained Optimization Software." *ACM Transactions on Mathematical Software* 7:17–41.

Plantenga, T. (2009). *HOPSPACK 2.0 User Manual (v 2.0.2)*. Technical report, Sandia National Laboratories.

Schütze, O., Esquivel, X., Lara, A., and Coello Coello, C. A. (2012). "Using the Averaged Hausdorff Distance as a Performance Measure in Evolutionary Multiobjective Optimization." *IEEE Transactions on Evolutionary Computation* 16:504–522. doi:10.1109/TEVC.2011.2161872.

Taddy, M. A., Lee, H. K. H., Gray, G. A., and Griffin, J. D. (2009). "Bayesian Guided Pattern Search for Robust Local Optimization." *Technometrics* 51:389–401.

Van Veldhuizen, D. A. (1999). *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. Ph.D. diss., Air Force Institute of Technology.

Chapter 4
The GA Procedure

Contents

Overview: GA Procedure

Genetic algorithms are a family of local search algorithms that seek optimal solutions to problems by applying the principles of natural selection and evolution. Genetic algorithms can be applied to almost any optimization problem and are especially useful for problems where other calculus-based techniques do not work, such as when the objective function has many local optima, when it is not differentiable or continuous, or when solution elements are constrained to be integers or sequences. In most cases genetic algorithms require more

computation than specialized techniques that take advantage of specific problem structures or characteristics. However, for optimization problems with no such techniques available, genetic algorithms provide a robust general method of solution.

In general, genetic algorithms use some variation of the following procedure to search for an optimal solution:

initialization: An initial population of solutions is randomly generated, and the objective function is evaluated for each member of this initial generation.

selection: Individual members of the current generation are chosen stochastically either to parent the next generation or to be passed on to it, such that those members who are the fittest are more likely to be selected. A solution's fitness is based on its objective value, with better objective values reflecting higher fitness.

crossover: Some of the selected solutions are passed to a crossover operator. The crossover operator combines two or more parents to produce new offspring solutions for the next generation. The crossover operator tends to produce new offspring that retain the common characteristics of the parent solutions, while combining the other traits in new ways. In this way new areas of the search space are explored, hopefully while retaining optimal solution characteristics.

mutation: Some of the next-generation solutions are passed to a mutation operator, which introduces random variations in the solutions. The purpose of the mutation operator is to ensure that the solution space is adequately searched to prevent premature convergence to a local optimum.

repeat: The current generation of solutions is replaced by the new generation. If the stopping criterion is not satisfied, the process returns to the *selection* phase.

The crossover and mutation operators are commonly called *genetic operators*. Selection and crossover distinguish genetic algorithms from a purely random search and direct the algorithm toward finding an optimum. Mutation is designed to ensure diversity in the search to prevent premature convergence to a local optimum.

There are many ways to implement the general strategy just outlined, and it is also possible to combine the genetic algorithm approach with other heuristic solution improvement techniques. In the traditional genetic algorithm, the solutions space is composed of bit strings, mapped to an objective function, and the genetic operators are modeled after biological processes. Although there is a theoretical foundation for the convergence of genetic algorithms formulated in this way, in practice most problems do not fit naturally into this paradigm. Modern research has shown that optimizations can be set up by using the natural solution domain (for example, a real vector or integer sequence) and applying crossover and mutation operators analogous to the traditional genetic operators, but more appropriate to the natural formulation of the problem. This is the approach, sometimes called *evolutionary computing*, taken in the GA procedure. It enables you to model your problem by using a variety of solution forms including sequences, integer or real vectors,

Boolean encodings, and combinations of these. The GA procedure also provides you with a choice of genetic operators appropriate for these encodings, while permitting you to write your own.

The GA procedure enables you to implement the basic genetic algorithm by default, and to employ other advanced techniques to handle constraints, accelerate convergence, and perform multiobjective optimizations. These advanced techniques are discussed in the section "Details: GA Procedure" on page 115.

Although genetic algorithms have been demonstrated to work well for a variety of problems, there is no guarantee of convergence to a global optimum. Also, the convergence of genetic algorithms can be sensitive to the choice of genetic operators, mutation probability, and selection criteria, so that some initial experimentation and fine-tuning of these parameters is often required.

Getting Started: GA Procedure

The optimization problem is described by using programming statements, which initialize problem data and specify the objective, genetic operators, and other optimization parameters. The programming statements are executed once, and are followed by a RUN statement to begin the optimization process. The GA procedure enables you to define subroutines and designate them to be called during the optimization process to calculate objective functions, perform genetic mutation or crossover operations, or monitor and control the optimization. All variables created within a subroutine are local to that routine; to access a global variable defined within the GA procedure, the subroutine must have a parameter with the same name as the variable.

To set up a genetic algorithm optimization, your program needs to perform the following steps:

1. Initialize the problem data, such as cost coefficients and parameter limits.
2. Specify five basic optimization parameters:
   - *Encoding*: the general structure and form of the solution
   - *Objective*: the function to be optimized
   - *Selection*: how members of the current solution generation are chosen to propagate the next generation
   - *Crossover*: how the attributes of parent solutions are combined to produce new offspring solutions
   - *Mutation*: how random variation is introduced into the new offspring solutions to maintain genetic diversity
3. Generate a population of solutions for the initial generation.
4. Control the execution of the algorithm and record your results.

The following sections discuss each of these items in detail.

Initializing the Problem Data

The GA procedure offers great flexibility in how you initialize the problem data. Either you can read data from SAS data sets that are created from other SAS procedures and DATA steps, or you can initialize the data with programming statements.

In the PROC GA statement, you can specify up to five data sets to be read with the DATA*n*= option, where *n* is a number from 1 to 5, that can be used to initialize parameters and data vectors applicable to the optimization problem. For example, weights and rewards for a knapsack problem could be stored in the variables WEIGHT and REWARD in a SAS data set. If you specify the data set with a DATA1= option, the arrays WEIGHT and REWARD are initialized at the start of the procedure and are available for computing the objective function and evaluating the constraints with program statements. You could store the number of items and weight limit constraint in another data set, as illustrated in the sample programming statements that follow:

```
data input1;
   input weight reward;
   datalines;
1    5
2    3
4    7
1    2
8    3
6    9
2    6
4    3
...
;


data input2;
   input nitems limit;
   datalines;
10  20
;

proc ga data1 = input1  /* creates arrays weight and reward */
        data2 = input2; /* creates variables nitems and limit */

function objective( selected[*], reward[*], nitems);
   array x[1] /nosym;
   call dynamic_array(x, nitems);
   call ReadMember(selected,1,x);
   obj = 0;
   do i=1 to nitems;
     obj = obj + reward[x[i]];
   end;
   return(obj);
endsub;

[Other statements follow]
```

With these statements, the DATA1= option first establishes the arrays weight and reward from the data set input1, and the DATA2= option causes the variables nitems and limit to be created and initialized from the data set input2. The reward array and the nitems variable are then used in the objective function.

For convenience in initializing two-dimensional data such as matrices, the GA procedure provides you with the MATRIX*n*= option, where *n* is a number from 1 to 5. A two-dimensional array is created within the GA procedure with the same name as the option, containing the numeric data in the specified data set. For example, a table of distances between cities for a traveling salesman problem could be stored as a SAS data set, and a MATRIX1= option specifying that data set would cause a two-dimensional array named MATRIX1 to be created containing the data at the start of the GA procedure. This is illustrated in the following program:

```
data distance;
   input d1-d10;
   datalines;
0 5 3 1 2 ...
5 0 4 2 6 ...
3 4 0 1 3 ...
...
;

proc ga matrix1 = distance;
ncities = 10;
call SetEncoding('S10');
call SetObj('TSP','distances',matrix1);

[Other statements follow]
```

In this example, the data set distance is used to create a two-dimensional array matrix1, where matrix1[i, j] is the distance from city i to city j. The GA procedure provides a simple traveling salesman Problem (TSP) objective function, which is specified by the user with the SetObj call. The distances between locations are specified with the distances property of the TSP objective, which is set in the call to be matrix1. Note that when a MATRIX*n*= option is used, the names of variables in the data set are not transferred to the GA procedure as they are with a DATA*n*= option; only the numeric data are transferred.

You can also initialize problem data with programming statements. The programming statements in the GA procedure are executed before the optimization process begins. The variables created and initialized can be used and modified as the optimization progresses. The programming statement syntax is much like the SAS DATA step, with a few differences (see the section "Syntax: GA Procedure" on page 89). Special calls are described in the next sections that enable you to specify the objective function and genetic operators, and to monitor and control the optimization process. In the following program, a two-dimensional matrix is set up with programming statements to provide the distances for a 10-city symmetric traveling salesman problem, between locations specified in a SAS data set:

```
data positions;
   input x y;
   datalines;
100 230
50  20
150 100
...
;
```

```
proc ga data1 = positions;

call SetEncoding('S10');
ncities = 10;

array distance[10,10] /nosym;

do i = 1 to ncities;
   do j = 1 to i;
      distance[i,j] = sqrt((x[i]-x[j])**2 + (y[i] - y[j])**2);
      distance[j,i] = distance[i,j];
   end;
end;

call SetObj('TSP','distances', distance);
```

In this example, the DATA1= option creates arrays x and y containing the coordinates of the cities in an *x*-*y* grid, read in from the positions data set. An ARRAY programming statement creates a matrix of distances between cities, and the loops calculate Euclidean distances from the position data. The ARRAY statement is used to create internal data vectors and matrices. It is similar to the ARRAY statement used in the SAS DATA step, but the /NOSYM option is used in this example to set up the array without links to other variables. This option enables the array elements to be indexed more efficiently and the array to be passed efficiently to subroutines. You should use the /NOSYM option whenever you are creating an array that might be passed as an argument to a function or call routine.

Choosing the Problem Encoding

Problem encoding refers to the structure or type of solution space that is to be optimized, such as real-valued fixed-length vectors or integer sequences. The GA procedure offers encoding options appropriate to several types of optimization problems. You specify the problem encoding with a SetEncoding CALL statement,

```
call SetEncoding('encoding');
```

where the *encoding* string is a letter followed by a number, which specifies the type of encoding and the number of elements. The permitted letters and corresponding types of encoding are as follows:

R or *r*: real-valued vector. This type of encoding is used for general non-linear optimization problems.

I or *i*: integer-valued vector. This encoding is used for integer-valued problems. The integer size is 32 bits, which imposes a maximum value of 2,147,483,647 and a minimum value of –2,147,483,648 for any element of the vector. The integer vector encoding can also be used to represent bit vectors, where the 0–1 value of each bit in the integer vector is treated as a separate element. An example might be an assignment problem, where the positions within the vector represent different tasks, and the integer values represent different machines or other resources that might be applied to each task.

B or *b*: Boolean vector. Each element represents one true/false value.

S or *s*: sequence or permutation. In this encoding, each solution is composed of a sequence of integers ranging from 1 to the number of elements, with different solutions distinguished by different

orderings of the elements. This encoding is commonly used for routing problems such as the traveling salesman problem or for scheduling problems.

For example, the following statement specifies a 10-element integer vector encoding:

```
call SetEncoding('I10');
```

For problems where the solution form requires more than one type of encoding, you can specify multiple encodings in the *encoding* string. For example, if you want to optimize the scheduling of 10 tasks and the assignment of resources to each task, you could use a *segmented* encoding, as follows:

```
call SetEncoding('I10S10');
```

Here the I10 (10-element integer vector) is assigned to the first segment, and represents the resource assignment. The S10 (10-element sequence) is assigned to a second segment, and represents the sequence of tasks. The use of segmented encodings is described in the section "Using Multisegment Encoding" on page 115.

Setting the Objective Function

Before executing a genetic algorithm, you must specify the objective function to be optimized. Either you can define a function in your GA procedure input and designate it to be your objective function with the SetObjFunc call, or you can specify an objective function that the GA procedure provides with a SetObj call. The GA procedure currently supports the traveling salesman problem objective.

Controlling the Selection Process

There are two competing factors that need to be balanced in the selection process: *selective pressure* and *genetic diversity*. Selective pressure, the tendency to select only the best members of the current generation to propagate to the next, is required to direct the genetic algorithm to an optimum. Genetic diversity, the maintenance of a diverse solution population, is also required to ensure that the solution space is adequately searched, especially in the earlier stages of the optimization process. Too much selective pressure can lower the genetic diversity so that the global optimum is overlooked and the genetic algorithm converges prematurely. Yet, with too little selective pressure, the genetic algorithm might not converge to an optimum in a reasonable time. A proper balance between the selective pressure and genetic diversity must be maintained for the genetic algorithm to converge in a reasonable time to a global optimum.

The GA procedure uses a standard technique for the selection process commonly known as *tournament selection*. In general, the tournament selection process randomly chooses a group of members from the current population, compares their fitness, and selects the fittest from the group to propagate to the next generation. Tournament selection is one of the fastest selection methods, and it offers good control over the selection pressure.

You can control the selective pressure by specifying the tournament size, the number of members chosen to compete in each tournament. This number should be 2 or greater, with 2 implying the weakest selection pressure. Tournament sizes from 2 to 10 have been successfully applied to various genetic algorithm

optimizations, with sizes over 4 or 5 considered to represent strong selective pressure. This selection option is chosen with the following SetSel call:

```
call SetSel('tournament', 'size', size);
```

where *size* is the desired tournament size.

For tournament size of 2, you can further weaken the selective pressure by specifying a probability for selecting the most fit solution from the 2 competing solutions. By default this probability is 1, but the GA procedure permits you to set it to a value between 0.5 (equivalent to pure random selection) and 1. This selection option is chosen with the following SetSel call:

```
call SetSel('duel', 'pbest', bestprob);
```

where *bestprob* is the probability for choosing the most fit solution. This option can prove useful when conventional tournament selection tends to result in premature convergence.

One potential problem with tournament selection is that it does not guarantee that the best solution in the current generation is passed on to the next. To resolve this problem, the GA procedure enables you to specify an *elite* parameter, which ensures that the very best solutions are passed on to the next generation unchanged by mutation or crossover. Use the SetElite call:

```
call SetElite(elite);
```

where *elite* is an integer greater than or equal to 0. The GA procedure preserves the *elite* best solutions in the current generation and ensures that they are passed on to the next generation unchanged. When writing out the final solution population, the first *elite* members are the best of the generation and are sorted by their fitness, such that the fittest is first. By default, if you do not call SetElite in your program, an *elite* value of 1 is used. Setting the *elite* parameter to a higher number accelerates the convergence of the genetic algorithm; however, it can also lead to premature convergence before reaching a global optimum, so it should be used with care.

If you do not call SetSel in your input, then the default behavior for the GA procedure is to use tournament selection with size 2.

Setting Crossover Parameters

There are two crossover parameters that need to be specified: the crossover probability and the crossover operator. Members of the current generation that have passed the selection process either go to the crossover operator or are passed unchanged into the next generation, according to the crossover probability. To set the probability, you use a SetCrossProb call:

```
call SetCrossProb(prob);
```

where *prob* is a real number between 0 and 1. A value of 1 implies that the crossover operator is always applied, while 0 effectively turns off crossover. If you do not explicitly set the crossover probability with this call, a default value of 0 is used.

The GA procedure enables you to choose your crossover operator from several standard crossover operators appropriate for each type of encoding, or to code your own crossover operator as a subroutine. To specify one of the crossover operators provided by the GA procedure, use a SetCross call. See the section "Crossover Operators" on page 116 for more detail on the available operators. To supply your own operator, use a SetCrossRoutine call:

```
call SetCrossRoutine('routine');
```

where *routine* is the name of your crossover subroutine. See the section "Defining User Genetic Operators" on page 126 for a description of defining genetic operators with user subroutines. If the crossover probability is greater than 0, you must use a SetCross or SetCrossRoutine call to set the crossover operator.

After initialization, you can reset any of the crossover parameters or their properties during the optimization process by calling one of the preceding routines from a user update routine. This makes it possible to adapt the crossover operators as desired in the optimization process.

Setting Mutation Parameters

There are two mutation parameters: the mutation probability and the mutation operator. Members of the next generation are chosen to undergo mutation with the mutation probability you specify. To set the mutation probability, you use a SetMutProb call:

```
call SetMutProb(prob);
```

where *prob* is a real number between 0 and 1. This probability is usually fairly low (0.05 or less), since mutation tends to slow the convergence of the genetic algorithm. If you do not explicitly set the mutation probability with this call, a default value of 0 is used.

The GA procedure enables you to choose your mutation operator from several standard mutation operators appropriate for each type of encoding, or to code your own mutation operator as a subroutine. To specify one of the mutation operators provided by the GA procedure, use a SetMut call. See the section "Mutation Operators" on page 122 for more detail on the available operators. To supply your own operator, use a SetMutRoutine call:

```
call SetMutRoutine('routine');
```

where *routine* is the name of your mutation subroutine. See the section "Defining User Genetic Operators" on page 126 for a description of defining genetic operators with user subroutines. If the mutation probability is greater than 0, you must use a SetMut or SetMutRoutine call to set the mutation operator.

After initialization, you can reset any of the mutation parameters or their properties during the optimization process by calling one of the preceding routines from a user update routine. This makes it possible to adapt the mutation operators to create more or less diversity as needed in the optimization process.

Creating the Initial Generation

The last step in the initialization for the genetic algorithm optimization is to create the initial solution population, the first generation. The GA procedure provides you with the Initialize call to accomplish this task. The procedure provides several options for initializing the first population or reinitializing the solution population during the optimization process. For example, you can specify a data set in the FIRSTGEN= option of the PROC GA statement to be read to populate the initial generation, and use the initialize call:

```
call Initialize('_dataset_', PopulationSize);
```

Other possible initialization options include generating solutions uniformly distributed over the solution domain, executing a user-defined initialization routine, carrying over a portion of the previous population (for

reinitialization), or any combination of those actions. See the section "Initialize Call" on page 96 for a full explanation of initialization actions.

Monitoring Progress and Reporting Results

The GA procedure enables your program to monitor and alter parameters during the optimization process and record final results.

If a data set is specified in the LASTGEN= option of the PROC GA statement, then the last generation of solutions is written to the data set. See the section "PROC GA Statement" on page 90 for a description of the data set created by the LASTGEN= option.

You can define a subroutine and designate it to be called at each iteration in an update phase, which occurs after the evaluation phase and before selection, crossover, and mutation. Your subroutine can check solution values and update and store variables you have defined, adjust any of the optimization parameters such as the mutation probability or *elite* value, or check termination criteria and end the optimization process. An update routine can be especially helpful in implementing advanced techniques such as multiobjective optimization. You can specify an update subroutine with a SetUpdateRoutine call:

```
call SetUpdateRoutine('routine');
```

where *routine* is the name of your subroutine to be executed at each iteration.

You can set the maximum number of iterations permitted for the optimization process with the MAXITER= option in the PROC GA statement. If none is specified, a default value of 500 iterations is used. You can also control the number of iterations dynamically in your program by using the ContinueFor call:

```
call ContinueFor(n);
```

where *n* is the number of iterations to permit beyond the current iteration. A value of 0 ends the optimization process at the current iteration. One common way this call might be used is to include it in the logic of an update subroutine declared in the SetUpdateRoutine call. The update subroutine could check the objective values and end the optimization process when the optimal value of the objective function has not improved for a specific number of iterations. A ContinueFor call overrides an iteration limit set with the MAXITER= option.

To perform post processing of the optimization data, you can use a SetFinalize call to instruct the GA procedure to call a subroutine you have defined, after the last iteration:

```
call SetFinalize('routine');
```

where *routine* is the name of a subroutine you have defined. Your finalize subroutine could perform some post processing tasks, such as applying heuristics or a local optimization technique to try to improve the final solution.

A Simple Example

The example that follows illustrates the application of genetic algorithms to function optimization over a real-valued domain. It finds the minimum of the Shubert function:

$$\left[\sum_{i=1}^{5} i\cos[(i+1)x_1+i]\right]\left[\sum_{i=1}^{5} i\cos[(i+1)x_2+i]\right]$$

where $-10 \le x_i \le 10$ for $i = 1, 2$.

```
proc ga seed = 12 maxiter = 30;
/* the objective function to be optimized */
function shubert(selected[*]);
   array x[2] /nosym;
   call ReadMember(selected,1,x);
   x1 = x[1];
   x2 = x[2];
   sum1 = 0;
   do i = 1 to 5;
      sum1 = sum1 + i * cos((i+1)* x1 + i);
   end;
   sum2 = 0;
   do i = 1 to 5;
      sum2 = sum2 + i * cos((i+1) * x2 + i);
   end;
   result = sum1 * sum2;
   return(result);
endsub;
/* Set the problem encoding */
call SetEncoding('R2');
/* Set upper and lower bounds on the solution components */
array LowerBound[2] /nosym (-10 -10);
array UpperBound[2] /nosym (10 10);
call SetBounds(LowerBound, UpperBound);
/* Set the objective function */
call SetObjFunc('shubert',0);
/* Set the crossover parameters */
call SetCrossProb(0.65);
call SetCross('Heuristic');
/* Set the mutation parameters */
call SetMutProb(0.15);
array del[2] /nosym (0.2 0.2);
call SetMut('Delta','nchange', 1, 'delta',del);
/* Set the selection criteria */
call SetSel('tournament','size', 2);
call SetElite(2);
/* Initialize the first generation, with 120 random solutions  */
call Initialize('DEFAULT',120);
/* Now execute the Genetic Algorithm */
run;
quit;
```

At the beginning of the program, the PROC GA statement sets the initial random number seed and sets the maximum number of iterations to 30.

A routine to compute the objective function (`function shubert`) is then defined. This function is called by the GA procedure once for each member of the solution population at each iteration. Note that the GA procedure passes the array selected as the first parameter of the function, and the function uses that array to

obtain the selected solution elements with a ReadMember call, which places the solution in the array x. The second parameter of the ReadMember call is 1, specifying that segment 1 of the solution be returned, which in this case is the only segment. The programming statements that follow compute the value of the objective function and return it to the GA procedure.

After the function definition, the 'R2' passed to the SetEncoding call specifies that solutions are single-segment, with that segment containing two elements that are real-valued. Next, a lower bound of –10 and an upper bound of 10 are set for each solution element with the SetBounds call. The SetObjFunc call specifies the previously defined Shubert function as the objective function for the optimization; the second parameter value of 0 indicates that a minimum is desired. The SetCrossProb call sets the crossover probability to 0.65, which means that, on average, 65% of the solutions passing the selection phase will undergo the crossover operation. The crossover operator is set to the heuristic operator by the SetCross call. Similarly, the mutation probability is set to 0.15 with the SetMutProb call, and the delta operator is set as the mutation operator with the SetMut call. The selection criteria are then set: a conventional tournament of size 2 is specified with SetSel call, and an *elite* value of 2 is specified with the SetElite call. The *elite* value implies that the best two solutions of each generation are always carried over to the next generation unchanged by mutation or crossover. The last step before beginning the optimization is the Initialize call. This call sets the population size at 120, and specifies the default initialization strategy for the first population. For real encoding, this means that an initial population randomly distributed between the upper and lower bounds specified in the SetBounds call is generated. Finally, when the RUN statement is encountered, the GA procedure begins the optimization process. It iterates through 30 generations, as set by the MAXITER= option.

The Shubert function has 760 local minima, 18 of which are global minima, with a minimum of –186.73. If you experiment with different random seeds with the SEED= option, PROC GA generally converges to a different global minimum each time. Figure 4.1 shows the output for the chosen seed.

Figure 4.1 Shubert Function Example Output

PROC GA Optimum Values

Objective
-186.7309031

Solution	
Element	Value
1	-7.708309818
2	-0.800371886

Syntax: GA Procedure

To initialize your data and describe your model, you use programming statements with a syntax similar to the SAS DATA step, augmented with some special function calls to communicate with the genetic algorithm optimizer. Most of the programming statements used in the SAS DATA step can be used in the GA procedure, and these are described fully in the *SAS Statements: Reference* and Base SAS documentation. Following is an alphabetical list of the statements and special function calls used.

PROC GA *options* **;**
ContinueFor Call ;
Cross Call ;
Dynamic_array Call ;
EvaluateLC Call ;
GetDimensions Call ;
GetObjValues Call ;
GetSolutions Call ;
Initialize Call ;
MarkPareto Call ;
Mutate Call ;
Objective Call ;
PackBits Call ;
Program Statements ;
ReadChild Call ;
ReadCompare Call ;
ReadMember Call ;
ReadParent Call ;
ReEvaluate Call ;
SetBounds Call ;
SetCross Call ;
SetCrossProb Call ;
SetCrossRoutine Call ;
SetElite Call ;
SetEncoding Call ;
SetFinalize Call ;
SetMut Call ;
SetMutProb Call ;
SetMutRoutine Call ;
SetObj Call ;
SetObjFunc Call ;
SetProperty Call ;
SetSel Call ;
SetUpdateRoutine Call ;
ShellSort Call ;
Shuffle Call ;
UnpackBits Function ;
UpdateSolutions Call ;
WriteChild Call ;
WriteMember Call ;

PROC GA Statement

PROC GA *options* **;**

The PROC GA statement invokes the GA procedure. The following options are used with the PROC GA statement.

DATA*n*=*SAS-data-set***

specifies a data set containing data required to specify the problem, where *n* is an integer from 1 to 5. The data set is read and variables created matching the variables of the data set. If the data set has more than one observation, then the newly created variables are vector arrays with the size equal to the number of observations.

FIRSTGEN=*SAS-data-set*

specifies a SAS data set containing the initial solution generation. Different segments in the solution should be identified by variable names consisting of a letter followed by numbers representing the elements in the segments, in alphabetical order. For example, if the first segment of the solution uses real encoding and contains 10 elements, it should be represented by the numeric variables A1, A2, . . . , A10. A second segment with integer encoding and five elements would be specified in the variables B1, B2, . . . , B5. For Boolean encoding each Boolean element is represented by one variable in the data set, and for sequence encoding each position in the sequence is represented by one variable. If the data set contains a field named OBJECTIVE, then the value of that field becomes the objective value for that solution at initialization time (overriding the value computed by the input objective function), unless the field value is a missing value. The FIRSTGEN= and LASTGEN= options are designed to work together, so that a data set generated from a run of the GA procedure with a LASTGEN= option set can be specified in the FIRSTGEN= option of a subsequent run of the GA procedure, to continue the optimization from where it finished. If the number of observations in the data set is less than the population size specified in the Initialize call, additional members are generated as specified in the Initialize call to complete the population. This feature makes it easy to seed an initial randomly generated population with chosen superior solutions generated by heuristics or a previous run of the GA procedure. If the data set contains more observations than the population size, the population is filled starting at the first observation, and the additional observations are not used.

LASTGEN=*SAS-data-set*

specifies a SAS data set into which the final solution generation is written. Different segments in the solution are identified by variable names consisting of a letter followed by numbers representing the elements in the segments, in alphabetical order. For example, if the first segment of the solution uses real encoding and contains 10 elements, it would be represented by the numeric variables A1, A2, . . . , A10. A second segment with integer encoding and five elements would be specified in the variables B1, B2, . . . , B5. For Boolean encoding each Boolean element is represented by one variable in the data set, and for sequence encoding each position in the sequence is represented by one variable. In addition to the solutions elements, the final objective value for each solution is output in the OBJECTIVE variable. The FIRSTGEN= and LASTGEN= options are designed to work together, so that a data set generated with a LASTGEN= option can be specified in the FIRSTGEN= option of a later run of the GA procedure.

LIBRARY=*library-list*

specifies a library or group of libraries for the procedure to search to resolve subroutine or function calls. The libraries can be created by using PROC FCMP and PROC PROTO. This option supplements the action of the CMPLIB= SAS option, but it permits you to designate libraries specific to this procedure invocation. Use the *libref.catalog* format to specify the two-level name of the library; *library-list* can be either a single library, a range of library names, or a list of libraries. The following examples demonstrate the use of the LIBRARY= option.

```
proc ga library = sasuser.xlib;
proc ga library = xlib1-xlib5;
```

```
proc ga library = (sasuser.xlib xlib1-xlib5 work.example);
```

MATRIX*n*=*SAS-data-set*
specifies a data set containing two-dimensional matrix data, where *n* is an integer from 1 to 5. A two-dimensional numeric array with the same name as the option is created and initialized from the data set. This option is provided to facilitate the input of tabular data to be used in setting up the optimization problem. Examples of data that might be provided by this option include a distance matrix for a traveling salesman problem or a matrix of coefficients for linear constraints.

MAXITER=*n*
specifies the maximum number of iterations to permit for the optimization process. A ContinueFor call overrides a limit set by this option.

NOVALIDATE=*n*
controls the amount of solution validity checking performed by the procedure. By default, the procedure verifies that valid solutions are being supplied at initialization and when the solution population is being updated by genetic operators or other user actions. If a solution segment has elements that exceed bounds set by a SetBounds call, those elements will be reset to the bound and a warning will be issued. If a solution segment contains other illegal values, an error will be signaled. This action is useful for maintaining the validity of the optimization and avoiding some errors that are often difficult to trace. However, this activity does consume CPU time, and you might want to use a strategy where you generate infeasible solutions at initialization or via genetic operators and then repair the solutions later in an update or objective routine. The NOVALIDATE= option enables you to do so, by turning off the validation checks. If *n* is 1, validation is turned off for initialization only, and if *n* is 2, validation is turned off for update only. If *n* is 3, all solution validity checking is turned off.

NOVALIDATEWARNING=*n*
controls the output of warning messages related to solution validation checking. Depending on the value of the NOVALIDATE= option, warning messages will be issued when the procedure repairs initial or updated solution segments to fit within bounds set with the SetBounds call. If *n* is 1, validation warnings are turned off for initialization only; and if *n* is 2, validation warnings are turned off for update only. If *n* is 3, all solution validation warnings are turned off.

SEED=*n*
specifies an initial seed to begin random number generation. This option is provided for reproducibility of results. If it is not specified, or if it is set to 0, a seed is chosen based on the system clock. The SEED value should be a nonnegative integer less than $2^{31} - 1$.

ContinueFor Call

call ContinueFor *(niter)* **;**

The ContinueFor call sets the number of additional iterations for the genetic algorithm optimization. The input to the ContinueFor subroutine is as follows:

niter specifies that the optimization continue for *niter* more iterations. To stop at the current iteration, set *niter* to 0.

Cross Call

call Cross *(selected, seg, type< , parameter1, parameter2, . . . >)* **;**

The Cross call executes a genetic crossover operator from within a user subroutine. The inputs to the subroutine are as follows:

selected	is an array that specifies the solutions to be crossed.
seg	is the desired segment of the solution to which the crossover operator should be applied.
type	is the type of crossover operator to apply, which also determines the number and type of parameters expected.
parameter1-n	are optional parameters applicable to some operators.

The accepted values for *type* and the corresponding parameters are summarized in Table 4.2.

Table 4.2 Crossover Operator Types

Type	Encodings	Parameters
'arithmetic'	real, integer	
'cycle'	sequence	
'heuristic'	real	
'null'	all encodings	
'order'	sequence	
'pmatch'	sequence	
'simple'	real, integer, Boolean	*alpha*
'twopoint'	real, integer, Boolean	*alpha*
'uniform'	real, integer, Boolean	*alpha*, *p*

The parameters are as follows:

alpha	is a number such that $0 < alpha \leq 1$.
p	is a probability such that $0 < p \leq 0.5$.

The Cross call should be made only from within a user crossover subroutine. The precise action of these crossover operators is described in the section "Crossover Operators" on page 116.

Dynamic_array Call

call Dynamic_array *(arrayname, dim1< , dim2, . . . , dim6>)* **;**

The Dynamic_array call allocates a numeric array. The inputs to the Dynamic_array call are as follows:

arrayname	is a previously declared array, whose dimensions are to be reallocated.
dim1	is the size of the first dimension.
dim2,. . . ,dim6	are optional. Up to six dimensions can be specified.

The Dynamic_array call is normally used to allocate working arrays when the required size of the array is data-dependent. It is often useful in user routines for genetic operators or objective functions to avoid hard-coding array dimensions that might depend on segment length or population size. The array to be allocated must first be declared in an ARRAY statement with the expected number of dimensions, as in the following example:

```
subroutine sub(nx, ny);
   array x[1] /nosym;
   call dynamic_array(x, nx);
   array xy[1,1] /nosym;
   call dynamic_array(xy, nx, ny);
   ...
```

EvaluateLC Call

call EvaluateLC *(lc, results, sum, selected, seg< , child>)* **;**

The EvaluateLC call evaluates linear constraints. The inputs to the EvaluateLC subroutine are as follows:

lc	is a two-dimensional array representing the linear constraints.
results	is a numeric array to receive the magnitude of the constraint violation for each linear constraint.
sum	is a variable to receive the sum of the constraint violations over all the constraints.
selected	is an array identifying the selected solution.
seg	is the segment of the solution to which the linear constraints apply.
child	is an optional parameter, and should be specified only when EvaluateLC is called from a user crossover operator.

The EvaluateLC routine can be called from a user crossover operator, mutation operator, or objective function to determine if a solution violates linear inequality constraints of the form $Ax \leq b$. For n linear constraints in m variables, the *lc* array should have dimension $n \times (m+1)$. For each linear constraint $i = 1, \ldots, n$, $lc[i, j] = A[i, j]$ for $j = 1, \ldots, m$, and $lc[i, m+1] = b[i]$. The *results* array should be one-dimensional with size n. The EvaluateLC call fills in the elements of *results* such that

$$results[i] = \begin{cases} 0, & \text{if } \sum_{j=1}^{m} A[i,j]x[j] \leq b[i] \\ \sum_{j=1}^{m} A[i,j]x[j] - b[i], & \text{otherwise} \end{cases}$$

In the variable *sum*, the EvaluateLC call returns the value $\sum_{i=1}^{n} results[i]$. Note that $sum \geq 0$, and *sum*=0 implies that no constraints are violated. When you call EvaluateLC from your user routine, the *selected* parameter of the EvaluateLC call must be the same as the first parameter passed to your user routine to properly identify the solution to be checked. The *seg* parameter identifies which segment of the solution

should be checked. Real, integer, or Boolean encodings can be checked with this routine. If EvaluateLC is called from a user crossover operator, the *child* parameter must be specified to indicate which offspring is to be checked. The value *child* = 1 requests the first offspring, *child* = 2 requests the second, and so on.

GetDimensions Call

call GetDimensions *(source, dest)* **;**

The GetDimensions call gets the dimensions of an array variable. The inputs to the GetDimensions subroutine are as follows:

source is the array variable whose dimensions are desired.

dest is an array to receive the dimensions of *source*.

The GetDimensions subroutine is used to get the dimensions of an array passed into a user subroutine. The input *dest* should be an array of one dimension, with at least as many elements as there are dimensions in *source* (a maximum of 6). Any extra elements in *dest* are filled with zeros.

GetObjValues Call

call GetObjValues *(dest, n)* **;**

The GetObjValues call retrieves objective function values from the current solution generation. The inputs to the GetObjValues subroutine are as follows:

dest is an array to receive the objective values.

n is the number of objective values to get.

The GetObjValues subroutine is used to retrieve the objective values for the current solution generation. It can be called from a user update routine or finalize routine. If it is called from a finalize routine, and if the *elite* parameter from a SetElite call is 1 or greater, then the first *elite* members of the population are the fittest of the population, and they are sorted in order, starting with the most fit. The input *dest* should be a dimensioned variable, with dimension greater than or equal to *n*.

GetSolutions Call

call GetSolutions *(sol, n, seg)* **;**

The GetSolutions call retrieves solutions from the current generation. The inputs to the GetSolutions subroutine are as follows:

sol is an array to receive the solution elements.

n is the number of solutions to get.

seg is the segment of the solution to retrieve.

The GetSolutions subroutine is used to retrieve solutions from the current generation. You would normally call it from an update or finalize subroutine for post processing or analysis. If the *elite* parameter has been set with a SetElite call, then the first *elite* members of the population are the fittest, and they are sorted

in order, starting with the most fit. The *sol* variable should have two dimensions, with the first dimension representing the solution number, and the second representing the element within the solution. For example, if the encoding of the problem was I10, then *sol*[2, 3] would be the value of the third element of the second solution in the current population. For real, integer, Boolean, and sequence encoding, each solution element is mapped to the corresponding element of the *sol* array. The *seg* parameter specifies the solution segment desired. For example, if the encoding was set in the SetEncoding call to 'R10I5', then segment 1 is R10 and segment 2 is I5.

Initialize Call

call Initialize *(option, size <,option, size> . . .)* **;**

The Initialize call creates the initial solution generation. The inputs to the Initialize subroutine are as follows:

option is a string that specifies an initialization option.

size is the number of solutions to create by using a given option.

The Initialize subroutine must be called to create the first solution generation, and can be used to reinitialize a solution population during the optimization process. The available options and their effect are as follows:

'_uniform_' generate uniformly distributed solutions —for integer and real encoded segments for which SetBounds has been called, segment elements will be uniformly distributed between the upper and lower bounds. If no bounds have been specified for an integer or real encoded segment, then the elements will be set to 0. For Boolean encoded segments the elements will be randomly assigned 0 or 1, and for sequence encoded segments random sequences will be generated.

'_dataset_' read solutions from the data set specified in a FIRSTGEN= option. If the data set has more observations than requested, the extra observations are ignored.

'default' read solutions from the data set specified in a FIRSTGEN= option, if one was specified. If none was specified or the data set has fewer observations than requested, fill in the remaining solution population by using the '_uniform_' option.

'_retain_' bring forward the best solutions from the current generation. This option cannot be used for the first initialization.

'*user-routine*' Any string not matching the preceding options is interpreted to be a user-defined initialization routine. See the section "Defining a User Initialization Routine" on page 131 for information about defining an initialization subroutine.

After the Initialize call, the current solution population size is the sum of the population sizes specified for each option. The following rules also apply to the option specifications:

1. All options must be literal quoted strings.

2. No option type can be specified more than once in the same Initialize call. No more than one user initialization routine can be specified.

3. 'default' cannot be specified in combination with the '_uniform_' or '_dataset_' options.

4. If the '_uniform_' option is specified, the solution encoding must include at least one segment that is either Boolean or sequential, or that has bounds specified with a SetBounds call.

MarkPareto Call

call MarkPareto *(result, n, objectives, minmax)* **;**

The MarkPareto call identifies the Pareto-optimal set from a population of solutions. The inputs to the MarkPareto call are as follows:

result	is a one-dimensional array to accept the results of the evaluation. Its size should be the same as the size of the population being evaluated; $result[i] = 1$ if solution i is Pareto optimal, and 0 otherwise.
n	is a variable to receive the number of Pareto-optimal solutions.
objectives	is a two-dimensional array that contains the multiple objective values for each solution. It should be dimensioned $[p, q]$, where p is the size of the population, and q is greater than or equal to the number of objectives to be considered.
minmax	is a one-dimensional array to specify how the objective values are to be used. It should be of size q, where q is greater than or equal to the number of objectives to be considered. $minmax[k] = -1$ if objective k is to be minimized, $minmax[k] = 1$ if objective k is to be maximized, $minmax[k] = 0$ if objective k is not to be considered, and $minmax[k] = -2$ designates an objective that prevents the member from being considered for Pareto optimality if it is nonzero.

The MarkPareto call is used to identify the Pareto-optimal subset from a population of solutions. See the section "Optimizing Multiple Objectives" on page 135 for a full discussion of Pareto optimality. MarkPareto can be called from a user update routine, which is called after the individual solution objective values have been calculated and before selection takes place. To make best use of this routine, in your encoding you need to set up a segment to record all the objective values you intend to use in the Pareto-optimal set evaluation. In a user objective function, you should calculate the multiple objectives and write them to the chosen segment. In an update routine (designated with a SetUpdateRoutine call), you can use a GetSolutions call to retrieve these segments, and then pass them to a MarkPareto call. The following statements shows how this could be done:

```
subroutine update(popsize);

   array objectives[1,1] /nosym;
   call dynamic_array(objectives, popsize, 3);

   array pareto[1] /nosym;
```

```
   call dynamic_array(pareto, popsize);

   array minmax[3] /nosym (1 -1 0);

   call GetSolutions(objectives, popsize, 2);

   call MarkPareto(pareto, npareto, objectives, minmax);

   do i = 1 to popsize;
      objectives[i,3] = pareto[i];
   end;

   call UpdateSolutions(objectives, popsize, 2);

   call SetElite(npareto);

endsub;
```

This is an example of a user update routine that might be used in a multiobjective optimization problem. It is assumed that a user objective function has calculated two different objectives and placed their values in the first two elements of segment 2 of the problem encoding. The first objective is to be maximized, and the second is to be minimized. Segment 2 has three elements, and the third element is used to mark the Pareto-optimal solutions. After dynamically allocating the necessary arrays from the *popsize* (population size) parameter, the update routine first retrieves the current solutions into the *objectives* array with the GetSolutions call. It then passes the *objectives* array directly to the MarkPareto call to perform the Pareto-optimal evaluations. Note that the *minmax* array directs the MarkPareto call to maximize the first element, minimize the second, and ignore the third. After the MarkPareto call, the update routine writes the results back to the third element of the *objectives* array, and writes the *objectives* array back to the solution population with the UpdateSolutions call. This marks the solutions that compose the Pareto-optimal set. The update routine then sets the *elite* parameter equal to the number of Pareto-optimal solutions with the SetElite call. It is assumed that the user has provided a fitness comparison function (designated with a SetCompareRoutine call) that always selects a Pareto-optimal solution over a non-Pareto-optimal one, so the *elite* setting guarantees that all the Pareto-optimal solutions are retained from generation to generation. Example 4.3 illustrates the use of the MarkPareto call.

Mutate Call

call Mutate *(selected, seg, type< , parameter1, parameter2, . . . >)* **;**

The Mutate call executes a genetic mutation operator from within a user subroutine. The inputs to the subroutine are as follows:

selected	is an array that specifies the solution to be mutated.
seg	is the desired segment of the solution to which the mutation should be applied.
type	is the type of mutation operator to apply, which also determines the number and type of parameters expected.
parameter1-n	are optional parameters applicable to some operators.

The accepted values for *type* and the corresponding parameters are summarized in Table 4.3.

Table 4.3 Mutation Operator Types

Type	Encodings	Parameters
'delta'	real, integer	*delta*
		n
'invert'	sequence	
'swap'	sequence	*n*
'uniform'	real, integer, Boolean	*np*

The parameters are as follows:

delta is a vector of delta values for each component of the solution, used only for the Delta mutation operator.

n specifies the number of solution elements that should be mutated for the Delta operator, and the number of swaps that should be made for the Swap operator.

np specifies the number of solution elements that should be mutated, if *np* is integer; specifies the mutation probability for each solution element if $0 < np < 1$.

The Mutate call should be made only from within a user mutation subroutine. The precise action of these mutation operators is described in the section "Mutation Operators" on page 122.

Objective Function

r = Objective *(selected, seg, type< , parameter1, parameter2, . . . >)* **;**

The Objective call evaluates a standard objective function from within a user subroutine. The inputs to the function are as follows:

selected is an array that specifies the solution to be evaluated.

seg is the desired segment of the solution to be evaluated.

type is objective function name, which also determines the number and type of parameters expected.

parameter1-n are optional parameters applicable to particular objective functions.

The accepted values for *type* and the corresponding parameters are summarized in Table 4.4.

Table 4.4 Objective Function Types

type	encodings	parameters
'TSP'	sequence	*distances*

The parameters are as follows:

distances is a matrix of distances between locations, such that $distances[i, j]$ is the distance between location i and j.

The Objective call should be made only from within a user objective function. The precise actions of the standard objective functions are described in the section "Objective Functions" on page 124.

PackBits Call

call PackBits *(array, start, width, value)* **;**

The PackBits call writes bits to a packed integer array. The inputs to the PackBits subroutine are as follows:

array is an array to which the value is to be assigned.

start is the starting position for the bit assignments.

width is the number of bits to assign.

value is the value to be assigned to the bits. For a single bit, this should be 0 or 1.

The PackBits subroutine facilitates the assignment of bit values into an integer array, effectively using the integer array as an array of bits. One common use for this routine is within a user genetic operator subroutine to pack bit values into an integer vector solution segment. The bit values assigned with the PackBits call can be retrieved with the UnpackBits function.

The *start* parameter is the lowest desired bit position in the bit vector, where the least significant bit of *value* is to be stored. The *start* parameter can range in value from 1 to *maxbits*, where *maxbits* is the product of 32 times the number of elements in the integer array.

The *width* parameter is the number of bits of *value* to be stored. It is bounded by $0 < width < (maxbits - start + 1)$ and must also not exceed 32.

Bits not within the range defined by *start* and *width* are not changed. If the magnitude of *value* is too large to express in *width* bits, then the *width* least significant bits of *value* are transferred. The following program fragment, which might occur in a mutation subroutine, first reads a selected solution segment into `s` with the ReadMember call and then overwrites the first and second least significant bits of the solution with ones before writing it back to the current generation.

```
array s[2];
call ReadMember(selected, seg, s);
...
/* intervening code */
...
call PackBits(s, 1, 2, 3);/* start = 1, width = 2,
                           * value = 3 = binary 11
                           */
call WriteMember(selected, seg, s);
```

Programming Statements

This section lists the programming statements used to initialize the model, code the objective function, and control the optimization process in the GA procedure. It documents the differences between programming statements in the GA procedure and programming statements in the DATA step. The syntax of programming statements used in PROC GA is identical to that of programming statements used in the FCMP procedure.

Most of the programming statements that can be used in the SAS DATA step can also be used in the GA procedure. See the *SAS Statements: Reference* or BASE SAS documentation for a description of the SAS programming statements.

variable = *expression*;
variable + *expression*;
arrayvar[*subscript*] = *expression*;
ABORT;
CALL *subroutine* < *(parameter-1* <, *...parameter-n* > *)* >;
DELETE;
DO *program-statements;* END;
DO *variable* = *expression* TO *expression* <BY *expression*>;
program-statements; END;
DO WHILE *expression* ;
program-statements; END;
DO UNTIL *expression* ;
program-statements; END;
GOTO *statement_label* ;
IF *expression* THEN *program-statement*;
<ELSE *program-statement*>;
PUT < *variable(s)*> <@ | @@> ;
RETURN <(*expression*)>;
SELECT <(*select-expression*)>;
WHEN-*1* (*expression-1* <. . . ,*expression-n*>)*program-statement* ;
<WHEN-*n* (*expression-1* <. . . ,*expression-n*>)*program-statement* ;>
<OTHERWISE *program-statement* ;>
STOP;
SUBSTR(*variable, index, length*) = *expression*;

For the most part, the SAS programming statements work as they do in the SAS DATA step as documented in the *SAS Statements: Reference*. However, there are several differences that should be noted.

- The ABORT statement does not permit any arguments.
- The DO statement does not permit a character index variable. Thus

  ```
  do i = 1,2,3;
  ```

  is supported; however,

  ```
  do i = 'A','B','C';
  ```

  is not.

- The PUT statement, used mostly for program debugging in PROC GA, supports only some of the features of the DATA step PUT statement, and has some new features that the DATA step PUT statement does not:
  - The PROC GA PUT statement does not support line pointers, factored lists, iteration factors, overprinting, _INFILE_, the colon (:) format modifier, or "$".
  - The PROC GA PUT statement does support expressions, but the expression must be enclosed inside parentheses. For example, the following statement displays the square root of x: `put (sqrt(x));`
  - The PROC GA PUT statement permits an array name without subscripts. The statement `PUT A;` prints all the elements of array `A`. The statement `PUT A=;` prints the elements of array `A` with each value labeled with the name of the element variable.
  - The PROC GA PUT statement supports the print item _PDV_ to print a formatted listing of all variables in the program. For example, the following statement displays a more readable listing of the variables than the _all_ print item: `put _pdv_;`
- The WHEN and OTHERWISE statements permit more than one target statement. That is, DO/END groups are not necessary for multiple-statement WHENs. For example, the following syntax is valid:

```
SELECT;
WHEN ( exp1 )   stmt1;
                stmt2;
WHEN ( exp2 )   stmt3;
                stmt4;
END;
```

ReadChild Call

call ReadChild *(selected, seg, n, values)* **;**

The ReadChild call reads a segment from a selected child solution into an array, within a user crossover operator. The inputs to the ReadChild subroutine are as follows:

selected	specifies the family (parents and children) obtained from the selection process.
seg	specifies the solution segment to be read.
n	specifies the child in the family from which to read the solution segment.
values	specifies an array to receive the solution elements.

The ReadChild call is used to obtain the solution values for manipulation within a user crossover operator subroutine. Normally it is needed only if you need to augment the action of a GA procedure-supplied crossover operator. You might need to make modifications to satisfy constraints, for example. The *selected* parameter is passed into the user subroutine by the GA procedure. The *seg* parameter is the desired segment of the solution to be obtained. Segments, which correspond to different encodings in the encoding string, are numbered, starting from 1 as the first segment. The parameter *n* should be 1 to get the first child and 2 for the second. The parameter *values* is an array, which should be dimensioned large enough to contain the segment's

encoding. For example, the following subroutine illustrates how you could use the Read/WriteChild calls to modify offspring generated with a standard genetic operator:

```
call SetEncoding('R5');

subroutine cross(selected[*]);

/* generate offspring with arithmetic crossover operator */
call CrossArithmetic(selected, 1); /* here 1 refers to segment 1*/

array child1[5];
array child2[5];

/* get elements of first child solution */
call ReadChild(selected, 1, 1, child1);

/* get elements of second child solution values */
call ReadChild(selected, 1, 2, child2);

...
/* code to modify elements in child1 and child2 */
...

call WriteChild(selected,1,1,child1);
call WriteChild(selected,1,2,child2);
```

ReadCompare Call

call ReadCompare *(selected, seg, n, values)* **;**

The ReadCompare call reads a segment from a selected solution into an array, within a user fitness comparison subroutine. The inputs to the ReadCompare subroutine are as follows:

selected	specifies the pair of solutions to be compared, obtained from the selection process.
seg	specifies the solution segment to be read.
n	specifies the solution (1 or 2) from which to read the segment.
values	specifies an array to receive the solution elements.

The ReadCompare call is used to obtain the solution values for manipulation within a user fitness comparison subroutine, which can be designated in a SetCompareRoutine call.

ReadMember Call

call ReadMember *(selected, seg, destination)* **;**

The ReadMember call reads the selected solution into an array for a user objective function or mutation operator.

The inputs to the ReadMember subroutine are as follows:

selected is a parameter passed to the user subroutine by the GA procedure, which points to the selected solution.

seg specifies which segment of the solution to retrieve.

destination specifies an array in which to store the solution elements.

The ReadMember call is used within a user objective function or mutation operator to obtain the elements of a selected solution and write them into a specified vector. They can then be used to compute an objective value, or in the case of a mutation operator, manipulated and written back out with a WriteMember call.

ReadParent Call

call ReadParent *(selected, seg, n, destination)* **;**

The ReadParent call reads selected solution elements into an array in a user crossover subroutine. The inputs to the ReadParent subroutine are as follows:

selected is a parameter passed to the user subroutine by the GA procedure, which points to the selected solution family.

seg is the segment of the desired parent solution to be obtained.

n is the number of the parent, starting at 1.

destination is an array in which to store the solution elements.

The ReadParent subroutine is called inside a user crossover operator subroutine to obtain the elements of selected parent solutions. Normally you would then manipulate and combine the elements of the two parents and use a WriteChild call to create the child offspring and complete the action of the crossover operator.

ReEvaluate Call

call ReEvaluate *(< index >)* **;**

The ReEvaluate call reruns the evaluation phase of the genetic algorithm. The inputs to the ReEvaluate subroutine are as follows:

index is a numeric scalar or array that specifies the indices of the solutions to be updated. The indices correspond to the order of the solutions obtained from a GetSolutions call.

The ReEvaluate call recomputes the objective values for the current generation. You do not normally need to use this call, because the GA procedure evaluates the objective function during the optimization process in the evaluation phase. This subroutine should be called from a user update or finalize routine if a parameter that affects the objective value or solution is changed. The optional *index* parameter enables you to restrict the recomputation to the solution or subset of solutions specified. If the *index* parameter is not supplied, then the objective values of all the solutions will be recomputed.

For example, you might have a user objective function that can perform an additional local optimization if a particular parameter is set. If your update routine changes that parameter, then you should call the ReEvaluate subroutine to update the solutions and objective function values.

SetBounds Call

call SetBounds *(lower, upper < , seg >)* **;**

The SetBounds call sets constant upper and lower bounds. The inputs to the SetBounds subroutine are as follows:

lower	is a lower bound for the solution components.
upper	is an upper bound for the solution components.
seg	is optional, and specifies a segment of the solution to which the bounds apply. If *seg* is not specified, then it defaults to a value of 1.

The SetBounds subroutine is used to establish upper and lower bounds on the solution space. It applies only to integer and real encoding. For multiple segment encoding, use the *seg* parameter to specify a segment other than the first. *upper* and *lower* must be arrays, with the same dimension as the encoding size. SetBounds must be called for all integer or real encoded segments to which you apply the Uniform mutation operator. The action of the standard mutation and crossover operators supplied by the GA procedure is automatically modified so that the bounds established by a SetBounds call are respected. For an integer encoded segment, only integer values are allowed for the upper and lower bounds.

SetCompareRoutine Call

call SetCompareRoutine *('routine')* **;**

The SetCompareRoutine call installs a user function to compare the fitness of solutions. The input to the SetCompareRoutine subroutine is as follows:

routine	is the name of a function you have defined, which is called when necessary to compare the fitness of solutions. This parameter must be a string literal; a variable is not accepted.

The SetCompareRoutine call enables you to designate a function you have defined to be used in the selection process when comparing the fitness of two solutions. The selector options that involve ranking solutions by fitness, including tournament and duel selection, will use the designated function instead of comparing the objective values directly. If the SetCompareRoutine is not called, or if it is called with an input value of 'default', then the objective function value will be used to compare solution fitness. The SetCompareRoutine call provides you with a way to factor multiple fitness criteria into the selection process. See the section "Defining a User Fitness Comparison Routine" on page 124 for a full description of how the fitness comparison routine should be structured and what it should return. You can use this feature to implement multiobjective optimization and other advanced optimization strategies.

SetCross Call

call SetCross *(type< , seg >< , pname, pvalue >< , pname, pvalue >. . .)* **;**

The SetCross call sets the crossover operator. The inputs to the SetCross subroutine are as follows:

type is the name of the crossover operator to be applied.

seg is optional, and specifies a segment of the solution to which the operator should be applied. If *seg* is not specified, then it defaults to a value of 1. *seg* needs to be specified only if multisegment encoding is used.

pname is optional, and specifies the name of a particular property to be set for the crossover operator.

pvalue specifies the value to be assigned to the corresponding property name.

The SetCross routine is used to assign a standard crossover operator. For multisegment encoding, the operator can be assigned to a particular solution segment with the *seg* parameter; otherwise a default segment of 1 is assumed. You can set different crossover operators for different segments with multiple SetCross calls. When a crossover event occurs, all segments in the parent solutions for which a crossover operator has been designated will undergo crossover. If more than one SetCross call is made for the same segment, then the last call nullifies any previous call for that segment. Also, a SetCrossRoutine call nullifies all previous SetCross calls, and a SetCross call nullifies a previous SetCrossRoutine call. Properties for the chosen crossover operator can be set with optional *pname-pvalue* pairs. It is also possible to set or reset operator properties with a SetProperty call.

The accepted values for *type* and the corresponding properties are summarized in Table 4.5. See the section "Crossover Operators" on page 116 for a full description of the available operators.

Table 4.5 Crossover Operator Types and Properties

type	encodings	properties
'arithmetic'	real, integer	
'cycle'	sequence	
'heuristic'	real	
'order'	sequence	
'pmatch'	sequence	
'simple'	real, integer, Boolean	'alpha'
'twopoint'	real, integer, Boolean	'alpha'
'uniform'	real, integer, Boolean	'alpha', 'p'

SetCrossProb Call

call SetCrossProb *(p)* **;**

The SetCrossProb call sets the crossover probability. The input to the SetCrossProb subroutine is as follows:

p is the crossover probability.

The SetCrossProb subroutine is used to set the crossover probability for the genetic algorithm optimization process. The crossover probability *p* should be between 0 and 1. Typical values for this parameter range from 0.6 to 1.0. The crossover probability will be overridden if required by a SetElite call. The elite solutions are passed on to the next generation without undergoing crossover, regardless of the crossover probability.

SetCrossRoutine Call

call SetCrossRoutine *('routine'< , nparents, nchildren>)* **;**

The SetCrossRoutine call installs a user subroutine for the crossover operator. The inputs to the SetCrossRoutine subroutine are as follows:

routine	is the name of a subroutine you have defined, which is called when the mutation operator is applied. This parameter must be a string literal; a variable is not accepted.
nparents	is optional, and specifies the number of parent solutions the operator requires. If not specified, 2 is assumed.
nchildren	is optional, and specifies the number of children solutions the operator will generate. If not specified, 2 is assumed.

SetElite Call

call SetElite *(elite)* **;**

The SetElite call sets the number of best solutions to pass to the next generation. The input to the SetElite subroutine is as follows:

elite	is the number of best solutions to be passed unmodified from the current solution generation to the next.

The SetElite subroutine is used to ensure that the best solutions encountered in the optimization are not lost by the random selection process. In pure tournament selection, although better solutions are more likely to be selected, it is also possible that any given solution will not be chosen to participate in a tournament, and even if it is selected, it might be modified by crossover or mutation. The SetElite call modifies the optimization process such that the best *elite* solutions in the current population are exactly preserved and passed on to the next generation. This behavior is observed regardless of the crossover or mutation settings. When a SetElite call is made, the first *elite* solutions in the population retrieved by a GetSolutions call or output to a data set are the fittest, and these *elite* solutions are sorted so that the most fit is first. In general, using the SetElite call speeds the convergence of the optimization process. However, it can also lead to premature convergence before a true global optimum is reached. If no SetElite call is made, a default *elite* value of 1 is used by the GA procedure to make sure that the best solution encountered in the optimization process is never lost.

SetEncoding Call

call SetEncoding *(encoding)* **;**

The SetEncoding call specifies the problem encoding. The input to the SetEncoding subroutine is as follows:

encoding	is a string used to specify the form of the solution.

The SetEncoding subroutine is used to establish the type of problem solution encoding. The *encoding* parameter should be a string of letter-number pairs, where the letter determines the type of encoding: I for integer, R for real-valued, S for sequences, and B for Boolean values. Each letter is followed by a number to indicate the number of components for that encoding. Multiple letter-number pairs can be used to

specify a multisegment encoding. For example, the following call specifies that solutions be in the form of a 10-member integer vector:

```
call SetEncoding('I10');
```

The following call specifies that solutions have a 5-component integer segment and a 10-component real-valued segment:

```
call SetEncoding('I5R10');
```

See the section "Using Multisegment Encoding" on page 115 for details about using multisegment encoding.

SetFinalize Call

call SetFinalize *('routine')* **;**

The SetFinalize call designates a user subroutine to perform post processing at the end of the optimization process. The input to the SetFinalize subroutine is as follows:

routine is the name of a subroutine you have defined, which is called when the optimization process ends. This parameter must be a string literal; a variable is not accepted.

The SetFinalize subroutine enables you to define a subroutine to be called at the end of the optimization process. You might use this subroutine to perform additional refinements of the best solution, or you could generate and write out additional data for plots or reports.

SetMut Call

call SetMut *type< , seg>< , pname, pvalue>< , pname, pvalue>. . .)* **;**

The SetMut call sets the mutation operator. The inputs to the SetMut subroutine are as follows:

type is the name of the mutation operator to be applied.

seg is optional, and specifies a segment of the solution to which the operator should be applied. If *seg* is not specified, then it defaults to a value of 1. *seg* needs to be specified only if multisegment encoding is used.

pname is optional, and specifies the name of a particular property to be set for the mutation operator.

pvalue specifies the value to be assigned to the corresponding property name.

The SetMut routine is used to assign a standard mutation operator. For multisegment encoding, the operator can be assigned to a particular solution segment with the *seg* parameter; otherwise a default segment of 1 is assumed. You can set different mutation operators for different segments with multiple SetMut calls.

When a mutation event occurs, all the operators will applied to the same solution. If more than one SetMut call is made for the same segment, then the last call nullifies any previous call for that segment. Also, a SetMutRoutine call nullifies all previous SetMut calls, and a SetMut call nullifies a previous SetMutRoutine call. Properties for the chosen mutation operator can be set with optional *pname-pvalue* pairs. It is also possible to set or reset operator properties with a SetProperty call.

The accepted values for *type* and the corresponding properties are summarized in Table 4.6. See the section "Mutation Operators" on page 122 for a full description of the available operators.

Table 4.6 Mutation Operator Types and Properties

type	encodings	properties
'delta'	real, integer	'delta' 'nchange'
'invert'	sequence	
'null'	all encodings	
'swap'	sequence	'nswap'
'uniform'	real, integer, Boolean	'nchange' 'pchange'

SetMutProb Call

call SetMutProb *(p)* **;**

The SetMutProb call sets the mutation probability. The input to the SetMutProb subroutine is as follows:

p is the mutation probability.

The SetMutProb subroutine is used to set the mutation probability for the genetic algorithm optimization. The probability *p* should be a number between 0 and 1, and is interpreted as the probability that a solution in the next generation should have the mutation operator applied to it. If a SetElite call has been made, then the elite solutions do not undergo mutation. Generally, a high mutation probability degrades the convergence of the genetic algorithm optimization, but some level of mutation is required to assure a thorough search and avoid premature convergence before the global optimum is found. Typical values for *p* are near 0.05 or less.

SetMutRoutine Call

call SetMutRoutine *('routine')* **;**

The SetMutRoutine call installs a user subroutine for the mutation operator. The input to the SetMutRoutine subroutine is as follows:

routine is the name of a subroutine you have defined, which is called when the mutation operator is applied. This parameter must be a string literal; a variable is not accepted.

The SetMutRoutine call enables you to designate a subroutine you have defined to be used for the mutation operator. Your subroutine will be called whenever the mutation operation is performed. See the section "Defining User Genetic Operators" on page 126 for more information about defining a mutation operator.

SetObj Call

call SetObj *(type, minmax, < , seg >< , pname, pvalue >< , pname, pvalue >. . .)* **;**

The SetObj call sets the objective function. The inputs to the SetObj subroutine are as follows:

type is the name of the objective function to be used.

minmax is an indicator to maximize or minimize the objective. A value of 0 is used to specify a minimization, and a value of 1 to specify maximizing the objective.

seg is optional, and specifies a segment of the solution to which the objective function should be applied. If *seg* is not specified, then it defaults to a value of 1. *seg* needs to be specified only if multisegment encoding is used.

pname is optional, and specifies the name of a particular property to be set for the objective function.

pvalue specifies the value to be assigned to the corresponding property name.

The SetObj routine is used to assign a procedure-supplied objective function. For multisegment encoding, the objective can be assigned to a particular solution segment with the *seg* parameter; otherwise a default segment of 1 is assumed. If more than one SetObj call is made, then the last call nullifies any previous call. Also, a SetObjFunc call nullifies all previous SetObj calls, and a SetObj call nullifies a previous SetObjFunc call. Properties for the chosen objective can be set with optional *pname-pvalue* pairs. It is also possible to set or reset objective properties with a SetProperty call.

The accepted values for *type* and the corresponding properties are summarized in Table 4.7. See the section "Objective Functions" on page 124 for a full description of the available objectives.

Table 4.7 Objective Function Types and Properties

type	encodings	properties
'TSP'	real, integer	'distances'

SetObjFunc Call

call SetObjFunc *('fname', minmax)* **;**

The SetObjFunc call sets the objective to a user-defined function. The inputs to the SetObjFunc subroutine are as follows:

fname is the name of a user objective function. This parameter must be a literal string.

minmax is set to 0 to minimize the objective, 1 to maximize.

The SetObjFunc subroutine is used to designate a user function to be the objective for the optimization process. The SetObjFunc call accepts a literal string only for the function name; you cannot use a variable or expression. See the section "Defining an Objective Function" on page 130 for more information about

defining your own objective function. If multiple SetObjFunc calls are made, only the last one is in effect; the last call nullifies any previous SetObjFunc or SetObj calls.

SetProperty Call

call SetProperty *(optype < , seg >, pname, pvalue < , pname, pvalue >. . .)* **;**

The SetProperty call modifies properties of genetic operators, objective functions, and selectors. The inputs to the SetProperty subroutine are as follows:

optype	is the type of operator. It should have a value of 'cross' for a crossover operator, 'mut' for a mutation operator, 'obj' for an objective function, or 'sel' for a selector.
seg	is optional, used only for mutation and crossover operators, and specifies the segment in which the operator resides. It is necessary only for multisegment encoding. The default value if *seg* is not specified is 1.
pname	specifies the name of a particular property to be set.
pvalue	specifies the value to be assigned to the corresponding property name. Multiple property name-value pairs can be supplied in a SetProperty call.

The SetProperty call is used to set or modify properties of a genetic operator, objective function, or a selector. It can be called anytime during the optimization process to dynamically adapt optimization parameters. For example, you might call SetProperty from a user update routine to reduce the magnitude of the *delta* vector of a *delta* mutation operator as the optimization progresses to an optimum.

SetSel Call

call SetSel *(selector < , pname, pvalue >< , pname, pvalue >. . .)* **;**

The SetSel call sets the selection parameters. The inputs to the SetSel subroutine are as follows:

selector	is the type of selection strategy to be used.
pname	is optional, and specifies the name of a particular property to be set for the selector operator.
pvalue	specifies the value to be assigned to the corresponding property name.

The SetSel call is used to specify a selector for the regeneration process, which selects members of the current generation to be propagated to the next. Generally, selection is based on solution fitness, with the fittest solutions more likely to be selected.

The supported values for *selector* and the corresponding selector properties and their default values are summarized in Table 4.8. See the section "Specifying the Selection Strategy" on page 132 for a full description of the available selectors and their properties.

Table 4.8 Selectors and Properties

Selector	Properties	Default values
'tournament'	'size'	2
'duel'	'pbest'	0.8

SetUpdateRoutine Call

call SetUpdateRoutine *('routine')* **;**

The SetUpdateRoutine call designates a control subroutine to be called at each iteration. The input to the SetUpdateRoutine subroutine is as follows:

routine is the name of a subroutine you have defined that is called once during each iteration of the optimization process. This parameter must be a string literal; a variable is not accepted.

The SetUpdate subroutine enables you to define a subroutine to be called at each iteration of the optimization process, in order to monitor the progress of the genetic algorithm, adjust optimization parameters, or perform calculations that depend on the population as a whole. The specified routine is called once at each iteration, just before the selection process and after the evaluation phase. See the section "Defining a User Update Routine" on page 128 for a discussion of how an update routine might be used.

ShellSort Call

call ShellSort *(x, < , by<, descend> >)* **;**

The ShellSort call sorts a numeric array. The inputs to the ShellSort subroutine are as follows:

x is a one or two dimensional array to be sorted.

by is an optional numeric scalar or array that specifies the columns by which the array is to be sorted. If not specified, column 1 is the default.

descend is an optional numeric scalar or array used to specify which columns in the *by* parameter are to be in descending order. Any columns not specified in a *descend* parameter will be in ascending order.

The ShellSort subroutine sorts the *x* array by the columns specified in the *by* parameter, with the first column having highest precedence, and subsequent columns applied within the preceding by groups. Sorting will be done in ascending order for each column unless that column is also specified in the *descend* parameter. In general the ShellSort routine does not preserve the original order in case of ties. For example, the following statements sort an array *x* into ascending order with respect to the first column, and sort groups with the same first column value by descending order of third column values:

```
array x[100, 3] /nosym;

...
```

```
array by[2] /nosym;
by[1] = 1;
by[2] = 3;
descend = 3;
call ShellSort(x, by, descend);
```

Shuffle Call

call Shuffle *(x)* **;**

The Shuffle call randomly reorders a numeric array. The input to the Shuffle subroutine is as follows:

x	is a numeric array to be randomly shuffled.

The Shuffle subroutine randomly rearranges the elements of the *x* array. One example of where it might be used is in a user-supplied initialization routine, to generate a random sequence-encoded solution segment.

UnpackBits Function

r = UnpackBits *(source, start, width)* **;**

The UnpackBits call retrieves bit values from a packed integer array. The inputs to the UnpackBits function are as follows:

source	is an array containing the packed bit values.
start	is the starting bit, with the lowest bit starting at 1.
width	is the number of bits to retrieve. A value of 1 retrieves a single bit.

The UnpackBits function facilitates the extraction of bit values from arbitrary locations in an integer array. One common use for it is to retrieve bit values from an integer solution segment in a user objective or genetic operator routine. The return value, *start*, and *width* parameters are consistent with the PackBits call, which you can use to store bit values to an integer array.

The *start* parameter is the lowest desired bit position in the bit vector, corresponding to the least significant bit of the return value. The *start* parameter can range in value from 1 to *maxbits*, where *maxbits* is the product of 32 times the number of elements in the integer array.

The *width* parameter is the number of bits to be read into the return value from the array. It is bounded by $0 < width < (maxbits - start + 1)$, and must also not exceed 32.

UpdateSolutions Call

call UpdateSolutions *(sol, n, seg)* **;**

The UpdateSolutions call updates the current solution population. The inputs to the UpdateSolutions subroutine are as follows:

sol is an array containing the replacement solution elements.

n is the number of solutions to update.

seg is the segment of the solution to replace.

The UpdateSolutions subroutine is used to replace the values of the selected solution segment with new values computed in an update routine. The update routine can be designated in a SetUpdateRoutine call. The UpdateSolutions call is often used to implement advanced strategies such as marking Pareto-optimal sets or employing local optimizations. The *sol* parameter should have 2 dimensions. The first dimension represents the solution number, and should have a value of *n* or greater. The second dimension represents the element within the solution *seg*, and should be equal to the segment size.

WriteChild Call

call WriteChild *(selected, seg, n, source)* **;**

The WriteChild call assigns values to a selected child solution from within a user crossover operator. The inputs to the WriteChild subroutine are as follows:

selected is an array specifying the selected family of solutions. The *selected* array is normally passed into the user subroutine that calls WriteChild, and should be passed unaltered to WriteChild.

seg is the segment to which the elements are to be written.

n is the child within the family to which the elements are to be written. A value of 1 is for the first child, 2 for the second, and so on.

source is an array containing the values to be written.

The WriteChild subroutine is called inside a user crossover operator subroutine to assign to the elements of a selected child solution. It is normally used to complete the action of the crossover operator.

WriteMember Call

call WriteMember *(selected, seg, source)* **;**

The WriteMember call assigns values to a selected solution from within a user objective function or mutation operator. The inputs to the WriteMember subroutine are as follows:

selected is an array specifying the selected family of solutions. The *selected* array is normally passed into the user subroutine that calls WriteMember, and should be passed unaltered to WriteMember.

seg is the segment to which the elements are to be written.

source is an array containing the values to be written.

The WriteMember subroutine is called inside a user objective function or mutation operator subroutine to assign values to the elements of a selected solution. It is normally used to complete the action of the objective function or mutation operator.

Details: GA Procedure

Using Multisegment Encoding

The GA procedure enables you to represent problems with solutions consisting of mixed parameter types by using multisegment encoding. Solutions can contain multiple segments, where each segment is a vector of one particular parameter type. Multiple segments can also be used to store additional information you want to keep for each solution for user objective functions or genetic operators. The utility functions provided by the GA procedure give you full access to read from and write to individual solution segments you define.

Segments are set up with a SetEncoding call. The input parameter to this call is a string consisting of letter-number pairs, with each pair describing the type and number of elements in one segment. The permitted letters and corresponding encodings are as follows:

R or *r* specifies real encoding. The elements of the solution segment are real numbers. One common problem where this encoding is used is nonlinear function optimization over the real domain.

I or *i* specifies integer encoding. The elements of the solution segment are integers. Examples of where this encoding might be used include assignment problems where the integers represent which resources are assigned to particular tasks, or problems involving real variables that are constrained to be integers.

B or *b* specifies Boolean encoding. The elements of the solution consist of binary (0 or 1) bits. This type of encoding might be used, for example, to represent variables in a variable selection problem, or inclusion of particular items in a 0/1 knapsack problem.

S or *s* specifies sequence encoding. The segment consists of randomly ordered sequences of integers ranging from 1 to the number of elements. For example, [2, 4, 5, 1, 3] is an example of S5 encoding, as is [5, 3, 2, 1, 4]. Sequence encoding is a natural way to represent routing optimizations like the traveling salesman problem, or any problem optimizing permutations of parameters.

Suppose the problem is to optimize the scheduling of 20 tasks, and for each task you need to choose one machine out of a set of appropriate machines for each task. The natural encoding for that problem could be set up with the following call:

```
call SetEncoding('I20S20');
```

This call specifies a two-segment solution encoding, with segment 1 (I20) an integer vector representing the machine assignment for each task, and segment 2 (S20) representing the sequence of tasks.

When you use multisegment encoding, you must specify the segment parameter in SetMut or SetCross calls to specify an operator in a segment other than the first one. If you code your own operator subroutines, you can use utility functions provided by the GA procedure to extract and write out values to individual segments of the solution, and routines provided by the GA procedure to perform standard genetic crossover and mutation operations on selected segments. See the section "Using Standard Genetic Operators and Objective Functions" on page 116 for a discussion of the operators provided by the GA procedure. See the section "Defining User Genetic Operators" on page 126 and the section "Defining an Objective Function" on page 130 for details of defining user routines.

Using Standard Genetic Operators and Objective Functions

The GA procedure includes a set of objective functions and crossover and mutation operators, and also enables you to define your own with a subroutine. The standard operators and objectives that are provided for each encoding type are summarized in Table 4.9.

Table 4.9 Standard Genetic Operators for Each Encoding

Encoding	Crossover	Mutation	Objective
real	arithmetic heuristic null simple twopoint uniform	delta null uniform	
integer	arithmetic null simple twopoint uniform	delta null uniform	
Boolean	simple null twopoint uniform	null uniform	
sequence	cycle null order pmatch	invert null swap	TSP

The following sections describe the standard genetic operators and objectives and how to invoke them.

Crossover Operators

Arithmetic

This operator is defined for real and integer encoding. It treats the solution segment as a vector, and computes offspring of parents P and Q as

$$child1 = aP + (1-a)Q$$

$$child2 = aQ + (1-a)P$$

where a is a random number between 0 and 1 generated by the GA procedure. For integer encoding, each component is rounded to the nearest integer. It has the advantage that it always produces feasible offspring for a convex solution space. A disadvantage of this operator is that it tends to produce offspring toward the interior of the search region, so it might not work if the optimum lies on or near the search region boundary. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCross( 'Arithmetic');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Arithmetic', segment);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, segment, 'Arithmetic');
```

where *selected* is the selection parameter passed to your subroutine, and *segment* is the segment to which the arithmetic crossover operator is to be applied.

Cycle

This operator is defined for sequence encoding. It produces offspring such that the position of each element value in the offspring comes from one of the parents. For example, consider parents P and Q,

$$P = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

$$Q = [8, 7, 9, 3, 4, 1, 2, 5, 6]$$

For the first child, pick the first element from the first parent:

$$child1 = [1, ., ., ., ., ., ., ., .]$$

To maintain the condition that the position of each element value must come from one of the parents, the position of the '8' value must come from P, because the '8' position in Q is already taken by the '1' in *child1*:

$$child1 = [1, ., ., ., ., ., ., 8, .]$$

Now the position of '5' must come from P, and so on until the process returns to the first position:

$$child1 = [1, ., 3, 4, 5, 6, ., 8, 9]$$

At this point, choose the remaining element positions from Q:

$$child1 = [1, 7, 3, 4, 5, 6, 2, 8, 9]$$

For the second child, starting with the first element from the second parent, similar logic produces

$$child2 = [8, 2, 9, 3, 4, 1, 7, 5, 6]$$

This operator is most useful when the absolute position of the elements is of most importance to the objective value. For single-segment encoding, you can specify this operator with the call

```
call SetCross( 'Cycle');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Cycle', segment);
```

From within a user crossover subroutine, you can use the use the call

```
call Cross(selected, segment, 'Cycle');
```

where *selected* is the selection parameter passed to your subroutine, and *segment* is the segment to which the cycle crossover operator is to be applied.

Heuristic

This operator is defined for real encoding. It treats the solution segments as real vectors. It computes the first offspring from two parents P and Q, where Q is the parent with the best objective value, as

$$child1 = a(Q - P) + Q$$

$$child2 = aQ + (1 - a)P$$

where a is a random number between 0 and 1 generated by the GA procedure. The first child is a projection, and the second child is a convex combination, as with the arithmetic operator. This operator is unusual in that it uses the objective value. It has the advantage of directing the search in a promising direction, and automatically fine-tuning the search in an area where solutions are clustered. If the solution space has upper and lower bound constraints, the offspring are checked against the bounds, and any component outside its bound is set equal to that bound. The heuristic operator performs best when the objective function is smooth, and might not work well if the objective function or its first derivative is discontinuous.

For single-segment encoding, you can specify this operator with the call

```
call SetCross( 'Heuristic');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Heuristic', segment);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, segment, 'Heuristic');
```

where *selected* is the selection parameter passed to your subroutine, and *segment* is the segment to which the heuristic crossover operator is to be applied.

Null

This operator is used to specify that no crossover operator be applied. It is not usually necessary to specify the null operator, except when you want to cancel a previous operator selection. You can specify the null operator with the call

```
call SetCross( 'null');
```

For multisegment encoding, you can specify a segment to which the operator should be applied with the call

```
call SetCross( 'null', segment);
```

Order

This operator is defined for sequence encoding. It produces offspring by transferring a randomly chosen subsequence of random length and position from one parent, and filling the remaining positions according to the order from the other parent. For parents P and Q, first choose two random cutpoints to define a subsequence:

$$P = [1, 2, |3, 4, 5, 6, |7, 8, 9]$$

$$Q = [8, 7, |9, 3, 4, 1, |2, 5, 6]$$

$$child1 = [., ., 3, 4, 5, 6, ., ., .]$$

$$child2 = [., ., 9, 3, 4, 1, ., ., .]$$

Starting at the second cutpoint and cycling back to the beginning, the elements of Q in order are as follows:

```
2 5 6 8 7 9 3 4 1
```

After removing 3, 4, 5 and 6, which have already been placed in *child1*, you have the following entries:

```
2 8 7 9 1
```

Placing these back in order starting at the second cutpoint yields the following sequence:

$$child1 = [9, 1, 3, 4, 5, 6, 2, 8, 7]$$

Applying this logic to *child2* yields the following sequence:

$$child2 = [5, 6, 9, 3, 4, 1, 7, 8, 2]$$

This operator maintains the similarity of the relative order, or adjacency, of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the traveling salesman problem. For single-segment encoding, you can specify this operator with the call

```
call SetCross( 'Order');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Order', segment);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, segment, 'Order');
```

where *selected* is the selection parameter passed to your subroutine, and *segment* is the segment to which the order crossover operator is to be applied.

Pmatch

The partial match operator is defined for sequence encoding. It produces offspring by transferring a subsequence from one parent, and filling the remaining positions in a way consistent with the position and ordering in the other parent. Start with two parents and randomly chosen cutpoints as indicated:

$$P = [1, 2, |3, 4, 5, 6, |7, 8, 9]$$

$$Q = [8, 7, |9, 3, 4, 1, |2, 5, 6]$$

The first step is to cross the selected subsegments as follows (note that '.' indicates positions yet to be determined):

$$child1 = [., ., 9, 3, 4, 1, ., ., .]$$

$$child2 = [., ., 3, 4, 5, 6, ., ., .]$$

Next, define a mapping according to the two selected subsegments:

```
9-3, 3-4, 4-5, 1-6
```

Then, fill in the positions where there is no conflict from the corresponding parent, as follows:

$$child1 = [., 2, 9, 3, 4, 1, 7, 8, .]$$

$$child2 = [8, 7, 3, 4, 5, 6, 2, ., .]$$

Last, fill in the remaining positions from the subsequence mapping. In this case, for the first child, $1 \rightarrow 6$ and $9 \rightarrow 3$, $3 \rightarrow 4$, $4 \rightarrow 5$, and for the second child, $5 \rightarrow 4$, $4 \rightarrow 3$, $3 \rightarrow 9$, and $6 \rightarrow 1$.

$$child1 = [6, 2, 9, 3, 4, 1, 7, 8, 5]$$

$$child2 = [8, 7, 3, 4, 5, 6, 2, 9, 1]$$

This operator tends to maintain similarity of both the absolute position and relative ordering of the sequence elements, it and is useful for a wide range of sequencing problems. For single-segment encoding, you can specify this operator with the call

```
call SetCross( 'Pmatch');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Pmatch', segment);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, segment, 'Pmatch');
```

where *selected* is the selection parameter passed to your subroutine, and *segment* is the segment to which the Pmatch crossover operator is to be applied.

Simple

This operator is defined for integer, real, and Boolean encoding. It has one property, $alpha$. This operator performs the following action: a position k within an encoding of length n is chosen at random, such that $1 \leq k < n$. Then for parents P and Q, the offspring are as follows:

$$child1 = [P_1, P_2, \ldots, P_k, Q_{k+1}, Q_{k+2}, \ldots, Q_n]$$

$$child2 = [Q_1, Q_2, \ldots, Q_k, P_{k+1}, P_{k+2}, \ldots, P_n]$$

For integer and real encoding, you can specify an additional $alpha$ property of value a, where $0 < a \leq 1$. It modifies the offspring as follows:

$$p_i = aP_i + (1-a)Q_i,\ i = k+1, k+2, \ldots, n$$

$$q_i = aQ_i + (1-a)P_i,\ i = k+1, k+2, \ldots, n$$

$$child1 = [P_1, P_2, \ldots, P_k, q_{k+1}, q_{k+2}, \ldots, q_n]$$

$$child2 = [Q_1, Q_2, \ldots, Q_k, p_{k+1}, p_{k+2}, \ldots, p_n]$$

For integer encoding, the elements are then rounded to the nearest integer. For Boolean encoding, the a parameter is ignored, and is effectively 1.

For single-segment encoding, you can specify this operator with the call

```
call SetCross( 'Simple', 'alpha', a);
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Simple', segment, 'alpha', a);
```

From within a user crossover subroutine, you can use

```
call Cross(selected, segment, 'Simple', a);
```

where *selected* is the selection passed to your subroutine, and *segment* is the segment to which the simple crossover operator is to be applied.

Twopoint

This operator is defined for integer, real, and Boolean encoding of length $n \geq 3$, and has one property, $alpha$. Two positions, $k1$ and $k2$, are chosen at random, such that $1 \leq k1 < k2 < n$. Element values between those positions are swapped between parents. For parents Q and P, the offspring are as follows:

$$child1 = [P_1, P_2, \ldots, P_{k1}, Q_{k1+1}, \ldots, Q_{k2}, P_{k2+1}, \ldots, P_n]$$

$$child2 = [Q_1, Q_2, \ldots, Q_{k1}, P_{k1+1}, \ldots, P_{k2}, Q_{k2+1}, \ldots, Q_n]$$

For integer and real encoding, you can specify an additional $alpha$ property of value a, where $0 < a \leq 1$. It modifies the offspring as follows:

$$p_i = aP_i + (1-a)Q_i,\ i = k1+1, k1+2, \ldots, k2$$

$$q_i = aQ_i + (1-a)P_i,\ i = k1+1, k1+2, \ldots, k2$$

$$child1 = [P_1, P_2, \ldots, P_{k1}, q_{k1+1}, \ldots, q_{k2}, P_{k2+1}, \ldots, P_n]$$

$$child2 = [Q_1, Q_2, \ldots, Q_{k1}, p_{k1+1}, \ldots, p_{k2}, Q_{k2+1}, \ldots, Q_n]$$

Note that small values of a reduce the difference between the offspring and parents. For Boolean encoding, a is always 1. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCross( 'Twopoint', 'alpha', a);
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross(selected, segment, 'Twopoint', 'alpha', a);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, segment, 'Twopoint', a);
```

where *selected* is the selection passed to your subroutine, and *seg* is the segment to which the two-point crossover operator is to be applied.

Uniform

This operator is defined for integer, real, and Boolean encoding of length $n \geq 3$, and has two properties, $alpha$ and p, where $0 < alpha \leq 1$ and $0 < p \leq 0.5$. For $alpha = a$ and parents S and T, offspring s and t are generated such that

$$s_i = \begin{cases} aT_i + (1-a)S_i, & \text{with probability } p \\ S_i, & \text{otherwise} \end{cases}$$

$$t_i = \begin{cases} aS_i + (1-a)T_i, & \text{with probability } p \\ T_i, & \text{otherwise} \end{cases}$$

Note that *alpha* and p determine how much interchange there is between parents. Lower values of *alpha* and p imply less change between offspring and parents. For Boolean encoding, *alpha* is always assigned to be 1. If you do not specify *alpha*, it defaults to a value of 1. If you do not specify p, it defaults to a value of 0.5.

For single-segment encoding, you can specify the use of this operator with the call

```
call SetCross( 'Uniform', 'alpha', a, 'p', p);
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetCross( 'Uniform', segment, 'alpha', a, 'p', p);
```

From within a user crossover subroutine, you can use the call

```
call Cross(selected, seg, 'Uniform', a, p);
```

where *selected* is the selection parameter passed to your crossover subroutine, and *seg* is the segment to which the uniform crossover operator is to be applied. In both cases, if the encoding is Boolean, the a parameter is ignored, and a is effectively 1.

Mutation Operators

Delta

This operator is defined for integer and real encoding. It has two properties, $delta$ and $nchange$. It first chooses n elements of the solution at random, where n is the value of the $nchange$ property, and then perturbs each chosen element by a fixed amount, set by d, the value of the $delta$ property. d must be an array with the same length as the encoding. A randomly chosen element k of the solution S is modified such that

$$S_k \in \{S_k - d_k, S_k + d_k\}$$

If upper and lower bounds are specified with a SetBounds call, then S_k is adjusted as necessary to fit within the bounds. This operator gives you the ability to fine-tune the search by modifying the magnitude of the $delta$ property. One possible strategy is to start with larger d values, and then reduce them as the search progresses and begins to converge to an optimum. This operator is also useful if the optimum is known to be on or near a boundary, in which case d can be set large enough to always perturb the solution element to a boundary. For single-segment encoding, you can specify this operator with the call

```
call SetMut( 'delta', 'nchange', n, 'delta', d);
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetMut( 'delta', segment, 'nchange', n, 'delta', d);
```

You can invoke this operator on a solution segment from within a user mutation subroutine with the call

```
call Mutate( selected, segment, 'delta', d, n);
```

where $selected$ is the selection parameter passed into the subroutine.

Invert

This operator is defined for sequence encoding. It picks two locations at random and reverses the order of elements between them. This operator is most often applied to the traveling salesman problem. For single-segment encoding, you can specify this operator with the call

```
call SetMut( 'invert');
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetMut( 'invert', segment);
```

You can invoke this operator on a solution segment from within a user mutation subroutine with the call

```
call Mutate( selected, segment, 'invert');
```

where $selected$ is the selection parameter passed into the subroutine.

Null

This operator is used to specify that no mutation operator be applied. It is not usually necessary to specify the null operator, except when you want to cancel a previous operator selection. You can specify the null operator with the call

```
call SetMut( 'null');
```

For multisegment encoding, you can specify a segment to which the operator should be applied with

```
call SetMut( 'null', segment);
```

Swap

This operator is defined for sequence problem encoding. It picks two random locations in the solution vector and swaps their values. You can also specify that multiple swaps be made for each mutation, by setting the $nswap$ property. For single-segment encoding, you can specify the swap operator with the call

```
call SetMut( 'swap', 'nswap', n);
```

where n is the number of swaps for each mutation. For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetMut( 'swap', segment, 'nswap', n);
```

You can invoke this operator on a solution segment from within a user mutation subroutine with the call

```
call Mutate( selected, segment, 'swap', n);
```

where $selected$ is the selection parameter passed into the subroutine.

Uniform

This operator is defined for Boolean encoding or for real or integer encoding with upper and lower bounds specified with a SetBounds call. To apply this operator, a position k is randomly chosen within the solution S, and S_k is modified to a random value between the upper and lower bounds for element k. There are two properties you can specify, $nchange$ and $pchange$. If you set the $nchange$ property to n, the operator is applied to n locations. If you set the $pchange$ property to a value p, where $0 < p < 1$, the operator is applied at each position with probability p. Only the last property setting is active, overriding any previous

settings. If no property is set, a default *nchange* value of 1 is used. You can specify this operator with one of the following two calls:

```
call SetMut( 'uniform', 'nchange', n);
call SetMut( 'uniform', 'pchange', p);
```

For multisegment encoding, you can specify the segment to which the operator should be applied with the call

```
call SetMut( 'uniform', segment, 'nchange', n);
```

You can invoke this operator on a solution segment from within a user mutation subroutine by using one of the following statements, where *selected* is the selection parameter passed into the subroutine:

```
call Mutate( selected, segment, 'uniform', n);
call Mutate( selected, segment, 'uniform', p);
```

This operator can prove especially useful in early stages of the optimization, since it tends to distribute solutions widely across the search space, and to avoid premature convergence to a local optimum. However, in later stages of an optimization, when the search needs to be fine-tuned to home in on an optimum, the uniform operator can hinder the optimization.

Objective Functions

TSP

The TSP objective calculates the value of the traveling salesman problem objective. It has one property, *distances*, which is a two-dimensional array representing distances between locations. If *distances* = *d*, then *d*[*i*,*j*] is the distance between location *i* and location *j*.

Defining a User Fitness Comparison Routine

Most of the selection techniques used in this procedure require comparisons of fitness to decide which solutions should be propagated into the next generation. Normally fitness is directly related to the solution objective value, with the better objective value indicating the higher fitness. However, sometimes it is useful to base fitness on multiple criteria, not just a single calculated objective value. Doing so enables you to solve multiobjective optimization problems, or to find optima that also satisfy secondary objectives. For example, for a multiobjective problem you might want to find a set of construction schedules that simultaneously minimize completion time and minimize cost, so that you can examine the tradeoff between the two objectives and choose a schedule that best meets your needs. You might also have a scheduling problem where there is more than one solution that meets your primary objective, but you would like to converge on the one that also provides adequate worker breaks, or minimizes overtime. One single-valued objective function might not be adequate to express these preferences.

You can define a function and designate that function to be used for comparing the fitness of two competing solutions with the call

```
call SetCompareRoutine( 'routine');
```

where *routine* is the name of the function you have defined. The function name must be a quoted string. The first parameter of the function, designated the *selection* parameter, must be a numeric array. The procedure

calls your compare function whenever it needs to compare the fitness of two solutions. Your function can also have an arbitrary number of additional parameters, whose names and data types should match variables defined in the global portion of your input. When the procedure calls your function, the *selection* parameter will be filled in with information identifying which solutions are to be compared, and any other parameters will be filled in with the corresponding global symbol values. Your function should not alter the selection parameter in any way, but pass it unchanged into a ReadCompare Call to obtain the solution elements your program needs to compare the fitness. If solution 1 has higher fitness than solution 2, then your function should return a positive number. If solution 2 is more fit than solution 1, then a negative number should be returned. If the two solutions have the same fitness then a value of 0 might be returned. Following is a simple example of a fitness comparison function.

```
/* This function is designed to maximize a
 * primary and secondary objective. The objectives
 * are stored in segment 2 of the encoding. If
 * the difference in primary objective is less than
 * the value of delta, then the secondary objective
 * is used to determine the fitness
 */
function compare2(selected[*], delta);

/* arrays to hold solution elements */
array member1[2] /nosym;
array member2[2] /nosym;

/* read segment 2 of solution 1 into member1 */
call ReadCompare(selected, 2, 1, member1);

/* read segment 2 of solution 2 into member2 */
call ReadCompare(selected, 2, 2, member2);

/* element 1 contains the primary objective value, */
/* element 2 contains a secondary objective value  */

/* if objective 1 is nearly the same, then use */
/* objective 2 to compare the fitness          */
if( abs(member1[1] - member2[1]) < delta) then do;
  /* primary objectives are nearly the same, check secondary */
  if(member1[2] > member2[2]) then
    return(1);
  if(member2[2] > member1[2]) then
    return(-1);
end;

/* base fitness on primary objective */
if(member1[1] > member2[1]) then
  return(1);
if(member2[1] > member1[1]) then
  return(-1);

/* all objectives are the same, return 0 */
return(0);
endsub;
```

```
/* in global scope of the input */
delta = 0.01;
call SetCompareRoutine('compare2');
```

For an example of a comparison routine used in a multiobjective optimization, see Example 4.3.

Defining User Genetic Operators

You can define new genetic operators with subroutines. The GA procedure calls your subroutine when it is necessary to perform mutation or crossover operations. You can designate that a subroutine be used for crossover with the call

```
call SetCrossRoutine('routine');
```

where *routine* is the name of the subroutine you have defined. Similarly, you can designate a subroutine for the mutation operator with the call

```
call SetMutRoutine('routine');
```

The subroutine name must be a quoted string. The first parameter of the crossover or mutation subroutine you define must be a numeric array. When the GA procedure calls your subroutine, it passes information in the first parameter, referred to as the *selection* parameter, which designates the selected members for the operation. You should not alter the selection parameter in any way, but pass it unchanged into special utility routines provided by the GA procedure in order to obtain the solution elements and write them to the selected members. You can define as many other parameters to your subroutine as you need; they are filled in with values from variables of the same name created in the global portion of your program. Any array parameters must be numeric and of the type /NOSYMBOLS.

For a crossover subroutine, use the ReadParent call to get the elements of the selected parents into arrays that you can then manipulate with programming statements. The results can be written to the designated offspring with a WriteChild call. The following code is an example of a crossover subroutine. The subroutine creates two new offspring from two selected parents by switching the odd-numbered elements between the two parents.

```
/* single-segment integer encoding of size 10 */
call SetEncoding('I10');

/* encoding size is 10 */
n = 10;

subroutine swapodd(selected[*], n);
  array child1[1] /nosym;
  array child2[1] /nosym;

  /* reallocate child arrays to right size */
  call dynamic_array(child1,n);
  call dynamic_array(child2,n);

  /* read segment 1 from parent 1 into child1 */
  call ReadParent(selected, 1, 1, child1);
```

```
  /* read segment 1 from parent 2 into child2 */
  call ReadParent(selected, 1, 2, child2);

  /* swap the odd elements in the solution */
  do i = 1 to n by 2;
    temp = child1[i];
    child1[i] = child2[i];
    child2[i] = temp;
  end;

  /* write offspring out to selected children */
  call WriteChild(selected, 1, 1, child1);
  call WriteChild(selected, 1, 2, child2);
endsub;

/* designate swapodd as the crossover routine */
call SetCrossRoutine('swapodd');
```

The next sample program illustrates a crossover routine that might be used for multisegment mixed integer and sequence encoding. The subroutine uses the standard Simple crossover operator for the integer segment, and the Pmatch operator for the sequence-encoded segment.

```
/* Solution has 2 segments, integer I5 and sequence S5 */
call SetEncoding('I5S5');

/* alpha parameter for Simple crossover operator */
alpha = 1;

subroutine mixedIS(selected[*], alpha);

  /* execute simple operator on segment 1 */
  call Cross(selected, 1, 'Simple', alpha);

  /* execute pmatch operator on segment 2 */
  call Cross(selected, 2, 'Pmatch');

endsub;

call SetCrossRoutine('mixedIS');
```

For a mutation subroutine, use a ReadMember call to obtain the elements of the solution selected to be mutated, and use a WriteMember call to write the mutated elements back to the solution. For example, the following statements define a mutation subroutine that swaps two adjacent elements at a randomly chosen position in a sequence:

```
/* Solution has 1 segment, sequence S10 */
call SetEncoding('S10');

n = 10;

subroutine swap2(selected[*], n);
```

```
  /* declare an array for working memory */
  array member[1] /nosym;

  /* allocate array to required length */
  call dynamic_array(member, n);

  /* read segment 1 of selected member into array */
  call ReadMember(selected,1,member);

  /* generate random number between 0 and 1 */
  r = rand('uniform');

  /* convert r to integer between 1 and n-1 */
  i = int(r * (n - 1)) + 1;

  /* swap element values */
  temp = member[i];
  member[i] = member[i+1];
  member[i+1] = temp;

  /* write result back out to solution */
  call WriteMember(selected,1,member);

endsub;

/* Set the mutation routine to swap2 */
call SetMutRoutine('swap2');
```

Defining a User Update Routine

The GA procedure enables you to define a routine that is to be called at each generation in the optimization process, after the designated objective function has been evaluated for each solution, before the selection process takes place. Some of the tasks that can be handled with an update routine include evaluating global solution fitness criteria, logging intermediate optimization progress, evaluating termination criteria and stopping the optimization, modifying the properties of the genetic operators to intensify or diversify the search process, or even to reinitialize portions of the solution population. You can designate a user update function with the call

```
call SetUpdateRoutine('name');
```

where *name* is the name of the routine you have defined.

The parameters defined for the update routine must correspond to variables of the same name established in the global portion of your procedure input. If you desire to update a parameter and have that update retained across generations, that parameter must also be declared in an OUTARGS statement in the update routine definition. The following program snippet illustrates how a user might specify an update routine that monitors the best objective value at each generation and terminates the optimization when no improvement is seen after a specified number of iterations.

```
subroutine no_improvement_terminator(iteration,
                                     saved_value,
                                     nsame,
                                     same_limit,
                                     populationSize);
outargs iteration, saved_value, nsame;

array objValues[1] /nosym;

/* dynamically allocate array to fit populationSize */
call dynamic_array(objValues, populationSize);

/* read in current objective values */
call GetObjValues(objValues, populationSize);

/* find best value */
current_best = objValues[1];
do i = 2 to populationSize;
  /* for a minimization problem, use < here */
  if(objValues[i] > current_best) then
     current_best = objValues[i];
end;

if iteration > 0 then do;
  /* for a minimization problem, use < here */
  if current_best > saved_value then do;
    /* reset same value counter */
    nsame = 1;
    saved_value = current_best;
  end;
  else do;
    /* increment same value counter */
    nsame = nsame + 1;
    /* if counter equals limit, then make this the last generation */
    if nsame >= same_limit then
      call ContinueFor(0);
  end;
end;
else do;
  saved_value = current_best;
  nsame = 1;
end;

iteration = iteration + 1;

endsub;

iteration = 0;
saved_value = 0;
nsame = 0;
/* terminate when no improvement after 30 generations */
same_limit = 30;
populationSize = 100;
call SetUpdateRoutine('no_improvement_terminator');
```

From within a user update routine you can use a GetObjValues call to get the current solution objective values, a GetSolutions call to get the current solution population, an UpdateSolutions call to reset solution values, a ReEvaluate call to recompute objective values, or an Initialize call to reinitialize and resize the solution population.

Defining an Objective Function

The GA procedure enables you to specify your objective to be optimized with a function you create, or as a standard objective function that the GA procedure provides. Currently the only standard objective you can specify without writing an objective function is the traveling salesman problem, which can be specified with a SetObj call. In the future, other objective functions will be added. You can designate a user objective function with the call

```
call SetObjFunc('name', minmax);
```

where *name* is the name of the function you have defined, and *minmax* is set to 0 to specify a minimum or 1 to specify a maximum.

A user objective function must have a numeric array as its first parameter. When the GA procedure calls your function, it passes an array in the first parameter that specifies the selected solution, which is referred to as the *selection* parameter. The selection parameter must not be altered in any way by your function. Your function should pass the selection parameter to a ReadMember call to read the elements of the selected solution into an array. Your function can then access this array to compute an objective value, which it must return. As with the genetic operator routines, you can define additional arguments to your objective function, and the GA procedure passes in variables with corresponding names that you have created in your global program. For example, the following statements set up an objective function that minimizes the sum of the squares of the solution elements:

```
call SetEncoding('R5');

n = 5;

function sumsq(selected[*], n);

  /* set up a scratch array to hold solution elements */
  array x[1] /nosym;

  /* allocate x to hold all solution elements */
  call dynamic_array(x, n);

  /* read members of the selected solution into x */
  call ReadMember(selected, 1, x);

  /* compute the sum of the squares */
  sum = 0;
  do i = 1 to n;
    sq = x[i] * x[i];
    sum = sum + sq;
  end;
```

```
  /* return the objective value */
  return(sum);
endsub;

call SetObjFunc('sumsq', 0);
```

In this example, the function SUMSQ is defined, and the SetObjFunc call establishes it as the objective function. The 0 for the second parameter of the SetObjFunc call indicates that the objective should be minimized. Note that the second parameter to the SUMSQ function, *n*, is defined in the procedure, and the value assigned to it there is passed into the function.

Defining a User Initialization Routine

For problems with simple constant bounds or simple sequencing problems it is not necessary to define a user initialization subroutine; simply specify 'DEFAULT' in the Initialize call. Defining a routine is necessary only if you need to satisfy more complicated constraints or apply some initial heuristics or local optimizations. A user initialization routine is specified with an Initialize call, as follows:

```
call Initialize('name', size);
```

where *name* is the name of your initialize routine. The first parameter of the subroutine you define must be a numeric array. When the GA procedure calls your subroutine, it passes information in the first parameter, referred to as the *selection* parameter, which designates the member selected for initialization. Your subroutine should generate one solution and write the values of the solution elements with a WriteMember call, using the selection parameter passed to your subroutine. The random number functions from BASE SAS are available to your subroutine, if needed. You can define as many other parameters to your subroutine as you need; they are filled in with values from variables of the same name created in your global program. The array used to write the generated solution to the population must be numeric and declared with the /NOSYMBOLS option, as well as any arrays passed as parameters into your subroutine.

The following sample statements illustrate how to define an initialization routine. The feasible region is a triangle with vertices (0,0), (0,1) and (1,1).

```
call SetEncoding('R2');

/* set vertices of triangle (0,0), (0,1), and (1,1) */
array vertex1[2] /nosym (0,0);
array vertex2[2] /nosym (0,1);
array vertex3[2] /nosym (1,1);

subroutine triangle(selected[*], vertex1[2], vertex2[2], vertex3[2]);

array x[2] /nosym;

/* select 3 random numbers 0 < r < 1 */
r1 = rand('uniform');
r2 = rand('uniform');
r3 = rand('uniform');

/* normalize so r1 + r2 + r3 = 1 */
sumr = r1 + r2 + r3;
```

```
r1 = r1 / sumr;
r2 = r2 / sumr;
r3 = r3 / sumr;

/* form a convex combination of vertices in x */
do i = 1 to 2;
  x[i] = r1 * vertex1[i] + r2 * vertex2[i] + r3 * vertex3[i];
end;

/* write x out to the selected population member, to segment 1 */
call WriteMember(selected, 1, x);
endsub;

[other programming statements]

call Initialize('triangle',100);
```

In this example, the triangle initialization subroutine generates a solution that is a random convex combination of three points, which places it in the interior of the triangular region defined by the points. Note the use of the BASE SAS `RAND()` function to get random numbers uniformly distributed between 0 and 1. The random numbers are then normalized so that their sum is 1. In the loop, they are used to compute a convex linear combination of the vertices, and the WriteMember call writes the solution to the selected population member. The encoding specified a single segment, so the WriteMember call specifies segment 1 as the target. When the GA procedure executes the Initialize call, it executes the triangle routine 100 times, once for each member of the initial population.

Specifying the Selection Strategy

There are a number of different strategies that can employed to select the most fit solutions from a population to be propagated into the next solution generation. Regardless of the strategy chosen, the user must try set the selection properties to maintain a productive balance between selection pressure and preservation of diversity. Enough selective pressure must be maintained to drive the algorithm toward an optimum in a reasonable time, but too much selective pressure will eliminate the diversity of the population too quickly and lead to premature convergence to a suboptimal solution. One effective technique is to start the optimization process at a lower selective pressure, and increase it as the optimization continues. You can easily do this in the GA procedure by modifying the selection strategy with a SetProperty call inside an update routine. A description of the selection strategies and their properties follows.

Tournament Selector

This selector has one property, *size*. The selector repeats the following process until the next generation of solutions has been specified: randomly choose a small subset of the current population, and then select the most fit from that subset. The selection pressure is controlled by the size of the subset chosen, specified by the *size* property. Tournament sizes from 2 to 10 have been successfully applied to various genetic algorithm optimizations, with sizes over 4 or 5 considered to represent strong selective pressure. If the *size* property is not set by the user, it defaults to a value of 2. The Duel selector is a modification of this selector that permits further lowering of selection pressure.

With this selector the objective value is normally used to compare the fitness of competing solutions, with better objective values corresponding to higher fitness. However, there are techniques for multiple objective optimization and constraint handling that use more complicated criteria than just the objective value. See the section "Optimizing Multiple Objectives" on page 135 for further discussion of this topic. You can set your own fitness comparison routine to implement those techniques with the SetCompareRoutine call, and that routine will be used by the tournament selector to compare solution fitness. For a simple single-objective problem you normally do not need to specify a special fitness comparison routine, unless have a secondary objective you might want to use to break a tie.

Duel Selector

This selector has one property, $bprob$. It operates in the same manner as the Tournament selector with tournament size 2, except it permits you to reduce selection pressure further by specifying a probability, $bprob$, for selecting the most fit solution from the pair. The $bprob$ property is assigned a default value of 0.8, but you can change it to a value between 0.5 (corresponding to pure random selection) and 1. By default, solution fitness is compared by comparing the solution objective values. However, you can also set your own fitness comparison routine with the SetCompareRoutine call. See the tournament selector for a discussion of when you would need to specify a special fitness comparison routine.

Incorporating Heuristics and Local Optimizations

It is often effective to combine the genetic algorithm technique and other local optimizations or heuristic improvements. This can be done within the GA procedure by incorporating a local optimization into a user objective function and returning an improved objective value. Either your user objective function can replace the original solution with the optimized one, or you can leave the solution unchanged, replacing it with the optimized one only at the final iteration.

Replacing the original solution with the locally optimized one speeds convergence, but it also increases the risk of converging prematurely. If you choose to do so, you can modify the solution by writing the changed solution back to the population with a WriteMember call. You could also consider replacing the original solution with some probability p. For some problems, values of p from 0.05 to 0.15 have been shown to significantly improve convergence while avoiding premature convergence to a local optimum. This technique is illustrated in Example 4.1.

Handling Constraints

Practical optimization problems usually involve constraints, which can make the problems harder to solve. Constraints are handled in genetic algorithms in several ways.

Encoding Strategy

The simplest approach is to set the problem encoding, genetic operators, and initialization such that the constraints are automatically satisfied. Fixed constant bounds are easily handled in this manner in the GA procedure with the SetBounds call. The default initialization process and genetic operators provided by the GA procedure automatically respect bounds specified in this manner. For some types of constraints, you might be able to create a direct mapping from a constrained solution domain to a second domain with simple

constant bounds. You could then define your genetic operator to map the solution into the second domain, apply one of the standard genetic operators, and then map the result back to the original domain.

If the problem contains equality constraints, you should try to transform the problem to eliminate the equality constraints and reduce the number of variables. This strategy is opposite from what is usually done in linear programming, where inequality constraints are turned into equality constraints by the addition of slack variables.

Repair Strategy

If the constraints are more complex and cannot be easily satisfied automatically by the genetic operators, you might be able to employ a repair strategy: check the solution and modify it to satisfy the constraints. The check and repair can be done in a user genetic operator when the solution is generated, or it can be done in the evaluation phase in a user objective function. Possible strategies for making a repair inside an objective function include projecting the solution onto the constraint boundary; while inside a genetic operator you might try adjusting an operator parameter until the constraint is satisfied. If you do the repair in the objective function, you should compute the objective value after performing the repair. You can write the repaired solution back out to the population with a WriteMember call from a user objective function or mutation subroutine, and with a WriteChild call from within a crossover subroutine. Example 4.2 illustrates the use of the repair strategy.

Penalty Strategy

Another technique is to permit solutions to violate constraints, but also to impose a fitness penalty that causes the population to evolve toward satisfying constraints as well as optimizing the objective. One way of employing this strategy is to simply add a penalty term to the objective function, but this approach should be used with care, because it is not always obvious how to construct the penalty function in a way that does not bias the optimization of the desired objective.

Direct Comparison Strategy

Using tournament selection opens another possibility for handling constraints. Define a fitness comparison routine (designated in a SetCompareRoutine call) that employs the following logic:

1 If neither solution is feasible, choose the one closest to satisfying the constraints.

2 If one solution is feasible, and the other is not, choose the feasible one.

3 If both solutions are feasible, choose the one with the best objective value.

This strategy has the advantage that the objective function does not have to be calculated for infeasible solutions. To implement this method, you need to provide a measure of constraint violation and compute it in a user objective function; this value can be used in the first comparison step outlined previously. For linear constraints, the GA procedure provides the EvaluateLC call for this purpose. The technique works best when the solution space normally contains a significant number of solutions that satisfy the constraints. Otherwise it is possible that a single feasible solution might quickly dominate the population. In such cases, a better approach might be the following Bicriteria Comparison Strategy.

Bicriteria Comparison Strategy

A variation of the direct comparison strategy that has proved effective in many applications is the multiobjective, bicriteria approach. This strategy involves adding a second objective function, which is the magnitude of the constraint violation. Based on the original and constraint violation objective functions, a Pareto-optimal set of solutions is evolved in the population, and the Pareto-optimal set is evolved toward zero constraint violation. This technique is illustrated in Example 4.3. See the section "Optimizing Multiple Objectives" on page 135 for a full discussion of Pareto optimality and how to apply this technique.

Optimizing Multiple Objectives

Many practical optimization problems involve more than one objective criteria, where the decision maker needs to examine trade-offs between conflicting objectives. With traditional optimization methods, these problems are often handled by aggregating multiple objectives into a single scalar objective, usually accomplished by some linear weighting of the multiple criteria. Other approaches involve turning objectives into constraints. One disadvantage of this strategy is that many separate optimizations with different weighting factors or constraints need to be performed to examine the trade-offs between different objectives. Genetic algorithms enable you to attack multiobjective problems directly, in order to evolve a set of solutions in one run of the optimization process instead of solving multiple separate problems.

This approach seeks to evolve the Pareto-optimal set: the set of solutions such that for each solution, all the objective criteria cannot be simultaneously improved. This is expressed mathematically by the concept of Pareto optimality. A Pareto-optimal set is the set of all nondominated solutions, according to the following definition of *dominated*:

For an n-objective minimizing optimization problem, for each objective function f_i, a solution p is *dominated by* q if

$$f_i(p) \geq f_i(q) \text{ for all } i = 1, \ldots, n \text{ and } f_j(p) > f_j(q) \text{ for some } j = 1, \ldots, n$$

The following is one strategy that can be employed in the GA procedure to evolve a set of Pareto-optimal solutions to a multiobjective optimization problem:

user objective function: Define and specify a user objective function in a SetObjFunc call that computes each of the objective criteria and stores the objective values in one single solution segment.

user update routine: Define and specify a user update routine in a SetUpdateRoutine call that examines the entire solution population and marks those in the Pareto-optimal set. This can be done with the MarkPareto call provided by the GA procedure. Also, set the *elite* parameter equal to the number of Pareto-optimal solutions found.

selection criteria: Define a fitness comparison routine that favors the least dominated solutions, and designate it in a SetCompareRoutine call. For selecting between two solutions when neither solution dominates the other, your routine can check a secondary criterion to direct the search to the area of ultimate interest.

The multiple objective values are recorded in one segment to enable the use of the MarkPareto call provided by the GA procedure. Setting the *elite* selection parameter to the size of the Pareto-optimal set, in conjunction with the comparison criteria, guarantees that the Pareto-optimal set in each generation is preserved to the next.

The secondary comparison criterion can be used to ensure that the final Pareto-optimal set is distributed in the area of ultimate interest. For example, for the Bicriteria Constraint Strategy described previously, the actual area of interest is where there is zero constraint violation, which is the second objective. The secondary comparison criterion in that case is to minimize the value of the constraint violation objective. After enough iterations, the population should evolve to the point that the best solution to the bicriteria problem is also the best solution to the original constrained problem, and the other Pareto-optimal solutions can be examined to analyze the sensitivity of the optimum solution to the constraints. For other types of problems, you might need to implement a more complicated secondary comparison criterion to avoid "crowding" of solutions about some arbitrary point, and ensure the evolved Pareto-optimal set is distributed over a range of objective values.

Examples: GA Procedure

Example 4.1: Traveling Salesman Problem with Local Optimization

This example illustrates the use of the GA procedure to solve a traveling salesman problem (TSP); it combines a genetic algorithm with a local optimization strategy. The procedure finds the shortest tour of 20 locations randomly oriented on a two-dimensional x-y plane, where $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The location coordinates are input in the following DATA step:

```
/* 20 random locations for a  Traveling Salesman Problem */
data locations;
   input x y;
   datalines;
0.0333692 0.9925079
0.6020896 0.0168807
0.1532083 0.7020444
0.3181124 0.1469288
0.1878440 0.8679120
0.9786112 0.4925364
0.7918010 0.7943144
0.5145329 0.0363478
0.5500754 0.8324617
0.3893757 0.6635483
0.9641841 0.6400201
0.7718126 0.5463923
0.7549037 0.4584584
0.2837881 0.7733415
0.3308411 0.1974851
0.7977221 0.1193149
0.3221207 0.7930478
0.9201035 0.1186234
0.2397964 0.1448552
0.3967470 0.6716172
;
```

First, the GA procedure is run with no local optimizations applied:

```
proc ga data1 = locations seed = 5554;
   call SetEncoding('S20');
   ncities = 20;
   array distances[20,20] /nosym;
   do i = 1 to 20;
      do j = 1 to i;
         distances[i,j] = sqrt((x[i] - x[j])**2 + (y[i] - y[j])**2);
         distances[j,i] = distances[i,j];
      end;
   end;
   call SetObj('TSP',0,'distances', distances);
   call SetCross('Order');
   call SetMut('Invert');
   call SetMutProb(0.05);
   call SetCrossProb(0.8);
   call SetElite(1);
   call Initialize('DEFAULT',200);
   call ContinueFor(140);
run;
```

The PROC GA statement uses a DATA1= option to get the contents of the locations data set, which creates array variables x and y from the corresponding fields of the data set. A solution will be represented as a circular tour, modeled as a 20-element sequence of locations, which is set up with the SetEncoding call. The array variable *distances* is created, and the loop initializes *distances* from the x and y location coordinates such that $distances[i, j]$ is the Euclidean distance between location i and j. Next, the SetObj call specifies that the GA procedure use the included TSP objective function with the *distances* array. Then, the genetic operators are specified, with SetCross for the crossover operator and SetMut for the mutation operator. Since the crossover probability and mutation probability are not explicitly set in this example, the default values of 1 and 0.05, respectively, are used. The selection parameters are not explicitly set (with SetSel and SetElite calls), so by default, a tournament of size 2 is used, and an *elite* parameter of 1 is used. Next, the Initialize call specifies default initialization (random sequences) and a population size of 100. The ContinueFor call specifies a run of 220 iterations. This value is a result of experimentation, after it was determined that the solution did not improve with more iterations. The output of this run of PROC GA is given in Output 4.1.1.

Output 4.1.1 Simple Traveling Salesman Problem

PROC GA Optimum Values

Objective
3.7465311323

Output 4.1.1 *continued*

Solution	
Element	Value
1	12
2	13
3	18
4	16
5	2
6	8
7	15
8	4
9	19
10	3
11	1
12	5
13	14
14	17
15	10
16	20
17	9
18	7
19	11
20	6

The following program illustrates how the problem can be solved in fewer iterations by employing a local optimization. Inside the user objective function, before computing the objective value, every adjacent pair of cities in the tour is checked to determine if reversing the pair order would improve the objective value. For a pair of locations S_i and S_{i+1}, this means comparing the distance traversed by the subsequence $\{S_{i-1}, S_i, S_{i+1}, S_{i+2}\}$ to the distance traversed by the subsequence $\{S_{i-1}, S_{i+1}, S_i, S_{i+2}\}$, with appropriate wrap-around at the endpoints of the sequence. If the distance for the swapped pair is smaller than the original pair, then the reversal is done, and the improved solution is written back to the population.

```
proc ga data1 = locations seed = 5554;
   call SetEncoding('S20');
   ncities = 20;
   array distances[20,20] /nosym;
   do i = 1 to 20;
      do j = 1 to i;
         distances[i,j] = sqrt((x[i] - x[j])**2 + (y[i] - y[j])**2);
         distances[j,i] = distances[i,j];
      end;
   end;
   /* Objective function with local optimization */
   function TSPSwap(selected[*],ncities,distances[*,*]);
   array s[1] /nosym;
   call dynamic_array(s,ncities);
   call ReadMember(selected,1,s);
```

```
   /* First try to improve solution by swapping adjacent cities */
   do i = 1 to ncities;
      city1 = s[i];
      inext = 1 + mod(i,ncities);
      city2 = s[inext];
      if i=1 then
         before = s[ncities];
      else
         before = s[i-1];
         after = s[1 + mod(inext,ncities)];
      if (distances[before,city1]+distances[city2,after]) >
         (distances[before,city2]+distances[city1,after]) then do;
         s[i] = city2;
         s[inext] = city1;
      end;
   end;
   call WriteMember(selected,1,s);
   /* Now compute distance of tour */
   distance = distances[s[ncities],s[1]];
   do i = 1 to (ncities - 1);
      distance + distances[s[i],s[i+1]];
   end;
   return(distance);
   endsub;
   call SetObjFunc('TSPSwap',0);
   call SetCross('Order');
   call SetMut('Invert');
   call SetMutProb(0.05);
   call SetCrossProb(0.8);
   call SetElite(1);
   call Initialize('DEFAULT',200);
   call ContinueFor(35);
run;
```

The output after 85 iterations is given in Output 4.1.2.

Output 4.1.2 Traveling Salesman Problem with Local Optimization

PROC GA Optimum Values

Objective
3.7465311323

Output 4.1.2 *continued*

Solution	
Element	Value
1	13
2	18
3	16
4	2
5	8
6	15
7	4
8	19
9	3
10	1
11	5
12	14
13	17
14	10
15	20
16	9
17	7
18	11
19	6
20	12

Since all tours are circular, the actual starting point does not matter, and this solution is equivalent to that reached with the simple approach without local optimization. It is reached after only 85 iterations, versus 220 with the simple approach.

NOTE: For an example of how to use PROC OPTNET to solve the TSP, see the "Examples" section in Chapter 2, "The OPTNET Procedure" (*SAS/OR User's Guide: Network Optimization Algorithms*).

Example 4.2: Nonlinear Objective with Constraints Using Repair Mechanism

This example illustrates the use of a repair mechanism to satisfy problem constraints. The problem is to minimize the six-hump camel-back function (Michalewicz 1996, Appendix B):

$$f(x) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1 x_2 + \left(-4 + 4x_2^2\right) x_2^2$$

where x is within the triangle with vertices $V_1 = (-2, 0)$, $V_2 = (0, 2)$, and $V_3 = (2, -2)$. The problem formulation takes advantage of the fact that all feasible solutions can be expressed as a convex combination of V_1, V_2, and V_3 in the following form:

$$x = aV_1 + bV_2 + cV_3$$

In this form, a, b, and c are nonnegative coefficients and satisfy the following linear equality constraint:

$$a + b + c = 1$$

Therefore, the solution encoding 'R3' is used with the three elements corresponding to the values of a, b, and c. Note that this strategy can be generalized to any solution domain that can be specified by a convex hull. An additional 'R2' segment is also created to store the corresponding x value. In the following program, **FUNCTION SIXHUMP** computes the objective value. Lower and upper bounds of 0 and 1, respectively, are used to ensure that solution elements are nonnegative. Violations of the linear equality constraint are fixed by a simple repair strategy, implemented in **FUNCTION SIXHUMP**: the sum of the solution elements is computed, and each element is divided by the sum, so that the sum of the new elements is 1. For the special case of all elements equal to 0, equal values are assigned to the solution elements.

```
proc ga seed = 555;
call SetEncoding('R3R2');
npoints = 3;
array cvxhull[3,2] /nosym ( -2 0
                            0 2
                            2 -2 );
/* Objective function */
function sixhump(selected[*],cvxhull[*,*],npoints);
   /* Function has global minimum value of -1.0316
    * at x = {-0.0898  0.7126} and
    *    x = { 0.0898 -0.7126}
    */
   array w[1] /nosym;
   call dynamic_array(w,npoints);
   array x[2] /nosym;
   call ReadMember(selected,1,w);
   /* make sure that weights add up to 1 */
   sum = 0;
   do i = 1 to npoints;
      sum + w[i];
   end;
   /* if all weights 0, then reinitialize */
   if sum=0 then do;
      sum = npoints;
      do i = 1 to npoints;
         w[i] = 1;
      end;
   end;
   /* re-normalize weights */
   do i = 1 to npoints;
      w[i] = w[i] / sum;
   end;
   call WriteMember(selected,1,w);
   /* convert weights to x-coordinate form */
   x[1] = 0;
   x[2] = 0;
   do i = 1 to npoints;
      x[1] + w[i] * cvxhull[i,1];
      x[2] + w[i] * cvxhull[i,2];
   end;
   /* write out x coordinates to second segment */
   call WriteMember(selected,2,x);
   /* compute objective value */
   r = (4 - 2.1*x[1]**2 + x[1]**4/3)*x[1]**2 + x[1]*x[2] +
```

```
        (-4 + 4*x[2]**2)*x[2]**2;
   return(r);
endsub;
call SetObjFunc('sixhump',0);
array lower[1] /nosym;
array upper[1] /nosym;
call dynamic_array(lower, npoints);
call dynamic_array(upper, npoints);
do i = 1 to npoints;
   lower[i] = 0;
   upper[i] = 1;
end;
call SetBounds(lower, upper, 1);
array delta[3] /nosym (0.01 0.01 0.01);
call SetMut('delta', 'nchange', 1, 'delta', delta);
call SetMutProb(0.05);
call SetCross('Twopoint', 'alpha', 0.9);
call SetCrossProb(0.8);
call SetSel('tournament', 'size', 2);
call SetElite(3);
call Initialize('DEFAULT', 200);
call ContinueFor(200);
run;
```

Note that this problem uses the standard genetic operators and default initialization, even though they generate solutions that violate the constraints. This is possible because all solutions are passed into the user objective function for evaluation, where they are repaired to fit the constraints. The output is shown in Output 4.2.1.

Output 4.2.1 Nonlinear Objective with Constraints Using Repair Mechanism

PROC GA Optimum Values

Objective
-1.031617314

Solution		
Segment	Element	Value
1	1	0.2439660392
1	2	0.5561415112
1	3	0.1998924497
2	1	-0.088147179
2	2	0.712498123

This objective function has a global minimum at -1.0316, at two different points: $(x_1, x_2) = (-0.0898, 0.7126)$ and $(x_1, x_2) = (0.0898, -0.7126)$. The genetic algorithm can converge to either of these minima, depending on the random number seed set by the SEED= option.

Example 4.3: Quadratic Objective with Linear Constraints, Using Bicriteria Approach

This example (Floudas and Pardalos 1992) illustrates the bicriteria approach to handling constraints. The problem has nine linear constraints and a quadratic objective function.

Minimize

$$f(x) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$$

subject to

$$\begin{aligned}
&2x_1 + 2x_2 + x_{10} + x_{11} \le 10\\
&2x_1 + 2x_3 + x_{10} + x_{12} \le 10\\
&2x_1 + 2x_3 + x_{11} + x_{12} \le 10\\
&-8x_1 + x_{10} \le 0\\
&-8x_2 + x_{11} \le 0\\
&-8x_3 + x_{12} \le 0\\
&-2x_4 - x_5 + x_{10} \le 0\\
&-2x_6 - x_7 + x_{11} \le 0\\
&-2x_8 - x_9 + x_{12} \le 0
\end{aligned}$$

and

$$\begin{aligned}
&0 \le x_i \le 1, && i = 1, 2, \ldots, 9\\
&0 \le x_i \le 100, && i = 10, 11, 12\\
&0 \le x_{13} \le 1
\end{aligned}$$

In this example, the linear constraint coefficients are specified in the SAS data set lincon and passed to the GA procedure with the MATRIX1= option. The upper and lower bounds are specified in the bounds data set specified with a DATA1= option, which creates the array variables upper and lower, matching the variables in the data set.

```
/* Input linear constraint matrix */
data lincon;
   input A1-A13 b;
   datalines;
 2  2  0  0  0  0  0  0  0  1  1  0  0 10
 2  0  2  0  0  0  0  0  0  1  0  1  0 10
 2  0  2  0  0  0  0  0  0  0  1  1  0 10
-8  0  0  0  0  0  0  0  0  1  0  0  0  0
 0 -8  0  0  0  0  0  0  0  0  1  0  0  0
 0  0 -8  0  0  0  0  0  0  0  0  1  0  0
 0  0  0 -2 -1  0  0  0  0  1  0  0  0  0
 0  0  0  0  0 -2 -1  0  0  0  1  0  0  0
 0  0  0  0  0  0  0 -2 -1  0  0  1  0  0
;

/* Input lower and upper bounds */
data bounds;
```

```
   input lower upper;
   datalines;
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  100
0  100
0  100
0  1
;

proc ga lastgen = out matrix1 = lincon seed = 555555
        data1 = bounds novalidate = 3;
```

Note also that the LASTGEN= option is used to designate a data set to store the final solution generation.

In the following statements, the solution encoding is specified, and a user function is defined and designated as the objective function.

```
call SetEncoding('R13R3');
nvar = 13;
ncon = 9;
function quad(selected[*], matrix1[*,*], nvar, ncon);

  array x[1] /nosym;

  array r[3] /nosym;

  array violations[1] /nosym;

  call dynamic_array(x, nvar);

  call dynamic_array(violations, ncon);

  call ReadMember(selected,1,x);

  sum1 = 0;

  do i = 1 to 4;

    sum1 + x[i] - x[i] * x[i];

  end;

  sum2 = 0;

  do i = 5 to 13;

    sum2 + x[i];
```

```
    end;

    obj = 5 * sum1 - sum2;

    call EvaluateLC(matrix1,violations,sumvio,selected,1);

    r[1] = obj;

    r[2] = sumvio;

    call WriteMember(selected,2,r);

    return(obj);

  endsub;

  call SetObjFunc('quad',0);
```

The SetEncoding call specifies two real-valued segments. The first segment, R13, holds the 13 variables, and the second segment, R3, holds the two objective criteria and the marker for Pareto optimality. As described in the section "Defining an Objective Function" on page 130, the first parameter of the objective function is a numeric array that designates which member of the solution population is to be evaluated. When the quad function is called by the GA procedure during the optimization process, the matrix1, nvar, and ncon parameters receive the values of the corresponding global variables; nvar is set to the number of variables, and ncon is set to the number of linear constraints. The function computes the original objective as the first objective criterion, and the magnitude of constraint violation as the second. With the first dynamic_array call, it allocates a working array, x, large enough to hold the number of variables, and a second array, violations, large enough to tabulate each constraint violation. The ReadMember call fills x with the elements of the first segment of the solution, and then the function computes the original objective $f(x)$. The EvaluateLC call is used to compute the linear constraint violation. The objective and sum of the constraint violations are then stored in the array r, and written back to the second segment of the solution with the WriteMember call. Note that the third element of r is not modified, because that element of the segment is used to store the Pareto-optimality mark, which cannot be determined until all the solutions have been evaluated.

Next, a user routine is defined and designated to be an update routine. This routine is called once at each iteration, after all the solutions have been evaluated with the quad function. The following program illustrates this:

```
  subroutine update(popsize);

    /* find pareto-optimal set */

    array minmax[3] /nosym (-1 -1 0);

    array results[1,1] /nosym;

    array scratch[1] /nosym;

    call dynamic_array(scratch, popsize);

    call dynamic_array(results,popsize,3);
```

```
    /* read original and constraint objectives, stored in
     * solution segment 2, into array */
    call GetSolutions(results,popsize,2);
    /* mark the pareto-optimal set */
    call MarkPareto(scratch, npareto,results, minmax);
    /* transfer the results to the solution segment */
    do i = 1 to popsize;
      results[i,3] = scratch[i];
    end;
    /* write updated segment 2 back into solution population */
    call UpdateSolutions(results,popsize,2);
    /* Set Elite parameter to preserve the first 15 pareto-optimal
     * solutions */
    if npareto < 16 then
       call SetElite(npareto);
    else
       call SetElite(15);
  endsub;
  call SetUpdateRoutine('update');
```

This subroutine has one parameter, popsize, defined within the GA procedure, which is expected to be the population size. The working arrays results, scratch, and minmax are declared. The minmax array is to be passed to a MarkPareto call, and is initialized to specify that the first two elements (the original objective and constraint violation) are to be minimized and the third element is not to be considered. The results and scratch arrays are then dynamically allocated to the dimensions required by the population size.

Next, the results array is filled with the second segment of the solution population, with the GetSolutions call. The minmax and results arrays are passed as inputs to the MarkPareto call, which returns the number of Pareto optimal solutions in the npareto variable. The MarkPareto call also sets the elements of the scratch array to 1 if the corresponding solution is Pareto-optimal, and to 0 otherwise. The next loop then records the results in the scratch array in the third column of the results array, effectively marking the Pareto-optimal solutions. The updated solution segments are written back to the population with the UpdateSolutions call.

The final step in the update routine is to set the elite selection parameter to guarantee the survival of at least a minimum of 15 of the fittest (Pareto-optimal) solutions through the selection process.

With the following statements, a routine is defined and designated as a fitness comparison routine with a SetCompareRoutine call. This routine works in combination with the update routine to evolve the solution population toward Pareto optimality and constraint satisfaction.

```
function paretocomp(selected[*]);
  array member1[3] /nosym;
  array member2[3] /nosym;
  call ReadCompare(selected,2,1, member1);
  call ReadCompare(selected,2,2, member2);
  /* if one member is in the pareto-optimal set
   * and the other is not, then it is the
   * most fit */
  if(member1[3] > member2[3]) then
     return(1);
  if(member2[3] > member1[3]) then
     return(-1);
  /* if both are in the pareto-optimal set, then
   * the one with the lowest constraint violation
   * is the most fit */
  if(member1[3] = 1) then do;
     if member1[2] <= member2[2] then
        return(1);
     return( -1);
  end;
  /* if neither is in the pareto-optimal set, then
   * take the one that dominates the other */
  if (member1[2] <= member2[2]) &
     (member1[1] <= member2[1]) then
     return(1);
```

```
   if (member2[2] <= member1[2]) &

      (member2[1] <= member1[1]) then

      return(-1);

   /* if neither dominates, then consider fitness to be

    * the same */

   return( 0);

endsub;

call SetSel('tournament', 'size', 2);

call SetCompareRoutine('paretocomp');
```

The **PARETOCOMP** subroutine is called in the selection process to compare the fitness of two competing solutions. The first parameter, selected, designates the two solutions to be compared.

The ReadCompare calls retrieve the second segments of the two solutions, where the objective criteria are stored, and writes the segments into the member1 and member2 arrays. The logic that follows first checks for the case where only one solution is Pareto optimal, and returns it. If both the solutions are Pareto optimal, then the one with the smallest constraint violation is chosen. If neither solution is Pareto optimal, then the dominant solution is chosen, if one exists. If neither solution is dominant, then no preference is indicated. After the function is defined, it is designated as a fitness comparison routine with the SetCompareRoutine call.

Next, subroutines are defined and designated as user crossover and mutation operators:

```
/* set up crossover parameters */

subroutine Cross1(selected[*], alpha);

   call Cross(selected,1,'twopoint', alpha);

endsub;

call SetCrossRoutine('Cross1',2,2);

alpha = 0.5;

call SetCrossProb(0.8);

/* set up mutation parameters */

subroutine Mut1(selected[*], delta[*]);

   call Mutate(selected,1,'delta',delta,1);

endsub;
```

```
call SetMutRoutine('Mut1');

array delta[13] /nosym (.5 .5 .5 .5 .5 .5 .5 .5 .5 10 10 10 .1);

call SetMutProb(0.05);
```

These routines execute the standard genetic operators *twopoint* for crossover and *delta* for mutation; see the section "Using Standard Genetic Operators and Objective Functions" on page 116 for a description of each. The alpha and delta variables defined in the procedure are passed as parameters to the user operators, and the crossover and mutation probabilities are set with the SetCrossProb and SetMutProb calls.

At this point, the GA procedure is directed to initialize the first population and begin the optimization process:

```
/* Initialize first population */

call SetBounds(lower, upper);

popsize = 100;

call Initialize('DEFAULT',popsize);

call ContinueFor(500);

run;
```

First, the upper and lower bounds are established with values in the lower and upper array variables, which were set up by the DATA1= option in the PROC GA statement. The SetBounds call sets the bounds for the first segment, which is the default if none is specified in the call. The desired population size of 100 is stored in the popsize variable, so it will be passed to the update subroutine as the popsize parameter. The Initialize call specifies the default initialization, which generates values randomly distributed between the lower and upper bounds for the first encoding segment. Since no bounds were specified for the second segment, it is filled with zeros. The ContinueFor call sets the requested number of iterations to 500, and the RUN statement ends the GA procedure input and begins the optimization process. The output of the procedure is shown in Output 4.3.1.

Output 4.3.1 Bicriteria Constraint Handling Example Output

Bicriteria Constraint Handling Example

PROC GA Optimum Values

Objective
-14.99871988

Output 4.3.1 *continued*

Solution		
Segment	Element	Value
1	1	1
1	2	0.9999997423
1	3	0.9999991741
1	4	0.9999997454
1	5	0.9999982195
1	6	1
1	7	0.9999674484
1	8	0.9999961238
1	9	0.9999691145
1	10	2.9999914332
1	11	2.9999103023
1	12	2.9988939259
1	13	1
2	1	-14.99871988
2	2	0
2	3	1

The minimum value of $f(x)$ is -15 at $x^* = (1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1)$.

References

Floudas, C. A., and Pardalos, P. M. (1992). *Recent Advances in Global Optimization*. Princeton, NJ: Princeton University Press.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. New York: Springer-Verlag.

Subject Index

Syntax Index