SAS/OR® 14.3 User's Guide
Local Search Optimization
The OPTLSO Procedure

This document is an individual chapter from *SAS/OR® 14.3 User's Guide: Local Search Optimization*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS/OR® 14.3 User's Guide: Local Search Optimization*. Cary, NC: SAS Institute Inc.

SAS/OR® 14.3 User's Guide: Local Search Optimization

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

September 2017

Chapter 3
The OPTLSO Procedure

Contents

Overview: OPTLSO Procedure **10**
Getting Started: OPTLSO Procedure **11**
Introductory Examples 12
Syntax: OPTLSO Procedure **17**
PROC OPTLSO Statement 17
PERFORMANCE Statement 21
READARRAY Statement 22
Details: OPTLSO Procedure **23**
The FCMP Procedure 23
The Variable Data Set 23
Describing the Objective Function 24
Describing Linear Constraints 29
Describing Nonlinear Constraints 30
The OPTLSO Algorithm 30
Multiobjective Optimization 32
Specifying and Returning Trial Points 34
Function Value Caching 35
Iteration Log 36
Procedure Termination Messages 36
ODS Tables 37
Macro Variable _OROPTLSO_ 38
Examples: OPTLSO Procedure **39**
Example 3.1: Using Dense Format 39
Example 3.2: Using MPS Format 42
Example 3.3: Using QPS Format 45
Example 3.4: Combining MPS and FCMP Function Definitions 47
Example 3.5: Linear Constraints and a Nonlinear Objective 50
Example 3.6: Using Nonlinear Constraints 52
Example 3.7: Using External Data Sets 55
Example 3.8: Johnson's Systems of Distributions 61
Example 3.9: Discontinuous Function with a Lookup Table 64
Example 3.10: Multiobjective Optimization 68
References **71**

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 2, "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 55.

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 34.

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 34.

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 29.

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 42. 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 29.

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 30.

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 24.

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 34.

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 45. 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 23.

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 34.

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 34.

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 35. 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 37.

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 2, "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 55 and "Example 3.8: Johnson's Systems of Distributions" on page 61.

READARRAY Statement

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

PROC FCMP (see "The FCMP Procedure" on page 23) 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 55. 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 55.

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 24 and "Describing Nonlinear Constraints" on page 30), 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 *SAS Visual Data Management and Utility 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 22 and the second program in "Example 3.7: Using External Data Sets" on page 55). 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 Global 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:

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

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 35. 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 50:

```
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 50 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 29). 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 47.) 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:

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

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

$$\begin{array}{rrl} & 2x_1 - 4x_3 & \leq 15 \\ -1 \leq & 2x_1 - 4x_3 & \end{array}$$

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 42 and "Example 3.3: Using QPS Format" on page 45. 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 25.

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 52.

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

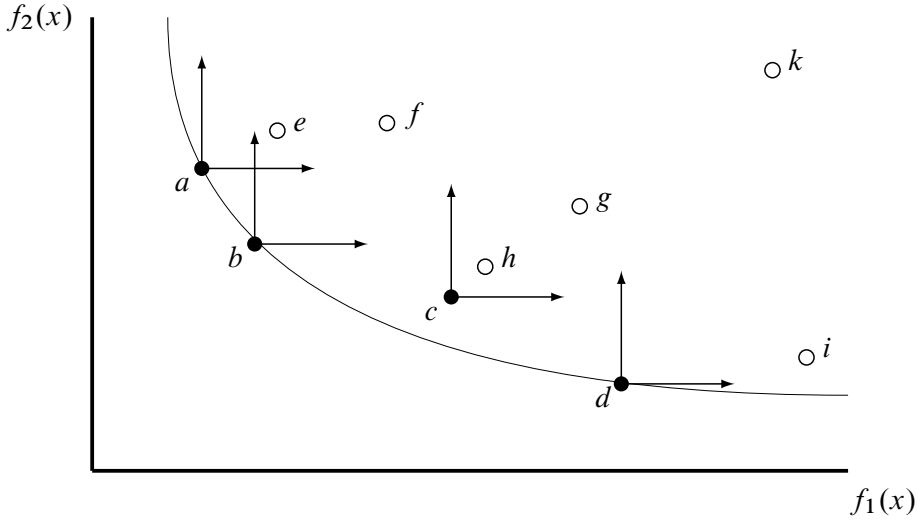
Many practical optimization problems involve more than one objective criterion, so the decision maker needs to examine trade-offs between conflicting objectives. For example, for a particular financial model you might want to maximize profit while minimizing risk, or for a structural model you might 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 68.

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 32. **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 32. **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	-15.00099855
Infeasibility	0.0009992354
Iterations	33
Evaluations	4104
Cached Evaluations	360
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	-15.00099855
Infeasibility	0.0009992354
Iterations	33
Evaluations	4104
Cached Evaluations	360
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	-15.00099855
Infeasibility	0.0009992354
Iterations	33
Evaluations	4104
Cached Evaluations	360
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 42, 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	-15.00099855
Infeasibility	0.0009992354
Iterations	33
Evaluations	4104
Cached Evaluations	360
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	-1.031628451
Infeasibility	0
Iterations	35
Evaluations	2002
Cached Evaluations	15
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	-0.08986
4	0	x2	0.71267
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 42, 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	400.04969132
Infeasibility	0.0009921961
Iterations	73
Evaluations	8435
Cached Evaluations	203
Global Searches	1
Population Size	160
Seed	1

Obs	_sol_	_id_	_value_
1	0	_obj_	400.050
2	0	_inf_	0.001
3	0	x1	0.000
4	0	x2	200.000
5	0	x3	0.000
6	0	x4	99.996
7	0	x5	100.001
8	0	x6	0.000
9	0	x7	99.997
10	0	x8	0.000
11	0	x9	100.002
12	0	x10	1.000
13	0	f	400.050
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
               11 399.924437480041 0.00001536044384     1398        1
               21 400.004990642825    0.00083062502     2669        1
               31 400.018066353994 0.00097200267115     3882        1
               41 400.039224361858 0.00093384733667     5004        1
               51 400.046005452639 0.00095230727416     6076        2
               61 400.048151026458  0.0009381346964     7184        2
               71 400.049691324014 0.00099219605775     8240        2
               73 400.049691324014 0.00099219605775     8435        2
NOTE: Function convergence criteria reached.
NOTE: There were 25 observations read from the data set WORK.NLCEX.
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
   primalout = 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
   primalout = 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	6.976791E-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.

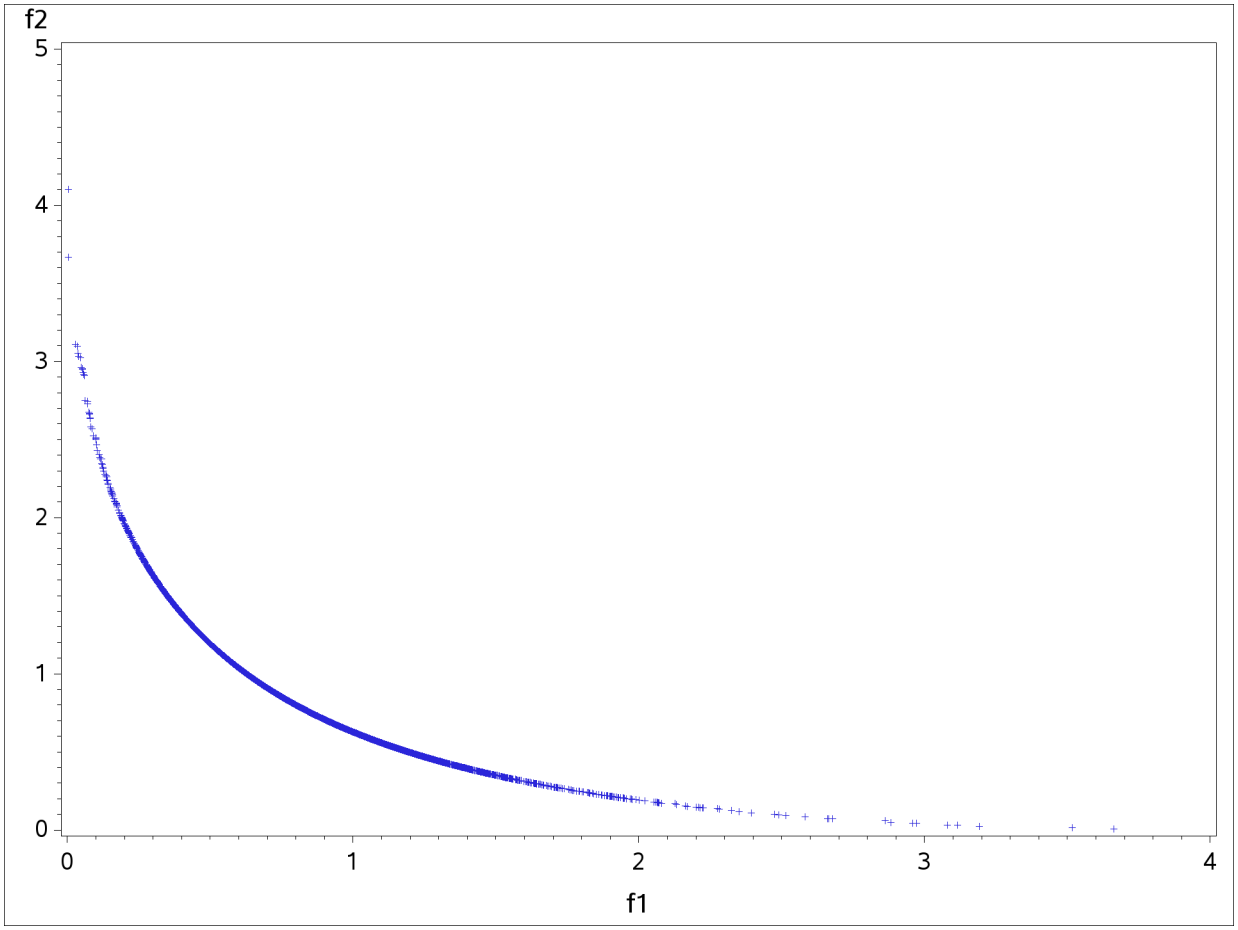
Output 3.10.2 Multiobjective: Log

```
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         4
              251      3429     0.000051118                0    13573         5
              301      3876     0.000001452                0    16163         7
              351      4386     0.000000716                0    18757         8
              401      4796     0.000002728                0    21375        10
              444      5000     0.000000698                0    23647        12
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. http://dx.doi.org/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.

Subject Index

Bard function, 55
Branin function, 12

derivative-free optimization
    OPTLSO procedure, 10
details
    OPTLSO procedure, 23
displayed output to log
    OPTLSO procedure, 23, 36

examples
    OPTLSO procedure, 39

FCMP basics
    OPTLSO procedure, 23
function caching
    OPTLSO procedure, 35
functional summary
    OPTLSO procedure, 17

genetic algorithms, 30

Intermediate functions
    OPTLSO procedure, 25

Johnson function, 61

large data
    OPTLSO procedure, 26
linear constraints
    OPTLSO procedure, 29
local search optimization, 23, 39

OROPTLSO
    OROPTLSO, 38
MPS and QPS objective functions
    OPTLSO procedure, 25
multiple objectives
    OPTLSO procedure, 32

nonlinear constraints
    OPTLSO procedure, 30

objective
    OPTLSO procedure, 24
ODS tables
    OPTLSO procedure, 37
options classified by function, *see* functional summary
OPTLSO examples
    bound-constrained optimization, 12
    introductory examples, 12, 14, 15
    linear constraint, 14
    maximum-likelihood estimates, 15
    nonlinear constraints, 14
OPTLSO procedure
    details, 23
    displayed output to log, 23, 36
    examples, 39
    FCMP basics, 23
    function caching, 35
    functional summary, 17
    Intermediate functions, 25
    large data, 26
    linear constraints, 29
    MPS and QPS objective functions, 25
    multiple objectives, 32
    nonlinear constraints, 30
    objective, 24
    ODS tables, 37
    options classified by function, 17
    overview, 10, 30
    procedure termination messages, 36
    specifying trial points, 34
    table of syntax elements, 17
    time limit, 20
overview
    OPTLSO procedure, 30

procedure termination messages
    OPTLSO procedure, 36

random numbers
    seed, 21

specifying trial points
    OPTLSO procedure, 34
syntax
    OPTLSO procedure, 17

table of syntax elements, *see* functional summary
termination criteria
    time limit, 20

Syntax Index

ABSFCONV= option
    PROC OPTLSO statement, 20

CACHEIN= option
    PROC OPTLSO statement, 18
CACHEMAX= option
    PROC OPTLSO statement, 21
CACHEOUT= option
    PROC OPTLSO statement, 19
CACHETOL= option
    PROC OPTLSO statement, 21

FEASTOL= option
    PROC OPTLSO statement, 20
FIRSTGEN= option
    PROC OPTLSO statement, 18

LASTGEN= option
    PROC OPTLSO statement, 19
LINCON= option
    PROC OPTLSO statement, 18
LOGFREQ= option
    PROC OPTLSO statement, 21
LOGLEVEL= option
    PROC OPTLSO statement, 21

MAXFUNC= option
    PROC OPTLSO statement, 20
MAXGEN= option
    PROC OPTLSO statement, 20
MAXTIME= option
    PROC OPTLSO statement, 20
MPSDATA= option
    PROC OPTLSO statement, 18

NGLOBAL= option
    PROC OPTLSO statement, 20
NLINCON= option
    PROC OPTLSO statement, 19
NLOCAL= option
    PROC OPTLSO statement, 20

OBJECTIVE= option
    PROC OPTLSO statement, 19
OPTLSO procedure
    PROC OPTLSO statement, 17

PARETOMAX= option
    PROC OPTLSO statement, 21
PERFORMANCE statement
    OPTLSO procedure, 21
POPSIZE= option
    PROC OPTLSO statement, 21
PRIMALIN= option
    PROC OPTLSO statement, 19
PRIMALOUT= option
    PROC OPTLSO statement, 20
PRINTLEVEL= option
    PROC OPTLSO statement, 21
PROC OPTLSO statement
    input data set options, 18
    optimization control options, 20
    output data set options, 19
    statement options, 17
    stopping condition options, 20
    technical options, 21

QPSDATA= option
    PROC OPTLSO statement, 19

READARRAY statement
    OPTLSO procedure, 22

SEED= option
    PROC OPTLSO statement, 21

VARIABLES= option
    PROC OPTLSO statement, 19