



THE
POWER
TO KNOW.

SAS/OR[®] 14.1 User's Guide: Mathematical Programming Legacy Procedures The NLP Procedure

This document is an individual chapter from *SAS/OR® 14.1 User's Guide: Mathematical Programming Legacy Procedures*.

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

SAS/OR® 14.1 User's Guide: Mathematical Programming Legacy Procedures

Copyright © 2015, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

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

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

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

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

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

July 2015

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

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

Chapter 6

The NLP Procedure

Contents

Overview: NLP Procedure	538
Getting Started: NLP Procedure	540
Introductory Examples	540
Syntax: NLP Procedure	550
Functional Summary	550
PROC NLP Statement	553
ARRAY Statement	571
BOUNDS Statement	572
BY Statement	572
CRPJAC Statement	573
DECVAR Statement	573
GRADIENT Statement	574
HESSIAN Statement	574
INCLUDE Statement	575
JACNLC Statement	575
JACOBIAN Statement	576
LABEL Statement	577
LINCON Statement	577
MATRIX Statement	578
MIN, MAX, and LSQ Statements	580
MINQUAD and MAXQUAD Statements	580
NLINCON Statement	582
PROFILE Statement	583
Program Statements	584
Details: NLP Procedure	589
Criteria for Optimality	589
Optimization Algorithms	591
Finite-Difference Approximations of Derivatives	601
Hessian and CRP Jacobian Scaling	603
Testing the Gradient Specification	604
Termination Criteria	605
Active Set Methods	606
Feasible Starting Point	608
Line-Search Methods	608
Restricting the Step Length	609
Computational Problems	610

Covariance Matrix	613
Input and Output Data Sets	616
Displayed Output	624
Missing Values	626
Computational Resources	627
Memory Limit	629
Rewriting NLP Models for PROC OPTMODEL	629
Examples: NLP Procedure	637
Example 6.1: Using the DATA= Option	637
Example 6.2: Using the INQUAD= Option	639
Example 6.3: Using the INEST=Option	641
Example 6.4: Restarting an Optimization	643
Example 6.5: Approximate Standard Errors	644
Example 6.6: Maximum Likelihood Weibull Estimation	649
Example 6.7: Simple Pooling Problem	655
Example 6.8: Chemical Equilibrium	664
Example 6.9: Minimize Total Delay in a Network	668
References	673

Overview: NLP Procedure

The NLP (nonlinear programming) procedure offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function $f(x)$ of n decision variables, $x = (x_1, \dots, x_n)^T$ with lower and upper bound, linear and nonlinear, equality and inequality constraints. This can be expressed as solving

$$\begin{aligned} \min_{x \in \mathcal{R}^n} \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0, \quad i = 1, \dots, m_e \\ & c_i(x) \geq 0, \quad i = m_e + 1, \dots, m \\ & l_i \leq x_i \leq u_i, \quad i = 1, \dots, n \end{aligned}$$

where f is the objective function, the c_i 's are the nonlinear functions, and the l_i 's and u_i 's are the lower and upper bounds. Problems of this type are found in many settings ranging from optimal control to maximum likelihood estimation.

The NLP procedure provides a number of algorithms for solving this problem that take advantage of a special structure on the objective function and constraints. One example is the quadratic programming problem,

$$\begin{aligned} \min (\max) \quad & f(x) = \frac{1}{2}x^T Gx + g^T x + b \\ \text{subject to} \quad & c_i(x) = 0, \quad i = 1, \dots, m_e \end{aligned}$$

where G is an $n \times n$ symmetric matrix, $g = (g_1, \dots, g_n)^T$ is a vector, b is a scalar, and the $c_i(x)$'s are linear functions.

Another example is the least squares problem:

$$\begin{aligned} \min \quad & f(x) = \frac{1}{2}\{f_1^2(x) + \dots + f_l^2(x)\} \\ \text{subject to} \quad & c_i(x) = 0, \quad i = 1, \dots, m_e \end{aligned}$$

where the $c_i(x)$'s are linear functions, and $f_1(x), \dots, f_l(x)$ are nonlinear functions of x .

The following problems are handled by PROC NLP:

- quadratic programming with an option for sparse problems
- unconstrained minimization/maximization
- constrained minimization/maximization
- linear complementarity problem

The following optimization techniques are supported in PROC NLP:

- Quadratic Active Set Technique
- Trust Region Method
- Newton-Raphson Method with Line Search
- Newton-Raphson Method with Ridging
- Quasi-Newton Methods
- Double Dogleg Method
- Conjugate Gradient Methods
- Nelder-Mead Simplex Method
- Levenberg-Marquardt Method
- Hybrid Quasi-Newton Methods

These optimization techniques require a continuous objective function f , and all but one (NMSIMP) require continuous first-order derivatives of the objective function f . Some of the techniques also require continuous second-order derivatives. There are three ways to compute derivatives in PROC NLP:

- analytically (using a special derivative compiler), the default method
- via finite-difference approximations
- via user-supplied exact or approximate numerical functions

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

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

- The **DATA=** data set enables you to specify data shared by all functions involved in a least squares problem.
- The **INQUAD=** data set contains the arrays appearing in a quadratic programming problem.
- The **INEST=** data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, and simple boundary and general linear constraints.
- The **MODEL=** data set specifies a model (functions, constraints, derivatives) saved at a previous execution of the NLP procedure.

PROC NLP uses data sets to output various results:

- The **OUTEST=** data set saves the values of the decision variables, the derivatives, the solution, and the covariance matrix at the solution.
- The **OUT=** output data set contains variables generated in the program statements defining the objective function as well as selected variables of the **DATA=** input data set, if available.
- The **OUTMODEL=** data set saves the programming statements. It can be used to input a model in the **MODEL=** input data set.

Getting Started: NLP Procedure

The NLP procedure solves general nonlinear programs. It has several optimizers that are tuned to best perform on a particular class of problems. Guidelines for choosing a particular optimizer for a problem can be found in the section “[Optimization Algorithms](#)” on page 591.

Regardless of the selected optimizer, it is necessary to specify an objective function and constraints that the optimal solution must satisfy. In PROC NLP, the objective function and the constraints are specified using SAS programming statements that are similar to those used in the SAS DATA step. Some of the differences are discussed in the section “[Program Statements](#)” on page 584 and in the section “[ARRAY Statement](#)” on page 571. As with any programming language, there are many different ways to specify the same problem. Some are more economical than others.

Introductory Examples

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

An Unconstrained Problem

Consider the simple example of minimizing the Rosenbrock function (Rosenbrock 1960):

$$\begin{aligned}
 f(x) &= \frac{1}{2} \{100(x_2 - x_1^2)^2 + (1 - x_1)^2\} \\
 &= \frac{1}{2} \{f_1^2(x) + f_2^2(x)\}, \quad x = (x_1, x_2)
 \end{aligned}$$

The minimum function value is $f(x^*) = 0$ at $x^* = (1, 1)$. This problem does not have any constraints.

The following statements can be used to solve this problem:

```
proc nlp;
  min f;
  decvar x1 x2;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
  f = .5 * (f1 * f1 + f2 * f2);
run;
```

The **MIN** statement identifies the symbol f that characterizes the objective function in terms of f_1 and f_2 , and the **DECVAR** statement names the decision variables x_1 and x_2 . Because there is no explicit optimizing algorithm option specified (**TECH=**), PROC NLP uses the Newton-Raphson method with ridging, the default algorithm when there are no constraints.

A better way to solve this problem is to take advantage of the fact that f is a sum of squares of f_1 and f_2 and to treat it as a least squares problem. Using the **LSQ** statement instead of the **MIN** statement tells the procedure that this is a least squares problem, which results in the use of one of the specialized algorithms for solving least squares problems (for example, Levenberg-Marquardt).

```
proc nlp;
  lsq f1 f2;
  decvar x1 x2;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
run;
```

The **LSQ** statement results in the minimization of a function that is the sum of squares of functions that appear in the **LSQ** statement. The least squares specification is preferred because it enables the procedure to exploit the structure in the problem for numerical stability and performance.

PROC NLP displays the iteration history and the solution to this least squares problem as shown in [Figure 6.1](#). It shows that the solution has $x_1 = 1$ and $x_2 = 1$. As expected in an unconstrained problem, the gradient at the solution is very close to 0.

Figure 6.1 Least Squares Minimization

PROC NLP: Least Squares Minimization

Levenberg-Marquardt Optimization

Scaling Update of More (1978)

Parameter Estimates	2
Functions (Observations)	2
Optimization Start	
Active Constraints	0 Objective Function 0.5697839458
Max Abs Gradient Element	20.512989424 Radius 409.78289428

Figure 6.1 continued

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	0	2	0	0.0000927	0.5697	0.2724	0	1.000
2	0	3	0	6.163E-33	0.000093	1.11E-16	0	1.000

Optimization Results			
Iterations	2	Function Calls	4
Jacobian Calls	3	Active Constraints	0
Objective Function	6.162976E-33	Max Abs Gradient Element	1.110223E-16
Lambda	0	Actual Over Pred Change	1
Radius	0.027239421		

ABSGCONV convergence criterion satisfied.

PROC NLP: Least Squares Minimization

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function
1	x1	1.000000	-1.11022E-16
2	x2	1.000000	0

Value of Objective Function = 6.162976E-33

Boundary Constraints on the Decision Variables

Bounds on the decision variables can be used. Suppose, for example, that it is necessary to constrain the decision variables in the previous example to be less than 0.5. That can be done by adding a **BOUNDS** statement.

```
proc nlp;
  lsq f1 f2;
  decvar x1 x2;
  bounds x1-x2 <= .5;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
run;
```

The solution in Figure 6.2 shows that the decision variables meet the constraint bounds.

Figure 6.2 Least Squares with Bounds Solution

PROC NLP: Least Squares Minimization

Value of Objective Function = 18.882523326

PROC NLP: Least Squares Minimization

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function Active Bound Constraint
1	x1	0.500000	-0.500000 Upper BC
2	x2	0.250000	0

Linear Constraints on the Decision Variables

More general linear equality or inequality constraints of the form

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

can be specified in a **LINCON** statement. For example, suppose that in addition to the bounds constraints on the decision variables it is necessary to guarantee that the sum $x_1 + x_2$ is less than or equal to 0.6. That can be achieved by adding a **LINCON** statement:

```
proc nlp;
  lsq f1 f2;
  decvar x1 x2;
  bounds x1-x2 <= .5;
  lincon x1 + x2 <= .6;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
run;
```

The output in Figure 6.3 displays the iteration history and the convergence criterion.

Figure 6.3 Least Squares with Bounds and Linear Constraints Iteration History

PROC NLP: Least Squares Minimization

Value of Objective Function = 29.25

PROC NLP: Least Squares Minimization

Levenberg-Marquardt Optimization

Parameter Estimates	2
Functions (Observations)	2
Lower Bounds	0
Upper Bounds	2
Linear Constraints	1

Optimization Start

Active Constraints	0	Objective Function	29.25
Max Abs Gradient Element	76.5	Radius	1074.0471358

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	0	3	0	8.19877	21.0512	39.5420	0.0170	0.729
2	0	4	0	1.05752	7.1412	13.6170	0.0105	0.885
3	0	5	1	1.04396	0.0136	18.6337	0	0.0128
4	0	6	1	0.16747	0.8765	0.5552	0	0.997
5	0	7	1	0.16658	0.000895	0.000324	0	0.998
6	0	8	1	0.16658	3.06E-10	5.911E-7	0	0.998

Optimization Results

Iterations	6	Function Calls	9
Jacobian Calls	7	Active Constraints	1
Objective Function	0.1665792899	Max Abs Gradient Element	5.9108825E-7
Lambda	0	Actual Over Pred Change	0.9981770421
Radius	0.0000532357		

GCONV convergence criterion satisfied.

Figure 6.4 shows that the solution satisfies the linear constraint. Note that the procedure displays the active constraints (the constraints that are tight) at optimality.

Figure 6.4 Least Squares with Bounds and Linear Constraints Solution

PROC NLP: Least Squares Minimization

Scaling Update of More (1978)

PROC NLP: Least Squares Minimization

Optimization Results			
Parameter Estimates			
			Gradient
N	Parameter	Estimate	Objective
			Function
1	x1	0.423645	-0.312000
2	x2	0.176355	-0.312001

Linear Constraints Evaluated at Solution

1 ACT 2.7756E-17 = 0.6000 - 1.0000 * x1 - 1.0000 * x2

Nonlinear Constraints on the Decision Variables

More general nonlinear equality or inequality constraints can be specified using an **NLINCON** statement. Consider the least squares problem with the additional constraint

$$x_1^2 - 2x_2 \geq 0$$

This constraint is specified by a new function **c1** constrained to be greater than or equal to 0 in the **NLINCON** statement. The function **c1** is defined in the programming statements.

```
proc nlp tech=QUANEW;
  min f;
  decvar x1 x2;
  bounds x1-x2 <= .5;
  lincon x1 + x2 <= .6;
  nlincon c1 >= 0;

  c1 = x1 * x1 - 2 * x2;

  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;

  f = .5 * (f1 * f1 + f2 * f2);
run;
```

Figure 6.5 shows the iteration history, and Figure 6.6 shows the solution to this problem.

Figure 6.5 Least Squares with Bounds, Linear and Nonlinear Constraints, Iteration History

PROC NLP: Nonlinear Minimization									
Dual Quasi-Newton Optimization									
Modified VMCWD Algorithm of Powell (1978, 1982)									
Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)									
Lagrange Multiplier Update of Powell(1982)									
		Parameter Estimates							2
		Lower Bounds							0
		Upper Bounds							2
		Linear Constraints							1
		Nonlinear Constraints							1
Optimization Start									
Objective Function			13.19670008			Maximum Constraint Violation		0	
Maximum Gradient of the Lagran Func			50.53101137						
Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Lagrange Function	Maximum Gradient Element of the	
1	0	4	0.49164	0	0.0125	1.000	0.980		
2	0	5	0.48063	0	0.0427	1.000	1.741		
3	0	6	0.43935	0	0.4840	1.000	1.427		
4	0	8	0.40175	0	0.5017	0.100	0.382		
5	0	10	0.37312	0	0.1381	0.100	0.716		
6	0	12	0.34899	0	0.0431	0.235	0.344		
7	0	13	0.34612	0	0.0351	1.000	1.450		
8	0	14	0.33248	0	0.00441	1.000	0.0896		
9	0	15	0.33017	0	0.000261	1.000	0.117		
10	0	16	0.33004	0	0.000027	1.000	0.00697		
11	0	17	0.33003	0	3.062E-7	1.000	0.00024		
Optimization Results									
Iterations			11			Function Calls		18	
Gradient Calls			14			Active Constraints		0	
Objective Function			0.3300308686			Maximum Constraint Violation		0	
Maximum Projected Gradient			3.0495445201			Value Lagrange Function		0.3300308686	
Maximum Gradient of the Lagran Func			3.0495445201			Slope of Search Direction		-3.061704E-7	

Figure 6.6 Least Squares with Bounds, Linear and Nonlinear Constraints, Solution

PROC NLP: Nonlinear Minimization

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function
1	x1	0.246956	0.753073	0.000013657
2	x2	0.030494	-3.049363	0.000003373

Value of Objective Function = 0.3300307263

Value of Lagrange Function = 0.3300307155

Linear Constraints Evaluated at Solution

1 0.32255 = 0.6000 - 1.0000 * x1 - 1.0000 * x2

Values of Nonlinear Constraints

Constraint	Value	Residual	Lagrange Multiplier
[2] c1_G	7.025E-9	7.025E-9	1.5247 Active NLIC

Not all of the optimization methods support nonlinear constraints. In particular the Levenberg-Marquardt method, the default for LSQ, does not support nonlinear constraints. (For more information about the particular algorithms, see the section “[Optimization Algorithms](#)” on page 591.) The Quasi-Newton method is the prime choice for solving nonlinear programs with nonlinear constraints. The option `TECH=QUANEW` in the `PROC NLP` statement causes the Quasi-Newton method to be used.

A Simple Maximum Likelihood Example

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

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

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

$$\max_{\mu, \sigma > 0} \sum_i l_i(\mu, \sigma)$$

where

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

In the following DATA step, values for x are input into SAS data set X; this data set provides the values of x_i .

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

In the following statements, the `DATA=X` specification drives the building of the objective function. When each observation in the `DATA=X` data set is read, a new term $l_i(\mu, \sigma)$ using the value of x_i is added to the objective function `LOGLIK` specified in the `MAX` statement.

```
proc nlp data=x vardef=n covariance=h pcov phes;
  profile mean sigma / alpha=.5 .1 .05 .01;
  max loglik;
  parms mean=0, sigma=1;
  bounds sigma > 1e-12;
  loglik=-0.5*((x-mean)/sigma)**2-log(sigma);
run;
```

After a few iterations of the default Newton-Raphson optimization algorithm, `PROC NLP` produces the results shown in Figure 6.7.

Figure 6.7 Maximum Likelihood Estimates

PROC NLP: Nonlinear Maximization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	Approx t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	-1.33149E-10
2	sigma	2.000000	0.632456	3.162278	0.025031	5.6064147E-9

Value of Objective Function = -5.965735903

In unconstrained maximization, the gradient (that is, the vector of first derivatives) at the solution must be very close to zero and the Hessian matrix at the solution (that is, the matrix of second derivatives) must have nonpositive eigenvalues. The Hessian matrix is displayed in Figure 6.8.

Figure 6.8 Hessian Matrix

PROC NLP: Nonlinear Maximization

Hessian Matrix		
	mean	sigma
mean	-1.250000003	1.331489E-10
sigma	1.331489E-10	-2.500000014

Determinant = 3.1250000245

Matrix has Only Negative Eigenvalues

Under reasonable assumptions, the approximate standard errors of the estimates are the square roots of the diagonal elements of the covariance matrix of the parameter estimates, which (because of the $COV=H$ specification) is the same as the inverse of the Hessian matrix. The covariance matrix is shown in Figure 6.9.

Figure 6.9 Covariance Matrix

PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	mean	sigma
mean	0.7999999982	4.260766E-11
sigma	4.260766E-11	0.3999999978

Factor sigm = 1

Determinant = 0.3199999975

Matrix has 2 Positive Eigenvalue(s)

The **PROFILE** statement computes the values of the profile likelihood confidence limits on SIGMA and MEAN, as shown in Figure 6.10.

Figure 6.10 Confidence Limits

PROC NLP: Nonlinear Maximization

Wald and PL Confidence Limits							
N	Parameter	Estimate	Alpha	Profile Likelihood Confidence Limits		Wald Confidence Limits	
1	mean	4.000000	0.500000	3.384431	4.615569	3.396718	4.603282
1	mean	.	0.100000	2.305716	5.694284	2.528798	5.471202
1	mean	.	0.050000	1.849538	6.150462	2.246955	5.753045
1	mean	.	0.010000	0.670351	7.329649	1.696108	6.303892
2	sigma	2.000000	0.500000	1.638972	2.516078	1.573415	2.426585
2	sigma	.	0.100000	1.283506	3.748633	0.959703	3.040297
2	sigma	.	0.050000	1.195936	4.358321	0.760410	3.239590
2	sigma	.	0.010000	1.052584	6.064107	0.370903	3.629097

Syntax: NLP Procedure

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

PROC NLP *options* ;
ARRAY *function names* ;
BOUNDS *boundary constraints* ;
BY *variables* ;
CRPJAC *variables* ;
DECVAR *function names* ;
GRADIENT *variables* ;
HESSIAN *variables* ;
INCLUDE *model files* ;
JACNLC *variables* ;
JACOBIAN *function names* ;
LABEL *decision variable labels* ;
LINCON *linear constraints* ;
MATRIX *matrix specification* ;
MIN, MAX, or LSQ *function names* ;
MINQUAD or MAXQUAD *matrix, vector, or number* ;
NLINCON *nonlinear constraints* ;
PROFILE *profile specification* ;
Program Statements ; ;

Functional Summary

The following table outlines the options in `PROC NLP` classified by function.

Table 6.1 Functional Summary

Description	Statement	Option
Input Data Set Options:		
Input data set	PROC NLP	DATA=
Initial values and constraints	PROC NLP	INEST=
Quadratic objective function	PROC NLP	INQUAD=
Program statements	PROC NLP	MODEL=
Skip missing value observations	PROC NLP	NOMISS
Output Data Set Options:		
Variables and derivatives	PROC NLP	OUT=
Result parameter values	PROC NLP	OUTEST=
Program statements	PROC NLP	OUTMODEL=
Combine various OUT... statements	PROC NLP	OUTALL
CRP Jacobian in the OUTEST= data set	PROC NLP	OUTCRPJAC
Derivatives in the OUT= data set	PROC NLP	OUTDER=

Description	Statement	Option
Grid in the OUTEST= data set	PROC NLP	OUTGRID
Hessian in the OUTEST= data set	PROC NLP	OUTHESIAN
Iterative output in the OUTEST= data set	PROC NLP	OUTITER
Jacobian in the OUTEST= data set	PROC NLP	OUTJAC
NLC Jacobian in the OUTEST= data set	PROC NLP	OUTNLCJAC
Time in the OUTEST= data set	PROC NLP	OUTTIME
Optimization Options:		
Minimization method	PROC NLP	TECH=
Update technique	PROC NLP	UPDATE=
Version of optimization technique	PROC NLP	VERSION=
Line-search method	PROC NLP	LINESEARCH=
Line-search precision	PROC NLP	LSPRECISION=
Type of Hessian scaling	PROC NLP	HESCAL=
Start for approximated Hessian	PROC NLP	INHESIAN=
Iteration number for update restart	PROC NLP	RESTART=
Initial Value Options:		
Produce best grid points	PROC NLP	BEST=
Infeasible points in grid search	PROC NLP	INFEASIBLE
Pseudorandom initial values	PROC NLP	RANDOM=
Constant initial values	PROC NLP	INITIAL=
Derivative Options:		
Finite-difference derivatives	PROC NLP	FD=
Finite-difference derivatives	PROC NLP	FDHESIAN=
Compute finite-difference interval	PROC NLP	FDINT=
Use only diagonal of Hessian	PROC NLP	DIAHES
Test gradient specification	PROC NLP	GRADCHECK=
Constraint Options:		
Range for active constraints	PROC NLP	LCEPSILON=
LM tolerance for deactivating	PROC NLP	LCDEACT=
Tolerance for dependent constraints	PROC NLP	LCSINGULAR=
Sum all observations for continuous functions	NLINCON	/ SUMOBS
Evaluate each observation for continuous functions	NLINCON	/ EVERYOBS
Termination Criteria Options:		
Maximum number of function calls	PROC NLP	MAXFUNC=
Maximum number of iterations	PROC NLP	MAXITER=
Minimum number of iterations	PROC NLP	MINITER=
Upper limit on real time	PROC NLP	MAXTIME=
Absolute function convergence criterion	PROC NLP	ABSCONV=
Absolute function convergence criterion	PROC NLP	ABSFCONV=
Absolute gradient convergence criterion	PROC NLP	ABSGCONV=

Description	Statement	Option
Absolute parameter convergence criterion	PROC NLP	ABSXCONV=
Relative function convergence criterion	PROC NLP	FCONV=
Relative function convergence criterion	PROC NLP	FCONV2=
Relative gradient convergence criterion	PROC NLP	GCONV=
Relative gradient convergence criterion	PROC NLP	GCONV2=
Relative parameter convergence criterion	PROC NLP	XCONV=
Used in FCONV, GCONV criterion	PROC NLP	FSIZE=
Used in XCONV criterion	PROC NLP	XSIZE=
Covariance Matrix Options:		
Type of covariance matrix	PROC NLP	COV=
σ^2 factor of COV matrix	PROC NLP	SIGSQ=
Determine factor of COV matrix	PROC NLP	VARDEF=
Absolute singularity for inertia	PROC NLP	ASINGULAR=
Relative M singularity for inertia	PROC NLP	MSINGULAR=
Relative V singularity for inertia	PROC NLP	VSINGULAR=
Threshold for Moore-Penrose inverse	PROC NLP	G4=
Tolerance for singular COV matrix	PROC NLP	COVSING=
Profile confidence limits	PROC NLP	CLPARG=
Printed Output Options:		
Display (almost) all printed output	PROC NLP	PALL
Suppress all printed output	PROC NLP	NOPRINT
Reduce some default output	PROC NLP	PSHORT
Reduce most default output	PROC NLP	PSUMMARY
Display initial values and gradients	PROC NLP	PINIT
Display optimization history	PROC NLP	PHISTORY
Display Jacobian matrix	PROC NLP	PJACOBI
Display crossproduct Jacobian matrix	PROC NLP	PCRPJAC
Display Hessian matrix	PROC NLP	PHESSIAN
Display Jacobian of nonlinear constraints	PROC NLP	PNLCJAC
Display values of grid points	PROC NLP	PGRID
Display values of functions in LSQ, MIN, MAX	PROC NLP	PFUNCTION
Display approximate standard errors	PROC NLP	PSTDERR
Display covariance matrix	PROC NLP	PCOV
Display eigenvalues for covariance matrix	PROC NLP	PEIGVAL
Print code evaluation problems	PROC NLP	PERROR
Print measures of real time	PROC NLP	PTIME
Display model program, variables	PROC NLP	LIST
Display compiled model program	PROC NLP	LISTCODE
Step Length Options:		
Damped steps in line search	PROC NLP	DAMPSTEP=
Maximum trust region radius	PROC NLP	MAXSTEP=
Initial trust region radius	PROC NLP	INSTEP=

Description	Statement	Option
Profile Point and Confidence Interval Options:		
Factor relating discrepancy function to χ^2 quantile	PROFILE	FFACTOR=
Scale for y values written to OUTEST= data set	PROFILE	FORCHI=
Upper bound for confidence limit search	PROFILE	FEASRATIO=
Write all confidence limit parameter estimates to OUTEST= data set	PROFILE	OUTTABLE
Miscellaneous Options:		
Number of accurate digits in objective function	PROC NLP	FDIGITS=
Number of accurate digits in nonlinear constraints	PROC NLP	CDIGITS=
General singularity criterion	PROC NLP	SINGULAR=
Do not compute inertia of matrices	PROC NLP	NOEIGNUM
Check optimality in neighborhood	PROC NLP	OPTCHECK=

PROC NLP Statement

PROC NLP *options* ;

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

ABSCONV=*r*

ABSTOL=*r*

specifies an absolute function convergence criterion. For minimization (maximization), termination requires $f(x^{(k)}) \leq (\geq) r$. The default value of ABSCONV is the negative (positive) square root of the largest double precision value.

ABSFCNV=*r*[*n*]

ABSFTOL=*r*[*n*]

specifies an absolute function convergence criterion. For all techniques except NMSIMP, termination requires a small change of the function value in successive iterations:

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

For the NMSIMP technique the same formula is used, but $x^{(k)}$ is defined as the vertex with the lowest function value, and $x^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is $r = 0$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ABSGCONV=r[n]**ABSGTOL=r[n]**

specifies the absolute gradient convergence criterion. Termination requires the maximum absolute gradient element to be small:

$$\max_j |g_j(x^{(k)})| \leq r$$

This criterion is not used by the NMSIMP technique. The default value is $r=1E-5$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ABSXCONV=r[n]**ABSXTOL=r[n]**

specifies the absolute parameter convergence criterion. For all techniques except NMSIMP, termination requires a small Euclidean distance between successive parameter vectors:

$$\|x^{(k)} - x^{(k-1)}\|_2 \leq r$$

For the NMSIMP technique, termination requires either a small length $\alpha^{(k)}$ of the vertices of a restart simplex

$$\alpha^{(k)} \leq r$$

or a small simplex size

$$\delta^{(k)} \leq r$$

where the simplex size $\delta^{(k)}$ is defined as the L_1 distance of the simplex vertex $y^{(k)}$ with the smallest function value to the other n simplex points $x_l^{(k)} \neq y^{(k)}$:

$$\delta^{(k)} = \sum_{x_l \neq y} \|x_l^{(k)} - y^{(k)}\|_1$$

The default value is $r=1E-4$ for the COBYLA NMSIMP technique, $r=1E-8$ for the standard NMSIMP technique, and $r=0$ otherwise. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

ASINGULAR=r**ASING=r**

specifies an absolute singularity criterion for measuring singularity of Hessian and crossproduct Jacobian and their projected forms, which may have to be converted to compute the covariance matrix. The default is the square root of the smallest positive double precision value. For more information, see the section “Covariance Matrix” on page 613.

BEST=i

produces the i best grid points only. This option not only restricts the output, it also can significantly reduce the computation time needed for sorting the grid point information.

CDIGITS=r

specifies the number of accurate digits in nonlinear constraint evaluations. Fractional values such as $CDIGITS=4.7$ are allowed. The default value is $r = -\log_{10}(\epsilon)$, where ϵ is the machine precision. The value of r is used to compute the interval length h for the computation of finite-difference approximations of the Jacobian matrix of nonlinear constraints.

CLPARM= PL | WALD | BOTH

is similar to but not the same as that used by other SAS procedures. Using CLPARM=BOTH is equivalent to specifying

```
PROFILE / ALPHA=0.5 0.1 0.05 0.01 OUTTABLE;
```

The CLPARM=BOTH option specifies that profile confidence limits (PL CLs) for all parameters and for $\alpha = .5, .1, .05, .01$ are computed and displayed or written to the OUTEST= data set. Computing the profile confidence limits for all parameters can be very expensive and should be avoided when a difficult optimization problem or one with many parameters is solved. The OUTTABLE option is valid only when an OUTEST= data set is specified in the PROC NLP statement. For CLPARM=BOTH, the table of displayed output contains the Wald confidence limits computed from the standard errors as well as the PL CLs. The Wald confidence limits are not computed (displayed or written to the OUTEST= data set) unless the approximate covariance matrix of parameters is computed.

COV= 1 | 2 | 3 | 4 | 5 | 6 | M | H | J | B | E | U**COVARIANCE= 1 | 2 | 3 | 4 | 5 | 6 | M | H | J | B | E | U**

specifies one of six formulas for computing the covariance matrix. For more information, see the section “Covariance Matrix” on page 613.

COVSING=*r*

specifies a threshold $r > 0$ that determines whether the eigenvalues of a singular Hessian matrix or crossproduct Jacobian matrix are considered to be zero. For more information, see the section “Covariance Matrix” on page 613.

DAMPSTEP[=*r*]**DS[=*r*]**

specifies that the initial step length value $\alpha^{(0)}$ for each line search (used by the QUANEW, HYQUAN, CONGRA, or NEWRAP technique) cannot be larger than r times the step length value used in the former iteration. If the DAMPSTEP option is specified but r is not specified, the default is $r=2$. The DAMPSTEP= r option can prevent the line-search algorithm from repeatedly stepping into regions where some objective functions are difficult to compute or where they could lead to floating point overflows during the computation of objective functions and their derivatives. The DAMPSTEP= r option can save time-costly function calls during the line searches of objective functions that result in very small steps. For more information, see the section “Restricting the Step Length” on page 609.

DATA=SAS-data-set

allows variables from the specified data set to be used in the specification of the objective function f . For more information, see the section “DATA= Input Data Set” on page 616.

DIAHES

specifies that only the diagonal of the Hessian or crossproduct Jacobian is used. This saves function evaluations but may slow the convergence process considerably. Note that the DIAHES option refers to both the Hessian and the crossproduct Jacobian when using the LSQ statement. When derivatives are specified using the HESSIAN or CRPJAC statement, these statements must refer only to the n diagonal derivative elements (otherwise, the $n(n + 1)/2$ derivatives of the lower triangle must be specified). The DIAHES option is ignored if a quadratic programming with a constant Hessian is specified by TECH=QUADAS or TECH=LICOMP.

FCONV= $r[n]$ **FTOL**= $r[n]$

specifies the relative function convergence criterion. For all techniques except NMSIMP, termination requires a small relative change of the function value in successive iterations:

$$\frac{|f(x^{(k)}) - f(x^{(k-1)})|}{\max(|f(x^{(k-1)})|, FSIZE)} \leq r$$

where $FSIZE$ is defined by the **FSIZE**= option. For the NMSIMP technique, the same formula is used, but $x^{(k)}$ is defined as the vertex with the lowest function value, and $x^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is $r = 10^{-FDIGITS}$ where $FDIGITS$ is the value of the **FDIGITS**= option. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

FCONV2= $r[n]$ **FTOL2**= $r[n]$

specifies another function convergence criterion. For least squares problems and all techniques except NMSIMP, termination requires a small predicted reduction

$$df^{(k)} \approx f(x^{(k)}) - f(x^{(k)} + s^{(k)})$$

of the objective function. The predicted reduction

$$\begin{aligned} df^{(k)} &= -g^{(k)T} s^{(k)} - \frac{1}{2} s^{(k)T} G^{(k)} s^{(k)} \\ &= -\frac{1}{2} s^{(k)T} g^{(k)} \\ &\leq r \end{aligned}$$

is based on approximating the objective function f by the first two terms of the Taylor series and substituting the Newton step

$$s^{(k)} = -G^{(k)-1} g^{(k)}$$

For the NMSIMP technique, termination requires a small standard deviation of the function values of the $n + 1$ simplex vertices $x_l^{(k)}$, $l = 0, \dots, n$,

$$\sqrt{\frac{1}{n+1} \sum_l (f(x_l^{(k)}) - \bar{f}(x^{(k)}))^2} \leq r$$

where $\bar{f}(x^{(k)}) = \frac{1}{n+1} \sum_l f(x_l^{(k)})$. If there are n_{act} boundary constraints active at $x^{(k)}$, the mean and standard deviation are computed only for the $n + 1 - n_{act}$ unconstrained vertices. The default value is $r=1E-6$ for the NMSIMP technique and the QUANEW technique with nonlinear constraints, and $r=0$ otherwise. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

FD[=**FORWARD** | **CENTRAL** | *number*]

specifies that all derivatives be computed using finite-difference approximations. The following specifications are permitted:

- FD=FORWARD uses forward differences.
- FD=CENTRAL uses central differences.
- FD=*number* uses central differences for the initial and final evaluations of the gradient, Jacobian, and Hessian. During iteration, start with forward differences and switch to a corresponding central-difference formula during the iteration process when one of the following two criteria is satisfied:
- The absolute maximum gradient element is less than or equal to *number* times the **ABSGCONV** threshold.
 - The term left of the **GCONV** criterion is less than or equal to $\max(1.0E-6, \textit{number} \times \text{GCONV threshold})$. The $1.0E-6$ ensures that the switch is done, even if you set the **GCONV** threshold to zero.
- FD is equivalent to FD=100.

Note that the FD and FDHESSIAN options cannot apply at the same time. The FDHESSIAN option is ignored when only first-order derivatives are used, for example, when the **LSQ** statement is used and the **HESSIAN** is not explicitly needed (displayed or written to a data set). For more information, see the section “[Finite-Difference Approximations of Derivatives](#)” on page 601.

FDHESSIAN[=FORWARD | CENTRAL]

FDHES[=FORWARD | CENTRAL]

FDH[=FORWARD | CENTRAL]

specifies that second-order derivatives be computed using finite-difference approximations based on evaluations of the gradients.

- FDHESSIAN=FORWARD uses forward differences.
- FDHESSIAN=CENTRAL uses central differences.
- FDHESSIAN uses forward differences for the Hessian except for the initial and final output.

Note that the FD and FDHESSIAN options cannot apply at the same time. For more information, see the section “[Finite-Difference Approximations of Derivatives](#)” on page 601.

FDIGITS=*r*

specifies the number of accurate digits in evaluations of the objective function. Fractional values such as FDIGITS=4.7 are allowed. The default value is $r = -\log_{10}(\epsilon)$, where ϵ is the machine precision. The value of r is used to compute the interval length h for the computation of finite-difference approximations of the derivatives of the objective function and for the default value of the **FCONV**= option.

FDINT= OBJ | CON | ALL

specifies how the finite-difference intervals h should be computed. For FDINT=OBJ, the interval h is based on the behavior of the objective function; for FDINT=CON, the interval h is based on the behavior of the nonlinear constraints functions; and for FDINT=ALL, the interval h is based on the behavior of the objective function and the nonlinear constraints functions. For more information, see the section “[Finite-Difference Approximations of Derivatives](#)” on page 601.

FSIZE=*r*

specifies the FSIZE parameter of the relative function and relative gradient termination criteria. The default value is $r = 0$. For more details, refer to the FCONV= and GCONV= options.

G4=*n*

is used when the covariance matrix is singular. The value $n > 0$ determines which generalized inverse is computed. The default value of n is 60. For more information, see the section “Covariance Matrix” on page 613.

GCONV=*r*[*n*]**GTOL=*r*[*n*]**

specifies the relative gradient convergence criterion. For all techniques except the CONGRA and NMSIMP techniques, termination requires that the normalized predicted function reduction is small:

$$\frac{g(x^{(k)})^T [G^{(k)}]^{-1} g(x^{(k)})}{\max(|f(x^{(k)})|, FSIZE)} \leq r$$

where FSIZE is defined by the FSIZE= option. For the CONGRA technique (where a reliable Hessian estimate G is not available),

$$\frac{\|g(x^{(k)})\|_2^2 \|s(x^{(k)})\|_2}{\|g(x^{(k)}) - g(x^{(k-1)})\|_2 \max(|f(x^{(k)})|, FSIZE)} \leq r$$

is used. This criterion is not used by the NMSIMP technique. The default value is $r=1E-8$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

GCONV2=*r*[*n*]**GTOL2=*r*[*n*]**

specifies another relative gradient convergence criterion,

$$\max_j \frac{|g_j(x^{(k)})|}{\sqrt{f(x^{(k)})G_{j,j}^{(k)}}} \leq r$$

This option is valid only when using the TRUREG, LEVMAR, NRRIDG, and NEWRAP techniques on least squares problems. The default value is $r = 0$. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

GRADCHECK[= NONE | FAST | DETAIL]**GC[= NONE | FAST | DETAIL]**

Specifying GRADCHECK=DETAIL computes a test vector and test matrix to check whether the gradient g specified by a GRADIENT statement (or indirectly by a JACOBIAN statement) is appropriate for the function f computed by the program statements. If the specification of the first derivatives is correct, the elements of the test vector and test matrix should be relatively small. For very large optimization problems, the algorithm can be too expensive in terms of computer time and memory. If the GRADCHECK option is not specified, a fast derivative test identical to the GRADCHECK=FAST specification is performed by default. It is possible to suppress the default derivative test by specifying GRADCHECK=NONE. For more information, see the section “Testing the Gradient Specification” on page 604.

HESCAL= 0 | 1 | 2 | 3**HS= 0 | 1 | 2 | 3**

specifies the scaling version of the Hessian or crossproduct Jacobian matrix used in NRRIDG, TRUREG, LEVMAR, NEWRAP, or DBLDOG optimization. If the value of the HESCAL= option is not equal to zero, the first iteration and each restart iteration sets the diagonal scaling matrix $D^{(0)} = \text{diag}(d_i^{(0)})$:

$$d_i^{(0)} = \sqrt{\max(|G_{i,i}^{(0)}|, \epsilon)}$$

where $G_{i,i}^{(0)}$ are the diagonal elements of the Hessian or crossproduct Jacobian matrix. In all other iterations, the diagonal scaling matrix $D^{(0)} = \text{diag}(d_i^{(0)})$ is updated depending on the HESCAL= option:

HESCAL=0 specifies that no scaling is done

HESCAL=1 specifies the Moré (1978) scaling update:

$$d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

HESCAL=2 specifies the Dennis, Gay, and Welsch (1981) scaling update:

$$d_i^{(k+1)} = \max\left(0.6d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

HESCAL=3 specifies that d_i is reset in each iteration:

$$d_i^{(k+1)} = \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}$$

where ϵ is the relative machine precision. The default value is HESCAL=1 for LEVMAR minimization and HESCAL=0 otherwise. Scaling of the Hessian or crossproduct Jacobian matrix can be time-consuming in the case where general linear constraints are active.

INEST=SAS-data-set**INVAR=SAS-data-set****ESTDATA=SAS-data-set**

can be used to specify the initial values of the parameters defined in a **DECVAR** statement as well as simple boundary constraints and general linear constraints. The INEST= data set can contain additional variables with names corresponding to constants used in the program statements. At the beginning of each run of **PROC NLP**, the values of the constants are read from the PARMS observation, initializing the constants in the program statements. For more information, see the section “INEST= Input Data Set” on page 616.

INFEASIBLE**IFP**

specifies that the function values of both feasible and infeasible grid points are to be computed, displayed, and written to the **OUTEST=** data set, although only the feasible grid points are candidates for the starting point $x^{(0)}$. This option enables you to explore the shape of the objective function of points surrounding the feasible region. For the output, the grid points are sorted first with decreasing values of the maximum constraint violation. Points with the same value of the maximum constraint

violation are then sorted with increasing (minimization) or decreasing (maximization) value of the objective function. Using the **BEST=** option restricts only the number of best grid points in the displayed output, not those in the data set. The **INFEASIBLE** option affects both the displayed output and the output saved to the **OUTEST=** data set. The **OUTGRID** option can be used to write the grid points and their function values to an **OUTEST=** data set. After small modifications (deleting unneeded information), this data set can be used with the **G3D** procedure of **SAS/GRAPH** to generate a three-dimensional surface plot of the objective function depending on two selected parameters. For more information on grids, see the section “**DECVAR Statement**” on page 573.

INHESIAN[=r]

INHES[=r]

specifies how the initial estimate of the approximate Hessian is defined for the quasi-Newton techniques **QUANEW**, **DBLDOG**, and **HYQUAN**. There are two alternatives:

- The **= r** specification is not used: the initial estimate of the approximate Hessian is set to the true Hessian or crossproduct Jacobian at $x^{(0)}$.
- The **= r** specification is used: the initial estimate of the approximate Hessian is set to the multiple of the identity matrix rI .

By default, if **INHESIAN=r** is not specified, the initial estimate of the approximate Hessian is set to the multiple of the identity matrix rI , where the scalar r is computed from the magnitude of the initial gradient. For most applications, this is a sufficiently good first approximation.

INITIAL=r

specifies a value r as the common initial value for all parameters for which no other initial value assignments by the **DECVAR** statement or an **INEST=** data set are made.

INQUAD=SAS-data-set

can be used to specify (the nonzero elements of) the matrix H , the vector g , and the scalar c of a quadratic programming problem, $f(x) = \frac{1}{2}x^T Hx + g^T x + c$. This option cannot be used together with the **NLINCON** statement. Two forms (*dense* and *sparse*) of the **INQUAD=** data set can be used. For more information, see the section “**INQUAD= Input Data Set**” on page 617.

INSTEP=r

For highly nonlinear objective functions, such as the **EXP** function, the default initial radius of the trust region algorithms **TRUREG**, **DBLDOG**, or **LEVMAR** or the default step length of the line-search algorithms can result in arithmetic overflows. If this occurs, decreasing values of $0 < r < 1$ should be specified, such as **INSTEP=1E-1**, **INSTEP=1E-2**, **INSTEP=1E-4**, and so on, until the iteration starts successfully.

- For trust region algorithms (**TRUREG**, **DBLDOG**, **LEVMAR**), the **INSTEP=** option specifies a factor $r > 0$ for the initial radius $\Delta^{(0)}$ of the trust region. The default initial trust region radius is the length of the scaled gradient. This step corresponds to the default radius factor of $r = 1$.
- For line-search algorithms (**NEWRAP**, **CONGRA**, **QUANEW**, **HYQUAN**), the **INSTEP=** option specifies an upper bound for the initial step length for the line search during the first five iterations. The default initial step length is $r = 1$.
- For the Nelder-Mead simplex algorithm (**NMSIMP**), the **INSTEP=r** option defines the size of the initial simplex.

For more details, see the section “**Computational Problems**” on page 610.

LCDEACT=*r***LCD=*r***

specifies a threshold r for the Lagrange multiplier that decides whether an active inequality constraint remains active or can be deactivated. For a maximization (minimization), an active inequality constraint can be deactivated only if its Lagrange multiplier is greater (less) than the threshold value r . For maximization, r must be greater than zero; for minimization, r must be smaller than zero. The default value is

$$r = \pm \min(0.01, \max(0.1 \times \text{ABSGCONV}, 0.001 \times gmax^{(k)}))$$

where the $+$ stands for maximization, the $-$ for minimization, **ABSGCONV** is the value of the absolute gradient criterion, and $gmax^{(k)}$ is the maximum absolute element of the (projected) gradient $g^{(k)}$ or $Z^T g^{(k)}$.

LCEPSILON=*r***LCEPS=*r*****LCE=*r***

specifies the range $r > 0$ for active and violated boundary and linear constraints. During the optimization process, the introduction of rounding errors can force **PROC NLP** to increase the value of r by a factor of 10, 100, If this happens it is indicated by a message written to the log. For more information, see the section “**Linear Complementarity (LICOMP)**” on page 595.

LCSINGULAR=*r***LCSING=*r*****LCS=*r***

specifies a criterion $r > 0$ used in the update of the QR decomposition that decides whether an active constraint is linearly dependent on a set of other active constraints. The default value is $r=1\text{E}-8$. The larger r becomes, the more the active constraints are recognized as being linearly dependent. If the value of r is larger than 0.1, it is reset to 0.1.

LINESEARCH=*i***LIS=*i***

specifies the line-search method for the CONGRA, QUANEW, HYQUAN, and NEWRAP optimization techniques. Refer to Fletcher (1987) for an introduction to line-search techniques. The value of i can be 1, . . . , 8. For CONGRA, QUANEW, and NEWRAP, the default value is $i=2$. A special line-search method is the default for the least squares technique HYQUAN that is based on an algorithm developed by Lindström and Wedin (1984). Although it needs more memory, this default line-search method sometimes works better with large least squares problems. However, by specifying $\text{LIS}=i$, $i = 1, \dots, 8$, it is possible to use one of the standard techniques with HYQUAN.

LIS=1 specifies a line-search method that needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation.

LIS=2 specifies a line-search method that needs more function than gradient calls for quadratic and cubic interpolation and cubic extrapolation; this method is implemented as shown in Fletcher (1987) and can be modified to an exact line search by using the **LSPRECISION=** option.

LIS=3 specifies a line-search method that needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation; this method is implemented as

shown in Fletcher (1987) and can be modified to an exact line search by using the **LSPRECISION=** option.

LIS=4	specifies a line-search method that needs the same number of function and gradient calls for stepwise extrapolation and cubic interpolation.
LIS=5	specifies a line-search method that is a modified version of LIS=4.
LIS=6	specifies golden section line search (Polak 1971), which uses only function values for linear approximation.
LIS=7	specifies bisection line search (Polak 1971), which uses only function values for linear approximation.
LIS=8	specifies the Armijo line-search technique (Polak 1971), which uses only function values for linear approximation.

LIST

displays the model program and variable lists. The LIST option is a debugging feature and is not normally needed. This output is not included in either the default output or the output specified by the **PALL** option.

LISTCODE

displays the derivative tables and the compiled program code. The LISTCODE option is a debugging feature and is not normally needed. This output is not included in either the default output or the output specified by the **PALL** option. The option is similar to that used in MODEL procedure in SAS/ETS software.

LSPRECISION=*r*

LSP=*r*

specifies the degree of accuracy that should be obtained by the line-search algorithms **LIS=2** and **LIS=3**. Usually an imprecise line search is inexpensive and sufficient for convergence to the optimum. For difficult optimization problems, a more precise and expensive line search may be necessary (Fletcher 1987). The second (default for NEWRAP, QUANEW, and CONGRA) and third line-search methods approach exact line search for small LSPRECISION= values. In the presence of numerical problems, it is advised to decrease the LSPRECISION= value to obtain a more precise line search. The default values are as follows:

TECH=	UPDATE=	LSP default
QUANEW	DBFGS, BFGS	$r = 0.4$
QUANEW	DDFP, DFP	$r = 0.06$
HYQUAN	DBFGS	$r = 0.1$
HYQUAN	DDFP	$r = 0.06$
CONGRA	all	$r = 0.1$
NEWRAP	no update	$r = 0.9$

For more details, refer to Fletcher (1987).

MAXFUNC=*i*

MAXFU=*i*

specifies the maximum number *i* of function calls in the optimization process. The default values are

- TRUREG, LEVMAR, NRRIDG, NEWRAP: 125

- QUANEW, HYQUAN, DBLDOG: 500
- CONGRA, QUADAS: 1000
- NMSIMP: 3000

Note that the optimization can be terminated only after completing a full iteration. Therefore, the number of function calls that are actually performed can exceed the number that is specified by the MAXFUNC= option.

MAXITER=*i* [*n*]

MAXIT=*i* [*n*]

specifies the maximum number *i* of iterations in the optimization process. The default values are:

- TRUREG, LEVMAR, NRRIDG, NEWRAP: 50
- QUANEW, HYQUAN, DBLDOG: 200
- CONGRA, QUADAS: 400
- NMSIMP: 1000

This default value is valid also when *i* is specified as a missing value. The optional second value *n* is valid only for **TECH=QUANEW** with nonlinear constraints. It specifies an upper bound *n* for the number of iterations of an algorithm used to reduce the violation of nonlinear constraints at a starting point. The default value is *n*=20.

MAXSTEP=*r* [*n*]

specifies an upper bound for the step length of the line-search algorithms during the first *n* iterations. By default, *r* is the largest double precision value and *n* is the largest integer available. Setting this option can increase the speed of convergence for **TECH=CONGRA**, **TECH=QUANEW**, **TECH=HYQUAN**, and **TECH=NEWRAP**.

MAXTIME=*r*

specifies an upper limit of *r* seconds of real time for the optimization process. The default value is the largest floating point double representation of the computer. Note that the time specified by the MAXTIME= option is checked only once at the end of each iteration. Therefore, the actual running time of the PROC NLP job may be longer than that specified by the MAXTIME= option. The actual running time includes the rest of the time needed to finish the iteration, time for the output of the (temporary) results, and (if required) the time for saving the results in an **OUTEST=** data set. Using the MAXTIME= option with a permanent **OUTEST=** data set enables you to separate large optimization problems into a series of smaller problems that need smaller amounts of real time.

MINITER=*i*

MINIT=*i*

specifies the minimum number of iterations. The default value is zero. If more iterations than are actually needed are requested for convergence to a stationary point, the optimization algorithms can behave strangely. For example, the effect of rounding errors can prevent the algorithm from continuing for the required number of iterations.

MODEL=*model-name, model-list*

MOD=*model-name, model-list*

MODFILE=*model-name, model-list*

reads the program statements from one or more input model files created by previous **PROC NLP** steps using the **OUTMODEL=** option. If it is necessary to include the program code at a special location in newly written code, the **INCLUDE** statement can be used instead of using the **MODEL=** option. Using both the **MODEL=** option and the **INCLUDE** statement with the same model file will include the same model twice, which can produce different results than including it once. The **MODEL=** option is similar to the option used in **PROC MODEL** in SAS/ETS software.

MSINGULAR=*r*

MSING=*r*

specifies a relative singularity criterion $r > 0$ for measuring singularity of Hessian and crossproduct Jacobian and their projected forms. The default value is $1E-12$ if the **SINGULAR=** option is not specified and $\max(10 \times \epsilon, 1E - 4 \times SINGULAR)$ otherwise. For more information, see the section “Covariance Matrix” on page 613.

NOEIGNUM

suppresses the computation and output of the determinant and the inertia of the Hessian, crossproduct Jacobian, and covariance matrices. The inertia of a symmetric matrix are the numbers of negative, positive, and zero eigenvalues. For large applications, the **NOEIGNUM** option can save computer time.

NOMISS

is valid only for those variables of the **DATA=** data set that are referred to in program statements. If the **NOMISS** option is specified, observations with any missing value for those variables are skipped. If the **NOMISS** option is not specified, the missing value may result in a missing value of the objective function, implying that the corresponding **BY** group of data is not processed.

NOPRINT

NOP

suppresses the output.

OPTCHECK[=*r***]**

computes the function values $f(x_l)$ of a grid of points x_l in a small neighborhood of x^* . The x_l are located in a ball of radius r about x^* . If the **OPTCHECK** option is specified without r , the default value is $r = 0.1$ at the starting point and $r = 0.01$ at the terminating point. If a point x_l^* is found with a better function value than $f(x^*)$, then optimization is restarted at x_l^* . For more information on grids, see the section “**DECVAR Statement**” on page 573.

OUT=*SAS-data-set*

creates an output data set that contains those variables of a **DATA=** input data set referred to in the program statements plus additional variables computed by performing the program statements of the objective function, derivatives, and nonlinear constraints. The **OUT=** data set can also contain first- and second-order derivatives of these variables if the **OUTDER=** option is specified. The variables and derivatives are evaluated at x^* ; for **TECH=NONE**, they are evaluated at x^0 .

OUTALL

If an **OUTEST=** data set is specified, this option sets the **OUTHESSIAN** option if the **MIN** or **MAX** statement is used. If the **LSQ** statement is used, the **OUTALL** option sets the **OUTCRPJAC** option. If nonlinear constraints are specified using the **NLINCON** statement, the **OUTALL** option sets the **OUTNLCJAC** option.

OUTCRPJAC

If an **OUTEST=** data set is specified, the crossproduct Jacobian matrix of the m functions composing the least squares function is written to the **OUTEST=** data set.

OUTDER= 0 | 1 | 2

specifies whether or not derivatives are written to the **OUT=** data set. For **OUTDER=2**, first- and second-order derivatives are written to the data set; for **OUTDER=1**, only first-order derivatives are written; for **OUTDER=0**, no derivatives are written to the data set. The default value is **OUTDER=0**. Derivatives are evaluated at x^* .

OUTEST=SAS-data-set**OUTVAR=SAS-data-set**

creates an output data set that contains the results of the optimization. This is useful for reporting and for restarting the optimization in a subsequent execution of the procedure. Information in the data set can include parameter estimates, gradient values, constraint information, Lagrangian values, Hessian values, Jacobian values, covariance, standard errors, and confidence intervals.

OUTGRID

writes the grid points and their function values to the **OUTEST=** data set. By default, only the feasible grid points are saved; however, if the **INFEASIBLE** option is specified, all feasible and infeasible grid points are saved. Note that the **BEST=** option does not affect the output of grid points to the **OUTEST=** data set. For more information on grids, see the section “**DECVAR Statement**” on page 573.

OUTHESSIAN**OUTHES**

writes the Hessian matrix of the objective function to the **OUTEST=** data set. If the Hessian matrix is computed for some other reason (if, for example, the **PHESSIAN** option is specified), the **OUTHESSIAN** option is set by default.

OUTITER

writes during each iteration the parameter estimates, the value of the objective function, the gradient (if available), and (if **OUTTIME** is specified) the time in seconds from the start of the optimization to the **OUTEST=** data set.

OUTJAC

writes the Jacobian matrix of the m functions composing the least squares function to the **OUTEST=** data set. If the **PJACOBI** option is specified, the **OUTJAC** option is set by default.

OUTMODEL=model-name**OUTMOD=model-name****OUTM=model-name**

specifies the name of an output model file to which the program statements are to be written. The program statements of this file can be included into the program statements of a succeeding **PROC NLP** run using the **MODEL=** option or the **INCLUDE** program statement. The **OUTMODEL=** option is

similar to the option used in PROC MODEL in SAS/ETS software. Note that the following statements are not part of the program code that is written to an OUTMODEL= data set: **MIN**, **MAX**, **LSQ**, **MINQUAD**, **MAXQUAD**, **DECVAR**, **BOUNDS**, **BY**, **CRPJAC**, **GRADIENT**, **HESSIAN**, **JACNLC**, **JACOBIAN**, **LABEL**, **LINCON**, **MATRIX**, and **NLINCON**.

OUTNLCJAC

If an **OUTEST=** data set is specified, the Jacobian matrix of the nonlinear constraint functions specified by the **NLINCON** statement is written to the **OUTEST=** data set. If the Jacobian matrix of the nonlinear constraint functions is computed for some other reason (if, for example, the **PNLCJAC** option is specified), the **OUTNLCJAC** option is set by default.

OUTTIME

is used if an **OUTEST=** data set is specified and if the **OUTITER** option is specified. If **OUTTIME** is specified, the time in seconds from the start of the optimization to the start of each iteration is written to the **OUTEST=** data set.

PALL

ALL

displays all optional output except the output generated by the **PSTDERR**, **PCOV**, **LIST**, or **LISTCODE** option.

PCOV

displays the covariance matrix specified by the **COV=** option. The **PCOV** option is set automatically if the **PALL** and **COV=** options are set.

PCRPJAC

PJTJ

displays the $n \times n$ crossproduct Jacobian matrix $J^T J$. If the **PALL** option is specified and the **LSQ** statement is used, this option is set automatically. If general linear constraints are active at the solution, the projected crossproduct Jacobian matrix is also displayed.

PEIGVAL

displays the distribution of eigenvalues if a G4 inverse is computed for the covariance matrix. The **PEIGVAL** option is useful for observing which eigenvalues of the matrix are recognized as zero eigenvalues when the generalized inverse is computed, and it is the basis for setting the **COVSING=** option in a subsequent execution of **PROC NLP**. For more information, see the section “Covariance Matrix” on page 613.

PERROR

specifies additional output for such applications where the program code for objective function or nonlinear constraints cannot be evaluated during the iteration process. The **PERROR** option is set by default during the evaluations at the starting point but not during the optimization process.

PFUNCTION

displays the values of all functions specified in a **LSQ**, **MIN**, or **MAX** statement for each observation read from the **DATA=** input data set. The **PALL** option sets the **PFUNCTION** option automatically.

PGRID

displays the function values from the grid search. For more information on grids, see the section “[DECVAR Statement](#)” on page 573.

PHESSIAN**PHES**

displays the $n \times n$ Hessian matrix G . If the [PALL](#) option is specified and the [MIN](#) or [MAX](#) statement is used, this option is set automatically. If general linear constraints are active at the solution, the projected Hessian matrix is also displayed.

PHISTORY**PHIS**

displays the optimization history. No optimization history is displayed for [TECH=LICOMP](#). This output is included in both the default output and the output specified by the [PALL](#) option.

PINIT**PIN**

displays the initial values and derivatives (if available). This output is included in both the default output and the output specified by the [PALL](#) option.

PJACOBI**PJAC**

displays the $m \times n$ Jacobian matrix J . Because of the memory requirement for large least squares problems, this option is not invoked when using the [PALL](#) option.

PNLCJAC

displays the Jacobian matrix of nonlinear constraints specified by the [NLINCON](#) statement. The [PNLCJAC](#) option is set automatically if the [PALL](#) option is specified.

PSHORT**SHORT****PSH**

restricts the amount of default output. If [PSHORT](#) is specified, then

- The initial values are not displayed.
- The listing of constraints is not displayed.
- If there is more than one function in the [MIN](#), [MAX](#), or [LSQ](#) statement, their values are not displayed.
- If the [GRADCHECK](#) option is used, only the test vector is displayed.

PSTDERR**STDERR****SE**

computes standard errors that are defined as square roots of the diagonal elements of the covariance matrix. The t values and probabilities $> |t|$ are displayed together with the approximate standard errors. The type of covariance matrix must be specified using the [COV=](#) option. The [SIGSQ=](#) option, the [VARDEF=](#) option, and the special variables [_NOBS_](#) and [_DF_](#) defined in the program statements can be used to define a scalar factor σ^2 of the covariance matrix and the approximate standard errors. For more information, see the section “[Covariance Matrix](#)” on page 613.

PSUMMARY**SUMMARY****SUM**

restricts the amount of default displayed output to a short form of iteration history and notes, warnings, and errors.

PTIME

specifies the output of four different but partially overlapping differences of real time:

- total running time
- total time for the evaluation of objective function, nonlinear constraints, and derivatives: shows the total time spent executing the programming statements specifying the objective function, derivatives, and nonlinear constraints, and (if necessary) their first- and second-order derivatives. This is the total time needed for code evaluation before, during, and after iterating.
- total time for optimization: shows the total time spent iterating.
- time for some CMP parsing: shows the time needed for parsing the program statements and its derivatives. In most applications this is a negligible number, but for applications that contain ARRAY statements or DO loops or use an optimization technique with analytic second-order derivatives, it can be considerable.

RANDOM=*i*

specifies a positive integer as a seed value for the pseudorandom number generator. Pseudorandom numbers are used as the initial value $x^{(0)}$.

RESTART=*i***REST=*i***

specifies that the QUANEW, HYQUAN, or CONGRA algorithm is restarted with a steepest descent/ascent search direction after at most $i > 0$ iterations. Default values are as follows:

- CONGRA with **UPDATE=PB**: restart is done automatically so specification of i is not used
- CONGRA with **UPDATE≠PB**: $i = \min(10n, 80)$, where n is the number of parameters
- QUANEW, HYQUAN: i is the largest integer available

SIGSQ=*sq*

specifies a scalar factor $sq > 0$ for computing the covariance matrix. If the SIGSQ= option is specified, **VARDEF=N** is the default. For more information, see the section “Covariance Matrix” on page 613.

SINGULAR=*r***SING=*r***

specifies the singularity criterion $r > 0$ for the inversion of the Hessian matrix and crossproduct Jacobian. The default value is $1E-8$. For more information, refer to the **MSINGULAR=** and **VSINGULAR=** options.

TECH=*name*

TECHNIQUE=*name*

specifies the optimization technique. Valid values for it are as follows:

- **CONGRA**
chooses one of four different conjugate gradient optimization algorithms, which can be more precisely specified with the **UPDATE=** option and modified with the **LINESEARCH=** option. When this option is selected, **UPDATE=PB** by default. For $n \geq 400$, CONGRA is the default optimization technique.
- **DBLDOG**
performs a version of double dogleg optimization, which can be more precisely specified with the **UPDATE=** option. When this option is selected, **UPDATE=DBFGS** by default.
- **HYQUAN**
chooses one of three different hybrid quasi-Newton optimization algorithms which can be more precisely defined with the **VERSION=** option and modified with the **LINESEARCH=** option. By default, **VERSION=2** and **UPDATE=DBFGS**.
- **LEVVAR**
performs the Levenberg-Marquardt minimization. For $n < 40$, this is the default minimization technique for least squares problems.
- **LICOMP**
solves a quadratic program as a linear complementarity problem.
- **NMSIMP**
performs the Nelder-Mead simplex optimization method.
- **NONE**
does not perform any optimization. This option can be used
 - to do grid search without optimization
 - to compute and display derivatives and covariance matrices which cannot be obtained efficiently with any of the optimization techniques
- **NEWRAP**
performs the Newton-Raphson optimization technique. The algorithm combines a line-search algorithm with ridging. The line-search algorithm **LINESEARCH=2** is the default.
- **NRRIDG**
performs the Newton-Raphson optimization technique. For $n \leq 40$ and non-linear least squares, this is the default.
- **QUADAS**
performs a special quadratic version of the active set strategy.
- **QUANEW**
chooses one of four quasi-Newton optimization algorithms which can be defined more precisely with the **UPDATE=** option and modified with the **LINESEARCH=** option. This is the default for $40 < n < 400$ or if there are nonlinear constraints.
- **TRUREG**
performs the trust region optimization technique.

UPDATE=method**UPD=method**

specifies the update method for the (dual) quasi-Newton, double dogleg, hybrid quasi-Newton, or conjugate gradient optimization technique. Not every update method can be used with each optimizer. For more information, see the section “[Optimization Algorithms](#)” on page 591. Valid values for *method* are as follows:

BFGS	performs the original BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the inverse Hessian matrix.
DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix.
DDFP	performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.
DFP	performs the original DFP (Davidon, Fletcher, & Powell) update of the inverse Hessian matrix.
PB	performs the automatic restart update method of Powell (1977) and Beale (1972).
FR	performs the Fletcher-Reeves update (Fletcher 1987).
PR	performs the Polak-Ribiere update (Fletcher 1987).
CD	performs a conjugate-descent update of Fletcher (1987).

VARDEF= DF | N

specifies the divisor d used in the calculation of the covariance matrix and approximate standard errors. If the **SIGSQ=** option is not specified, the default value is VARDEF=DF; otherwise, VARDEF=N is the default. For more information, see the section “[Covariance Matrix](#)” on page 613.

VERSION= 1 | 2 | 3**VS= 1 | 2 | 3**

specifies the version of the hybrid quasi-Newton optimization technique or the version of the quasi-Newton optimization technique with nonlinear constraints.

For the hybrid quasi-Newton optimization technique,

VS=1 specifies version HY1 of Fletcher and Xu (1987).

VS=2 specifies version HY2 of Fletcher and Xu (1987).

VS=3 specifies version HY3 of Fletcher and Xu (1987).

For the quasi-Newton optimization technique with nonlinear constraints,

VS=1 specifies update of the μ vector like Powell (1978a, b) (update like VF02AD).

VS=2 specifies update of the μ vector like Powell (1982b) (update like VMCWD).

In both cases, the default value is VS=2.

VSINGULAR=*r***VSING=*r***

specifies a relative singularity criterion $r > 0$ for measuring singularity of Hessian and crossproduct Jacobian and their projected forms, which may have to be converted to compute the covariance matrix. The default value is $1E-8$ if the **SINGULAR=** option is not specified and the value of **SINGULAR** otherwise. For more information, see the section “Covariance Matrix” on page 613.

XCONV=*r*[*n*]**XTOL=*r*[*n*]**

specifies the relative parameter convergence criterion. For all techniques except NMSIMP, termination requires a small relative parameter change in subsequent iterations:

$$\frac{\max_j |x_j^{(k)} - x_j^{(k-1)}|}{\max(|x_j^{(k)}|, |x_j^{(k-1)}|, \text{XSIZE})} \leq r$$

For the NMSIMP technique, the same formula is used, but $x_j^{(k)}$ is defined as the vertex with the lowest function value and $x_j^{(k-1)}$ is defined as the vertex with the highest function value in the simplex. The default value is $r=1E-8$ for the NMSIMP technique and $r=0$ otherwise. The optional integer value n specifies the number of successive iterations for which the criterion must be satisfied before the process can be terminated.

XSIZE=*r*

specifies the parameter $r > 0$ of the relative parameter termination criterion. The default value is $r = 0$. For more details, see the **XCONV=** option.

ARRAY Statement

ARRAY *arrayname* [*dimensions*] [*\$*] [*variables and constants*] ; ;

The ARRAY statement is similar to, but not the same as, the ARRAY statement in the SAS DATA step. The ARRAY statement is used to associate a name (of no more than eight characters) with a list of variables and constants. The array name is used with subscripts in the program to refer to the array elements. The following code illustrates this:

```
array r[8] r1-r8;

do i = 1 to 8;
  r[i] = 0;
end;
```

The ARRAY statement does not support all the features of the DATA step ARRAY statement. It cannot be used to give initial values to array elements. Implicit indexing of variables cannot be used; all array references must have explicit subscript expressions. Only exact array dimensions are allowed; lower-bound specifications are not supported and a maximum of six dimensions is allowed.

On the other hand, the ARRAY statement does allow both variables and constants to be used as array elements. (Constant array elements cannot have values assigned to them.) Both dimension specification and the list of

elements are optional, but at least one must be given. When the list of elements is not given or fewer elements than the size of the array are listed, array variables are created by suffixing element numbers to the array name to complete the element list.

BOUNDS Statement

BOUNDS *b_con* [*, b_con...*];

where *b_con* is given in one of the following formats:

- number *operator* parameter_list *operator* number
- number *operator* parameter_list
- parameter_list *operator* number

and *operator* is \leq , $<$, \geq , $>$, or $=$.

Boundary constraints are specified with a BOUNDS statement. One- or two-sided boundary constraints are allowed. The list of boundary constraints are separated by commas. For example,

```
bounds 0 <= a1-a9 x <= 1, -1 <= c2-c5;
bounds b1-b10 y >= 0;
```

More than one BOUNDS statement can be used. If more than one lower (upper) bound for the same parameter is specified, the maximum (minimum) of these is taken. If the maximum l_j of all lower bounds is larger than the minimum of all upper bounds u_j for the same variable x_j , the boundary constraint is replaced by $x_j = l_j = \min(u_j)$ defined by the minimum of all upper bounds specified for x_j .

BY Statement

BY *variables* ;

A BY statement can be used with PROC NLP to obtain separate analyses on DATA= data set observations in groups defined by the BY variables. That means, for values of the TECH= option other than NONE, an optimization problem is solved for each BY group separately. When a BY statement appears, the procedure expects the input DATA= data set to be sorted in order of the BY variables. If the input data set is not sorted in ascending order, it is necessary to use one of the following alternatives:

- Use the SORT procedure with a similar BY statement to sort the data.
- Use the BY statement option NOTSORTED or DESCENDING in the BY statement for the NLP procedure. As a cautionary note, the NOTSORTED option does not mean that the data are unsorted but rather that the data are arranged in groups (according to values of the BY variables) and that these groups are not necessarily in alphabetical or increasing numeric order.
- Use the DATASETS procedure (in Base SAS software) to create an index on the BY variables.

For more information on the BY statement, refer to the discussion in *SAS Language Reference: Concepts*.
For more information on the DATASETS procedure, refer to the *SAS Procedures Guide*.

CRPJAC Statement

CRPJAC *variables* ;

The CRPJAC statement defines the crossproduct Jacobian matrix $J^T J$ used in solving least squares problems. For more information, see the section “Derivatives” on page 590. If the DIAHES option is not specified, the CRPJAC statement lists $n(n + 1)/2$ variable names, which correspond to the elements $(J^T J)_{j,k}$, $j \geq k$ of the lower triangle of the symmetric crossproduct Jacobian matrix listed by rows. For example, the statements

```
lsq f1-f3;
decvar x1-x3;
crpjac jj1-jj6;
```

correspond to the crossproduct Jacobian matrix

$$J^T J = \begin{bmatrix} JJ1 & JJ2 & JJ4 \\ JJ2 & JJ3 & JJ5 \\ JJ4 & JJ5 & JJ6 \end{bmatrix}$$

If the DIAHES option is specified, only the n diagonal elements must be listed in the CRPJAC statement. The n rows and n columns of the crossproduct Jacobian matrix must be in the same order as the n corresponding parameter names listed in the DECVAR statement. To specify the values of nonzero derivatives, the variables specified in the CRPJAC statement have to be defined at the left-hand side of algebraic expressions in programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=levmar;
  lsq f1 f2;
  decvar x1 x2;
  gradient g1 g2;
  crpjac cpj1-cpj3;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
  g1 = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
  g2 = 100 * (x2 - x1 * x1);
  cpj1 = 400 * x1 * x1 + 1 ;
  cpj2 = -200 * x1;
  cpj3 = 100;
run;
```

DECVAR Statement

DECVAR *name_list* [=numbers] [, *name_list* [=numbers] ...] ;

VAR *name_list* [=numbers] [, *name_list* [=numbers] ...] ;

PARMS *name_list* [=numbers] [, *name_list* [=numbers] ...] ;

PARAMETERS *name_list* [=numbers] [, *name_list* [=numbers] ...] ;

The DECVAR statement lists the names of the $n > 0$ decision variables and specifies grid search and initial values for an iterative optimization process. The decision variables listed in the DECVAR statement cannot also be used in the **MIN**, **MAX**, **MINQUAD**, **MAXQUAD**, **LSQ**, **GRADIENT**, **HESSIAN**, **JACOBIAN**, **CRPJAC**, or **NLINCON** statement.

The DECVAR statement contains a list of decision variable names (not separated by commas) optionally followed by an equals sign and a list of numbers. If the number list consists of only one number, this number defines the initial value for all the decision variables listed to the left of the equals sign.

If the number list consists of more than one number, these numbers specify the grid locations for each of the decision variables listed left of the equals sign. The **TO** and **BY** keywords can be used to specify a number list for a grid search. When a grid of points is specified with a DECVAR statement, **PROC NLP** computes the objective function value at each grid point and chooses the best (feasible) grid point as a starting point for the optimization process. The use of the **BEST=** option is recommended to save computing time and memory for the storing and sorting of all grid point information. Usually only feasible grid points are included in the grid search. If the specified grid contains points located outside the feasible region and you are interested in the function values at those points, it is possible to use the **INFEASIBLE** option to compute (and display) their function values as well.

GRADIENT Statement

GRADIENT *variables* ;

The GRADIENT statement defines the gradient vector which contains the first-order derivatives of the objective function f with respect to x_1, \dots, x_n . For more information, see the section “Derivatives” on page 590. To specify the values of nonzero derivatives, the variables specified in the GRADIENT statement must be defined on the left-hand side of algebraic expressions in programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=congra;
  min y;
  decvar x1 x2;
  gradient g1 g2;
  y1 = 10 * (x2 - x1 * x1);
  y2 = 1 - x1;
  y = .5 * (y1 * y1 + y2 * y2);
  g1 = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
  g2 = 100 * (x2 - x1 * x1);
run;
```

HESSIAN Statement

HESSIAN *variables* ;

The HESSIAN statement defines the Hessian matrix G containing the second-order derivatives of the objective function f with respect to x_1, \dots, x_n . For more information, see the section “Derivatives” on page 590.

If the **DIAHES** option is not specified, the **HESSIAN** statement lists $n(n + 1)/2$ variable names which correspond to the elements $G_{j,k}$, $j \geq k$, of the lower triangle of the symmetric Hessian matrix listed by rows. For example, the statements

```
min f;
decvar x1 - x3;
hessian g1-g6;
```

correspond to the Hessian matrix

$$G = \begin{bmatrix} G1 & G2 & G4 \\ G2 & G3 & G5 \\ G4 & G5 & G6 \end{bmatrix} = \begin{bmatrix} \partial^2 f / \partial x_1^2 & \partial^2 f / \partial x_1 \partial x_2 & \partial^2 f / \partial x_1 \partial x_3 \\ \partial^2 f / \partial x_2 \partial x_1 & \partial^2 f / \partial x_2^2 & \partial^2 f / \partial x_2 \partial x_3 \\ \partial^2 f / \partial x_3 \partial x_1 & \partial^2 f / \partial x_3 \partial x_2 & \partial^2 f / \partial x_3^2 \end{bmatrix}$$

If the **DIAHES** option is specified, only the n diagonal elements must be listed in the **HESSIAN** statement. The n rows and n columns of the Hessian matrix G must correspond to the order of the n parameter names listed in the **DECVAR** statement. To specify the values of nonzero derivatives, the variables specified in the **HESSIAN** statement must be defined on the left-hand side of algebraic expressions in the programming statements. For example, consider the Rosenbrock function:

```
proc nlp tech=nrridg;
  min f;
  decvar x1 x2;
  gradient g1 g2;
  hessian h1-h3;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
  f = .5 * (f1 * f1 + f2 * f2);
  g1 = -200 * x1 * (x2 - x1 * x1) - (1 - x1);
  g2 = 100 * (x2 - x1 * x1);
  h1 = -200 * (x2 - 3 * x1 * x1) + 1;
  h2 = -200 * x1;
  h3 = 100;
run;
```

INCLUDE Statement

INCLUDE *model files* ;

The **INCLUDE** statement can be used to append model code to the current model code. The contents of included model files, created using the **OUTMODEL=** option, are inserted into the model program at the position in which the **INCLUDE** statement appears.

JACNLC Statement

JACNLC *variables* ;

The **JACNLC** statement defines the Jacobian matrix for the system of constraint functions $c_1(x), \dots, c_{mc}(x)$. The statements list the $mc \times n$ variable names which correspond to the elements $CJ_{i,j}$, $i = 1, \dots, mc$; $j = 1, \dots, n$, of the Jacobian matrix by rows.

For example, the statements

```
nlincon c1-c3;
decvar x1-x2;
jacnlc cj1-cj6;
```

correspond to the Jacobian matrix

$$CJ = \begin{bmatrix} CJ1 & CJ2 \\ CJ3 & CJ4 \\ CJ5 & CJ6 \end{bmatrix} = \begin{bmatrix} \partial c_1/\partial x_1 & \partial c_1/\partial x_2 \\ \partial c_2/\partial x_1 & \partial c_2/\partial x_2 \\ \partial c_3/\partial x_1 & \partial c_3/\partial x_2 \end{bmatrix}$$

The mc rows of the Jacobian matrix must be in the same order as the mc corresponding names of nonlinear constraints listed in the **NLINCON** statement. The n columns of the Jacobian matrix must be in the same order as the n corresponding parameter names listed in the **DECVAR** statement. To specify the values of nonzero derivatives, the variables specified in the **JACNLC** statement must be defined on the left-hand side of algebraic expressions in programming statements.

For example,

```
array cd[3,4] cd1-cd12;
nlincon c1-c3 >= 0;
jacnlc cd1-cd12;

c1 = 8 - x1 * x1 - x2 * x2 - x3 * x3 - x4 * x4 -
      x1 + x2 - x3 + x4;
c2 = 10 - x1 * x1 - 2 * x2 * x2 - x3 * x3 - 2 * x4 * x4 +
      x1 + x4;
c3 = 5 - 2 * x1 * x2 - x2 * x2 - x3 * x3 - 2 * x1 + x2 + x4;

cd[1,1]= -1 - 2 * x1;   cd[1,2]= 1 - 2 * x2;
cd[1,3]= -1 - 2 * x3;   cd[1,4]= 1 - 2 * x4;
cd[2,1]= 1 - 2 * x1;   cd[2,2]= -4 * x2;
cd[2,3]= -2 * x3;     cd[2,4]= 1 - 4 * x4;
cd[3,1]= -2 - 4 * x1;  cd[3,2]= 1 - 2 * x2;
cd[3,3]= -2 * x3;     cd[3,4]= 1;
```

JACOBIAN Statement

JACOBIAN *variables* ;

The **JACOBIAN** statement defines the **JACOBIAN** matrix J for a system of objective functions. For more information, see the section “[Derivatives](#)” on page 590.

The **JACOBIAN** statement lists $m \times n$ variable names that correspond to the elements $J_{i,j}$, $i = 1, \dots, m$; $j = 1, \dots, n$, of the Jacobian matrix listed by rows.

For example, the statements

```
lsq f1-f3;
decvar x1 x2;
jacobian j1-j6;
```

correspond to the Jacobian matrix

$$J = \begin{bmatrix} J1 & J2 \\ J3 & J4 \\ J5 & J6 \end{bmatrix} = \begin{bmatrix} \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 \\ \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 \\ \partial f_3 / \partial x_1 & \partial f_3 / \partial x_2 \end{bmatrix}$$

The m rows of the Jacobian matrix must correspond to the order of the m function names listed in the **MIN**, **MAX**, or **LSQ** statement. The n columns of the Jacobian matrix must correspond to the order of the n decision variables listed in the **DECVAR** statement. To specify the values of nonzero derivatives, the variables specified in the **JACOBIAN** statement must be defined on the left-hand side of algebraic expressions in programming statements.

For example, consider the Rosenbrock function:

```
proc nlp tech=levmar;
  array j[2,2] j1-j4;
  lsq f1 f2;
  decvar x1 x2;
  jacobian j1-j4;
  f1 = 10 * (x2 - x1 * x1);
  f2 = 1 - x1;
  j[1,1] = -20 * x1;
  j[1,2] = 10;
  j[2,1] = -1;
  j[2,2] = 0; /* is not needed */
run;
```

The **JACOBIAN** statement is useful only if more than one objective function is given in the **MIN**, **MAX**, or **LSQ** statement, or if a **DATA=** input data set specifies more than one function. If the **MIN**, **MAX**, or **LSQ** statement contains only one objective function and no **DATA=** input data set is used, the **JACOBIAN** and **GRADIENT** statements are equivalent. In the case of least squares minimization, the crossproduct Jacobian is used as an approximate Hessian matrix.

LABEL Statement

LABEL *variable='label' [,variable='label'...] ;*

The **LABEL** statement can be used to assign labels (up to 40 characters) to the decision variables listed in the **DECVAR** statement. The **INEST=** data set can also be used to assign labels. The labels are attached to the output and are used in an **OUTEST=** data set.

LINCON Statement

LINCON *l_con [, l_con ...] ;*

where *l_con* is given in one of the following formats:

- *linear_term operator* number
- number *operator* *linear_term*

and `linear_term` is of the following form:

`< +|- > < number* > variable < +|- < number* > variable ... >`

The value of *operator* can be one of the following: `<`, `>`, `=`, `>=`, or `<=`.

The LINCON statement specifies equality or inequality constraints

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

separated by commas. For example, the constraint $4x_1 - 3x_2 = 0$ is expressed as

```
decvar x1 x2;
lincon 4 * x1 - 3 * x2 = 0;
```

and the constraints

$$10x_1 - x_2 \geq 10$$

$$x_1 + 5x_2 \geq 15$$

are expressed as

```
decvar x1 x2;
lincon 10 <= 10 * x1 - x2,
      x1 + 5 * x2 >= 15;
```

MATRIX Statement

MATRIX *M_name pattern_definitions* ;

The MATRIX statement defines a matrix H and the vector g , which can be given in the MINQUAD or MAXQUAD statement. The matrix H and vector g are initialized to zero, so that only the nonzero elements are given. The five different forms of the MATRIX statement are illustrated with the following example:

$$H = \begin{bmatrix} 100 & 10 & 1 & 0 \\ 10 & 100 & 10 & 1 \\ 1 & 10 & 100 & 10 \\ 0 & 1 & 10 & 100 \end{bmatrix} \quad g = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad c = 0$$

Each MATRIX statement first names the matrix or vector and then lists its elements. If more than one MATRIX statement is given for the same matrix, the later definitions override the earlier ones.

The rows and columns in matrix H and vector g correspond to the order of decision variables in the DECVAR statement.

- **Full Matrix Definition:** The MATRIX statement consists of *H_name* or *g_name* followed by an equals sign and all (nonredundant) numerical values of the matrix H or vector g . Assuming symmetry, only the elements of the lower triangular part of the matrix H must be listed. This specification should be used mainly for small problems with almost dense H matrices.

```

MATRIX H= 100
          10 100
           1  10 100
           0   1  10 100;
MATRIX G= 1  2  3  4;

```

- **Band-diagonal Matrix Definition:** This form of *pattern definition* is useful if the H matrix has (almost) constant band-diagonal structure. The MATRIX statement consists of H_name followed by empty brackets $[,]$, an equals sign, and a list of numbers to be assigned to the diagonal and successive subdiagonals.

```

MATRIX H[, ]= 100 10 1;
MATRIX G= 1  2  3  4;

```

- **Sparse Matrix Definitions:** In each of the following three specification types, the H_name or g_name is followed by a list of *pattern definitions* separated by commas. Each *pattern definition* consists of a location specification in brackets on the left side of an equals sign that is followed by a list of numbers.

- **(Sub)Diagonalwise:** This form of *pattern definition* is useful if the H matrix contains nonzero elements along diagonals or subdiagonals. The starting location is specified by an index pair in brackets $[i, j]$. The expression $k * num$ on the right-hand side specifies that num is assigned to the elements $[i, j], \dots, [i + k - 1, j + k - 1]$ in a diagonal direction of the H matrix. The special case $k = 1$ can be used to assign values to single nonzero element locations in H .

```

MATRIX H [1,1]= 4 * 100,
          [2,1]= 3 *  10,
          [3,1]= 2 *   1;
MATRIX G [1,1]= 1  2  3  4;

```

- **Columnwise Starting in Diagonal:** This form of *pattern definition* is useful if the H matrix contains nonzero elements columnwise starting in the diagonal. The starting location is specified by only one index j in brackets $[, j]$. The k numbers at the right-hand side are assigned to the elements $[j, j], \dots, [\min(j + k - 1, n), j]$.

```

MATRIX H [,1]= 100  10  1,
          [,2]= 100  10  1,
          [,3]= 100  10,
          [,4]= 100;
MATRIX G [,1]= 1  2  3  4;

```

- **Rowwise Starting in First Column:** This form of *pattern definition* is useful if the H matrix contains nonzero elements rowwise ending in the diagonal. The starting location is specified by only one index i in brackets $[i,]$. The k numbers at the right-hand side are assigned to the elements $[i, 1], \dots, [i, \min(k, i)]$.

```

MATRIX H [1, ]= 100,
          [2, ]=  10 100,
          [3, ]=   1  10 100,
          [4, ]=   0   1  10 100;
MATRIX G [1, ]= 1  2  3  4;

```

MIN, MAX, and LSQ Statements

MIN *variables* ;

MAX *variables* ;

LSQ *variables* ;

The MIN, MAX, or LSQ statement specifies the objective functions. Only one of the three statements can be used at a time and at least one must be given. The MIN and LSQ statements are for minimizing the objective function, and the MAX statement is for maximizing the objective function. The MIN, MAX, or LSQ statement lists one or more variables naming the objective functions $f_i, i = 1, \dots, m$ (later defined by SAS program code).

- If the MIN or MAX statement lists m function names f_1, \dots, f_m , the objective function f is

$$f(x) = \sum_{i=1}^m f_i$$

- If the LSQ statement lists m function names f_1, \dots, f_m , the objective function f is

$$f(x) = \frac{1}{2} \sum_{i=1}^m f_i^2(x)$$

Note that the LSQ statement can be used only if **TECH=LEVMAR** or **TECH=HYQUAN**.

MINQUAD and MAXQUAD Statements

MINQUAD *H_name* [, *g_name* [, *c_number*]] ;

MAXQUAD *H_name* [, *g_name* [, *c_number*]] ;

The MINQUAD and MAXQUAD statements specify the matrix H , vector g , and scalar c that define a quadratic objective function. The MINQUAD statement is for minimizing the objective function and the MAXQUAD statement is for maximizing the objective function.

The rows and columns in H and g correspond to the order of decision variables given in the **DECVAR** statement. Specifying the objective function with a MINQUAD or MAXQUAD statement indirectly defines the analytic derivatives for the objective function. Therefore, statements specifying derivatives are not valid in these cases. Also, only use these statements when **TECH=LICOMP** or **TECH=QUADAS** and no nonlinear constraints are imposed.

There are three ways of using the MINQUAD or MAXQUAD statement:

- **Using ARRAY Statements:**

The names *H_name* and *g_name* specified in the MINQUAD or MAXQUAD statement can be used in **ARRAY** statements. This specification is mainly for small problems with almost dense H matrices.

```

proc nlp pall;
  array h[2,2] .4 0
           0 4;
  minquad h, -100;
  decvar x1 x2 = -1;
  bounds 2 <= x1 <= 50,
         -50 <= x2 <= 50;
  lincon 10 <= 10 * x1 - x2;
run;

```

- **Using Elementwise Setting:**

The names H_name and g_name specified in the MINQUAD or MAXQUAD statement can be followed directly by one-dimensional indices specifying the corresponding elements of the matrix H and vector g . These element names can be used on the left side of numerical assignments. The one-dimensional index value l following H_name , which corresponds to the element H_{ij} , is computed by $l = (i - 1)n + j, i \geq j$. The matrix H and vector g are initialized to zero, so that only the nonzero elements must be given. This specification is efficient for small problems with sparse H matrices.

```

proc nlp pall;
  minquad h, -100;
  decvar x1 x2;
  bounds 2 <= x1 <= 50,
         -50 <= x2 <= 50;
  lincon 10 <= 10 * x1 - x2;
  h1 = .4; h4 = 4;
run;

```

- **Using MATRIX Statements:**

The names H_name and g_name specified in the MINQUAD or MAXQUAD statement can be used in **MATRIX** statements. There are different ways to specify the nonzero elements of the matrix H and vector g by **MATRIX** statements. The following example illustrates one way to use the **MATRIX** statement.

```

proc nlp all;
  matrix h[1,1] = .4 4;
  minquad h, -100;
  decvar x1 x2 = -1;
  bounds 2 <= x1 <= 50,
         -50 <= x2 <= 50;
  lincon 10 <= 10 * x1 - x2;
run;

```

NLINCON Statement

```
NLINCON nlcon [, nlcon ...] [/ option ] ;
```

```
NLC nlcon [, nlcon ...] [/ option ] ;
```

where *nlcon* is given in one of the following formats:

- number *operator* variable_list *operator* number
- number *operator* variable_list
- variable_list *operator* number

and *operator* is \leq , $<$, \geq , $>$, or $=$. The value of *option* can be SUMOBS or EVERYOBS.

General nonlinear equality and inequality constraints are specified with the NLINCON statement. The syntax of the NLINCON statement is similar to that of the **BOUNDS** statement with two small additions:

- The **BOUNDS** statement can contain only the names of decision variables. The NLINCON statement can also contain the names of continuous functions of the decision variables. These functions must be computed in the program statements, and since they can depend on the values of some of the variables in the **DATA=** data set, there are two possibilities:
 - If the continuous functions should be summed across all observations read from the **DATA=** data set, the NLINCON statement must be terminated by the / SUMOBS option.
 - If the continuous functions should be evaluated separately for each observation in the data set, the NLINCON statement must be terminated by the / EVERYOBS option. One constraint is generated for each observation in the data set.
- If the continuous function should be evaluated only once for the entire data set, the NLINCON statement has the same form as the **BOUNDS** statement. If this constraint does depend on the values of variables in the **DATA=** data set, it is evaluated using the data of the first observation.

One- or two-sided constraints can be specified in the NLINCON statement. However, equality constraints must be one-sided. The pairs of operators ($<$, $<=$) and ($>$, $>=$) are treated in the same way.

These three statements require the values of the three functions v_1 , v_2 , v_3 to be between zero and ten, and they are equivalent:

```
nlincon 0 <= v1-v3,  
          v1-v3 <= 10;
```

```
nlincon 0 <= v1-v3 <= 10;
```

```
nlincon 10 >= v1-v3 >= 0;
```

Also, consider the Rosen-Suzuki problem. It has three nonlinear inequality constraints:

$$\begin{aligned} 8 - x_1^2 - x_2^2 - x_3^2 - x_4^2 - x_1 + x_2 - x_3 + x_4 &\geq 0 \\ 10 - x_1^2 - 2x_2^2 - x_3^2 - 2x_4^2 + x_1 + x_4 &\geq 0 \\ 5 - 2x_1^2 - x_2^2 - x_3^2 - 2x_1 + x_2 + x_4 &\geq 0 \end{aligned}$$

These are specified as

```
nlincon c1-c3 >= 0;

c1 = 8 - x1 * x1 - x2 * x2 - x3 * x3 - x4 * x4 -
      x1 + x2 - x3 + x4;
c2 = 10 - x1 * x1 - 2 * x2 * x2 - x3 * x3 - 2 * x4 * x4 +
      x1 + x4;
c3 = 5 - 2 * x1 * x1 - x2 * x2 - x3 * x3 - 2 * x1 + x2 + x4;
```

NOTE: QUANEW and NMSIMP are the only optimization subroutines that support the NLINCON statement.

PROFILE Statement

PROFILE *parms* [/ [*ALPHA= values*] [*options*]] ;

where *parms* is given in the format *pnam_1 pnam_2 ... pnam_n*, and *values* is the list of α values in (0,1).

The PROFILE statement

- writes the (x , y) coordinates of profile points for each of the listed parameters to the **OUTEST=** data set
- displays, or writes to the **OUTEST=** data set, the profile likelihood confidence limits (PL CLs) for the listed parameters for the specified α values. If the approximate standard errors are available, the corresponding Wald confidence limits can be computed.

When computing the profile points or likelihood profile confidence intervals, **PROC NLP** assumes that a maximization of the log likelihood function is desired. Each point of the profile and each endpoint of the confidence interval is computed by solving corresponding nonlinear optimization problems.

The keyword **PROFILE** must be followed by the names of parameters for which the profile or the PL CLs should be computed. If the parameter name list is empty, the profiles and PL CLs for all parameters are computed. Then, optionally, the α values follow. The list of α values may contain **TO** and **BY** keywords. Each element must satisfy $0 < \alpha < 1$. The following is an example:

```
profile l11-l15 u1-u5 c /
      alpha= .9 to .1 by -.1 .09 to .01 by -.01;
```

Duplicate α values or values outside (0, 1) are automatically eliminated from the list.

A number of additional options can be specified.

FFACTOR= r	specifies the factor relating the discrepancy function $f(\theta)$ to the χ^2 quantile. The default value is $r = 2$.
FORCHI= F CHI	defines the scale for the y values written to the OUTEST= data set. For FORCHI=F, the y values are scaled to the values of the log likelihood function $f = f(\theta)$; for FORCHI=CHI, the y values are scaled so that $\hat{y} = \chi^2$. The default value is FORCHI=F.
FEASRATIO= r	specifies a factor of the Wald confidence limit (or an approximation of it if standard errors are not computed) defining an upper bound for the search for confidence limits. In general, the range of x values in the profile graph is between $r = 1$ and $r = 2$ times the length of the corresponding Wald interval. For many examples, the χ^2 quantiles corresponding to small α values define a y level $\hat{y} - \frac{1}{2}q_1(1 - \alpha)$, which is too far away from \hat{y} to be reached by $y(x)$ for x within the range of twice the Wald confidence limit. The search for an intersection with such a y level at a practically infinite value of x can be computationally expensive. A smaller value for r can speed up computation time by restricting the search for confidence limits to a region closer to \hat{x} . The default value of $r = 1000$ practically disables the FEASRATIO= option.
OUTTABLE	specifies that the complete set θ of parameter estimates rather than only $x = \theta_j$ for each confidence limit is written to the OUTEST= data set. This output can be helpful for further analyses on how small changes in $x = \theta_j$ affect the changes in the $\theta_i, i \neq j$.

For some applications, it may be computationally less expensive to compute the PL confidence limits for a few parameters than to compute the approximate covariance matrix of many parameters, which is the basis for the Wald confidence limits. However, the computation of the profile of the discrepancy function and the corresponding CLs in general will be much more time-consuming than that of the Wald CLs.

Program Statements

This section lists the program statements used to code the objective function and nonlinear constraints and their derivatives, and it documents the differences between program statements in the NLP procedure and program statements in the DATA step. The syntax of program statements used in **PROC NLP** is identical to that used in the CALIS, GENMOD, and MODEL procedures (refer to the *SAS/ETS User's Guide*).

Most of the program statements which can be used in the SAS DATA step can also be used in the NLP procedure. See the *SAS Language Guide* or base SAS documentation for a description of the SAS program statements.

```

ABORT;
CALL name [ ( expression [, expression ... ] ) ];
DELETE;
DO [ variable = expression
    [ TO expression ] [ BY expression ]
    [, expression [ TO expression ] [ BY expression ] ... ]
    ]
    [ WHILE expression ] [ UNTIL expression ];
END;
GOTO statement_label;
IF expression;
IF expression THEN program_statement;
    ELSE program_statement;
variable = expression;
variable + expression;
LINK statement_label;
PUT [ variable] [=] [...];
RETURN;
SELECT [ ( expression ) ];
STOP;
SUBSTR( variable, index, length ) = expression;
WHEN ( expression) program_statement;
    OTHERWISE program_statement;

```

statements work as they do in the SAS DATA step as documented in the *SAS Language Guide*. However, there are several differences that should be noted.

For the most part, the SAS program

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


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

 is supported; however,


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

 is not.
- The PUT statement, used mostly for program debugging in [PROC NLP](#), supports only some of the features of the DATA step PUT statement, and has some new features that the DATA step PUT statement does not:
 - The PROC NLP PUT statement does not support line pointers, factored lists, iteration factors, overprinting, `_INFILE_`, the colon (:) format modifier, or “\$”.
 - The PROC NLP PUT statement does support expressions, but the expression must be enclosed inside of parentheses. For example, the following statement displays the square root of x: `put (sqrt(x));`
 - The PROC NLP PUT statement supports the print item `_PDV_` to print a formatted listing of all variables in the program. For example, the following statement displays a more readable listing of the variables than the `_all_` print item: `put _pdv_;`

- The WHEN and OTHERWISE statements allow more than one target statement. That is, DO/END groups are not necessary for multiple statement WHENs. For example, the following syntax is valid:

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

It is recommended to keep some kind of order in the input of NLP, that is, between the statements that define decision variables and constraints and the program code used to specify objective functions and derivatives.

Use of Special Variables in Program Code

Except for the quadratic programming techniques (QUADAS and LICOMP) that do not execute program statements during the iteration process, several special variables in the program code can be used to communicate with PROC NLP in special situations:

- **_OBS_** If a DATA= input data set is used, it is possible to access a variable **_OBS_** which contains the number of the observation processed from the data set. You should not change the content of the **_OBS_** variable. This variable enables you to modify the programming statements depending on the observation number processed in the DATA= input data set. For example, to set variable A to 1 when observation 10 is processed, and otherwise to 2, it is possible to specify

```
IF _OBS_ = 10 THEN A=1; ELSE A=2;
```

- **_ITER_** This variable is set by PROC NLP, and it contains the number of the current iteration of the optimization technique as it is displayed in the optimization history. You should not change the content of the **_ITER_** variable. It is possible to read the value of this variable in order to modify the programming statements depending on the iteration number processed. For example, to display the content of the variables A, B, and C when there are more than 100 iterations processed, it is possible to use

```
IF _ITER_ > 100 THEN PUT A B C;
```

- **_DPROC_** This variable is set by PROC NLP to indicate whether the code is called only to obtain the values of the m objective functions f_i (**_DPROC_=0**) or whether specified derivatives (defined by the GRADIENT, JACOBIAN, CRPJAC, or HESSIAN statement) also have to be computed (**_DPROC_=1**). You should not change the content of the **_DPROC_** variable. Checking the **_DPROC_** variable makes it possible to save computer time by not performing derivative code that is not needed by the current call. In particular, when a DATA= input data set is used, the code is processed many times to compute only the function values. If the programming statements in the program contain the specification of computationally expensive first- and second-order derivatives, you can put the derivative code in an IF statement that is processed only if **_DPROC_** is not zero.

- **_INDF_** The **_INDF_** variable is set by **PROC NLP** to inform you of the source of calls to the function or derivative programming.

INDF=0 indicates the first function call in a grid search. This is also the first call evaluating the programming statements if there is a grid search defined by grid values in the **DECVAR** statement.

INDF=1 indicates further function calls in a grid search.

INDF=2 indicates the call for the feasible starting point. This is also the first call evaluating the programming statements if there is no grid search defined.

INDF=3 indicates calls from a gradient-checking algorithm.

INDF=4 indicates calls from the minimization algorithm. The **_ITER_** variable contains the iteration number.

INDF=5 If the active set algorithm leaves the feasible region (due to rounding errors), an algorithm tries to return it into the feasible region; **_INDF_=5** indicates a call that is done when such a step is successful.

INDF=6 indicates calls from a factorial test subroutine that tests the neighborhood of a point x for optimality.

INDF=7, 8 indicates calls from subroutines needed to compute finite-difference derivatives using only values of the objective function. No nonlinear constraints are evaluated.

INDF=9 indicates calls from subroutines needed to compute second-order finite-difference derivatives using analytic (specified) first-order derivatives. No nonlinear constraints are evaluated.

INDF=10 indicates calls where only the nonlinear constraints but no objective function are needed. The analytic derivatives of the nonlinear constraints are computed.

INDF=11 indicates calls where only the nonlinear constraints but no objective function are needed. The analytic derivatives of the nonlinear constraints are not computed.

INDF=-1 indicates the last call at the final solution.

You should not change the content of the **_INDF_** variable.

- **_LIST_** You can set the **_LIST_** variable to control the output during the iteration process:

LIST=0 is equivalent to the **NOPRINT** option. It suppresses all output.

LIST=1 is equivalent to the **PSUMMARY** but not the **PHISTORY** option. The optimization start and termination messages are displayed. However, the **PSUMMARY** option suppresses the output of the iteration history.

LIST=2 is equivalent to the **PSHORT** option or to a combination of the **PSUMMARY** and **PHISTORY** options. The optimization start information, the iteration history, and termination message are displayed.

LIST=3 is equivalent to not **PSUMMARY**, not **PSHORT**, and not **PALL**. The optimization start information, the iteration history, and the termination message are displayed.

LIST=4 is equivalent to the **PALL** option. The extended optimization start information (also containing settings of termination criteria and other control parameters) is displayed.

LIST=5 In addition to the iteration history, the vector $x^{(k)}$ of parameter estimates is displayed for each iteration k .

LIST=6 In addition to the iteration history, the vector $x^{(k)}$ of parameter estimates and the gradient $g^{(k)}$ (if available) of the objective function are displayed for each iteration k .

It is possible to set the **_LIST_** variable in the program code to obtain more or less output in each iteration of the optimization process. For example,

```
IF _ITER_ = 11      THEN _LIST_=5;
ELSE IF _ITER_ > 11 THEN _LIST_=1;
                   ELSE _LIST_=3;
```

- **_TOOBIG_** The value of **_TOOBIG_** is initialized to 0 by **PROC NLP**, but you can set it to 1 during the iteration, indicating problems evaluating the program statements. The objective function and derivatives must be computable at the starting point. However, during the iteration it is possible to set the **_TOOBIG_** variable to 1, indicating that the programming statements (computing the value of the objective function or the specified derivatives) cannot be performed for the current value of x_k . Some of the optimization techniques check the value of **_TOOBIG_** and try to modify the parameter estimates so that the objective function (or derivatives) can be computed in a following trial.
- **_NOBS_** The value of the **_NOBS_** variable is initialized by **PROC NLP** to the product of the number of functions *mfun* specified in the **MIN**, **MAX** or **LSQ** statement and the number of valid observations *nobs* in the current BY group of the **DATA=** input data set. The value of the **_NOBS_** variable is used for computing the scalar factor of the covariance matrix (see the **COV=**, **VARDEF=**, and **SIGSQ=** options). If you reset the value of the **_NOBS_** variable, the value that is available at the end of the iteration is used by **PROC NLP** to compute the scalar factor of the covariance matrix.
- **_DF_** The value of the **_DF_** variable is initialized by **PROC NLP** to the number n of parameters specified in the **DECVAR** statement. The value of the **_DF_** variable is used for computing the scalar factor d of the covariance matrix (see the **COV=**, **VARDEF=**, and **SIGSQ=** options). If you reset the value of the **_DF_** variable, the value that is available at the end of the iteration is used by **PROC NLP** to compute the scalar factor of the covariance matrix.
- **_LASTF_** In each iteration (except the first one), the value of the **_LASTF_** variable is set by **PROC NLP** to the final value of the objective function that was achieved during the last iteration. This value should agree with the value that is displayed in the iteration history and that is written in the **OUTEST=** data set when the **OUTITER** option is specified.

Details: NLP Procedure

Criteria for Optimality

PROC NLP solves

$$\begin{array}{ll} \min_{x \in \mathcal{R}^n} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i = 1, \dots, m_e \\ & c_i(x) \geq 0, \quad i = m_e + 1, \dots, m \end{array}$$

where f is the objective function and the c_i 's are the constraint functions.

A point x is feasible if it satisfies all the constraints. The feasible region \mathcal{G} is the set of all the feasible points. A feasible point x^* is a global solution of the preceding problem if no point in \mathcal{G} has a smaller function value than $f(x^*)$. A feasible point x^* is a local solution of the problem if there exists some open neighborhood surrounding x^* in that no point has a smaller function value than $f(x^*)$. Nonlinear programming algorithms cannot consistently find global minima. All the algorithms in PROC NLP find a local minimum for this problem. If you need to check whether the obtained solution is a global minimum, you may have to run PROC NLP with different starting points obtained either at random or by selecting a point on a grid that contains \mathcal{G} .

Every local minimizer x^* of this problem satisfies the following local optimality conditions:

- The gradient (vector of first derivatives) $g(x^*) = \nabla f(x^*)$ of the objective function f (projected toward the feasible region if the problem is constrained) at the point x^* is zero.
- The Hessian (matrix of second derivatives) $G(x^*) = \nabla^2 f(x^*)$ of the objective function f (projected toward the feasible region \mathcal{G} in the constrained case) at the point x^* is positive definite.

Most of the optimization algorithms in PROC NLP use iterative techniques that result in a sequence of points x^0, \dots, x^n, \dots , that converges to a local solution x^* . At the solution, PROC NLP performs tests to confirm that the (projected) gradient is close to zero and that the (projected) Hessian matrix is positive definite.

Karush-Kuhn-Tucker Conditions

An important tool in the analysis and design of algorithms in constrained optimization is the *Lagrangian function*, a linear combination of the objective function and the constraints:

$$L(x, \lambda) = f(x) - \sum_{i=1}^m \lambda_i c_i(x)$$

The coefficients λ_i are called *Lagrange multipliers*. This tool makes it possible to state necessary and sufficient conditions for a local minimum. The various algorithms in PROC NLP create sequences of points, each of which is closer than the previous one to satisfying these conditions.

Assuming that the functions f and c_i are twice continuously differentiable, the point x^* is a *local minimum* of the nonlinear programming problem, if there exists a vector $\lambda^* = (\lambda_1^*, \dots, \lambda_m^*)$ that meets the following conditions.

1. First-order Karush-Kuhn-Tucker conditions:

$$\begin{aligned} c_i(x^*) &= 0, & i &= 1, \dots, m_e \\ c_i(x^*) &\geq 0, & \lambda_i^* &\geq 0, \quad \lambda_i^* c_i(x^*) = 0, & i &= m_e + 1, \dots, m \\ \nabla_x L(x^*, \lambda^*) &= 0 \end{aligned}$$

2. Second-order conditions: Each nonzero vector $y \in \mathcal{R}^n$ that satisfies

$$y^T \nabla_x c_i(x^*) = 0 \begin{cases} i = 1, \dots, m_e \\ \forall i \in \{m_e + 1, \dots, m : \lambda_i^* > 0\} \end{cases}$$

also satisfies

$$y^T \nabla_x^2 L(x^*, \lambda^*) y > 0$$

Most of the algorithms to solve this problem attempt to find a combination of vectors x and λ for which the gradient of the Lagrangian function with respect to x is zero.

Derivatives

The first- and second-order conditions of optimality are based on first and second derivatives of the objective function f and the constraints c_i .

The gradient vector contains the first derivatives of the objective function f with respect to the parameters x_1, \dots, x_n , as follows:

$$g(x) = \nabla f(x) = \left(\frac{\partial f}{\partial x_j} \right)$$

The $n \times n$ symmetric Hessian matrix contains the second derivatives of the objective function f with respect to the parameters x_1, \dots, x_n , as follows:

$$G(x) = \nabla^2 f(x) = \left(\frac{\partial^2 f}{\partial x_j \partial x_k} \right)$$

For least squares problems, the $m \times n$ Jacobian matrix contains the first-order derivatives of the m objective functions $f_i(x)$ with respect to the parameters x_1, \dots, x_n , as follows:

$$J(x) = (\nabla f_1, \dots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

In the case of least squares problems, the crossproduct Jacobian

$$J^T J = \left(\sum_{i=1}^m \frac{\partial f_i}{\partial x_j} \frac{\partial f_i}{\partial x_k} \right)$$

is used as an approximate Hessian matrix. It is a very good approximation of the Hessian if the residuals at the solution are “small.” (If the residuals are not sufficiently small at the solution, this approach may result in

slow convergence.) The fact that it is possible to obtain Hessian approximations for this problem that do not require any computation of second derivatives means that least squares algorithms are more efficient than unconstrained optimization algorithms. Using the vector $f(x) = (f_1(x), \dots, f_m(x))^T$ of function values, PROC NLP computes the gradient $g(x)$ by

$$g(x) = J^T(x)f(x)$$

The $mc \times n$ Jacobian matrix contains the first-order derivatives of the mc nonlinear constraint functions $c_i(x)$, $i = 1, \dots, mc$, with respect to the parameters x_1, \dots, x_n , as follows:

$$CJ(x) = (\nabla c_1, \dots, \nabla c_{mc}) = \left(\frac{\partial c_i}{\partial x_j} \right)$$

PROC NLP provides three ways to compute derivatives:

- It computes analytical first- and second-order derivatives of the objective function f with respect to the n variables x_j .
- It computes first- and second-order finite-difference approximations to the derivatives. For more information, see the section “Finite-Difference Approximations of Derivatives” on page 601.
- The user supplies formulas for analytical or numerical first- and second-order derivatives of the objective function in the **GRADIENT**, **JACOBIAN**, **CRPJAC**, and **HESSIAN** statements. The **JACNLC** statement can be used to specify the derivatives for the nonlinear constraints.

Optimization Algorithms

There are three groups of optimization techniques available in PROC NLP. A particular optimizer can be selected with the **TECH=** option in the **PROC NLP** statement.

Table 6.2 Karush-Kuhn-Tucker Conditions

Algorithm	TECH=
Linear Complementarity Problem	LICOMP
Quadratic Active Set Technique	QUADAS
Trust-Region Method	TRUREG
Newton-Raphson Method with Line Search	NEWRAP
Newton-Raphson Method with Ridging	NRRIDG
Quasi-Newton Methods (DBFGS, DDFP, BFGS, DFP)	QUANEW
Double Dogleg Method (DBFGS, DDFP)	DBLDOG
Conjugate Gradient Methods (PB, FR, PR, CD)	CONGRA
Nelder-Mead Simplex Method	NMSIMP
Levenberg-Marquardt Method	LEVMAR
Hybrid Quasi-Newton Methods (DBFGS, DDFP)	HYQUAN

Since no single optimization technique is invariably superior to others, PROC NLP provides a variety of optimization techniques that work well in various circumstances. However, it is possible to devise problems for which none of the techniques in PROC NLP can find the correct solution. Moreover, nonlinear optimization can be computationally expensive in terms of time and memory, so care must be taken when matching an algorithm to a problem.

All optimization techniques in PROC NLP use $O(n^2)$ memory except the conjugate gradient methods, which use only $O(n)$ memory and are designed to optimize problems with many variables. Since the techniques are iterative, they require the repeated computation of

- the function value (optimization criterion)
- the gradient vector (first-order partial derivatives)
- for some techniques, the (approximate) Hessian matrix (second-order partial derivatives)
- values of linear and nonlinear constraints
- the first-order partial derivatives (Jacobian) of nonlinear constraints

However, since each of the optimizers requires different derivatives and supports different types of constraints, some computational efficiencies can be gained. The following table shows, for each optimization technique, which derivatives are needed (FOD: first-order derivatives; SOD: second-order derivatives) and what kinds of constraints (BC: boundary constraints; LIC: linear constraints; NLC: nonlinear constraints) are supported.

Algorithm	FOD	SOD	BC	LIC	NLC
LICOMP	-	-	x	x	-
QUADAS	-	-	x	x	-
TRUREG	x	x	x	x	-
NEWRAP	x	x	x	x	-
NRRIDG	x	x	x	x	-
QUANEW	x	-	x	x	x
DBLDOG	x	-	x	x	-
CONGRA	x	-	x	x	-
NMSIMP	-	-	x	x	x
LEVMAR	x	-	x	x	-
HYQUAN	x	-	x	x	-

Preparation for Using Optimization Algorithms

It is rare that a problem is submitted to an optimization algorithm “as is.” By making a few changes in your problem, you can reduce its complexity, which would increase the chance of convergence and save execution time.

- Whenever possible, use linear functions instead of nonlinear functions. PROC NLP will reward you with faster and more accurate solutions.

- Most optimization algorithms are based on quadratic approximations to nonlinear functions. You should try to avoid the use of functions that cannot be properly approximated by quadratic functions. Try to avoid the use of rational functions.

For example, the constraint

$$\frac{\sin(x)}{x+1} > 0$$

should be replaced by the equivalent constraint

$$\sin(x)(x+1) > 0$$

and the constraint

$$\frac{\sin(x)}{x+1} = 1$$

should be replaced by the equivalent constraint

$$\sin(x) - (x+1) = 0$$

- Try to avoid the use of exponential functions, if possible.
- If you can reduce the complexity of your function by the addition of a small number of variables, it may help the algorithm avoid stationary points.
- Provide the best starting point you can. A good starting point leads to better quadratic approximations and faster convergence.

Choosing an Optimization Algorithm

The factors that go into choosing a particular optimizer for a particular problem are complex and may involve trial and error. Several things must be taken into account. First, the structure of the problem has to be considered: Is it quadratic? least squares? Does it have linear or nonlinear constraints? Next, it is important to consider the type of derivatives of the objective function and the constraints that are needed and whether these are analytically tractable or not. This section provides some guidelines for making the right choices.

For many optimization problems, computing the gradient takes more computer time than computing the function value, and computing the Hessian sometimes takes *much* more computer time and memory than computing the gradient, especially when there are many decision variables. Optimization techniques that do not use the Hessian usually require more iterations than techniques that do use Hessian approximations (such as finite differences or BFGS update) and so are often slower. Techniques that do not use Hessians at all tend to be slow and less reliable.

The derivative compiler is not efficient in the computation of second-order derivatives. For large problems, memory and computer time can be saved by programming your own derivatives using the **GRADIENT**, **JACOBIAN**, **CRPJAC**, **HESSIAN**, and **JACNLC** statements. If you are not able to specify first- and second-order derivatives of the objective function, you can rely on finite-difference gradients and Hessian update formulas. This combination is frequently used and works very well for small and medium problems. For

large problems, you are advised not to use an optimization technique that requires the computation of second derivatives.

The following provides some guidance for matching an algorithm to a particular problem.

- Quadratic Programming
 - **QUADAS**
 - **LICOMP**
- General Nonlinear Optimization
 - Nonlinear Constraints
 - * **Small Problems: NMSIMP**
Not suitable for highly nonlinear problems or for problems with $n > 20$.
 - * **Medium Problems: QUANEW**
 - Only Linear Constraints
 - * **Small Problems: TRUREG (NEWRAP, NRRIDG)**
($n \leq 40$) where the Hessian matrix is not expensive to compute. Sometimes NRRIDG can be faster than TRUREG, but TRUREG can be more stable. NRRIDG needs only one matrix with $n(n + 1)/2$ double words; TRUREG and NEWRAP need two such matrices.
 - * **Medium Problems: QUANEW (DBLDOG)**
($n \leq 200$) where the objective function and the gradient are much faster to evaluate than the Hessian. QUANEW and DBLDOG in general need more iterations than TRUREG, NRRIDG, and NEWRAP, but each iteration can be much faster. QUANEW and DBLDOG need only the gradient to update an approximate Hessian. QUANEW and DBLDOG need slightly less memory than TRUREG or NEWRAP (essentially one matrix with $n(n + 1)/2$ double words).
 - * **Large Problems: CONGRA**
($n > 200$) where the objective function and the gradient can be computed much faster than the Hessian and where too much memory is needed to store the (approximate) Hessian. CONGRA in general needs more iterations than QUANEW or DBLDOG, but each iteration can be much faster. Since CONGRA needs only a factor of n double-word memory, many large applications of PROC NLP can be solved only by CONGRA.
 - * **No Derivatives: NMSIMP**
($n \leq 20$) where derivatives are not continuous or are very difficult to compute.
- Least Squares Minimization
 - **Small Problems: LEVMAR (HYQUAN)**
($n \leq 60$) where the crossproduct Jacobian matrix is inexpensive to compute. In general, LEVMAR is more reliable, but there are problems with high residuals where HYQUAN can be faster than LEVMAR.
 - **Medium Problems: QUANEW (DBLDOG)**
($n \leq 200$) where the objective function and the gradient are much faster to evaluate than the crossproduct Jacobian. QUANEW and DBLDOG in general need more iterations than LEVMAR or HYQUAN, but each iteration can be much faster.

- **Large Problems: CONGRA**
- **No Derivatives: NMSIMP**

Quadratic Programming Method

The QUADAS and LICOMP algorithms can be used to minimize or maximize a quadratic objective function,

$$f(x) = \frac{1}{2}x^T Gx + g^T x + c, \quad \text{with } G^T = G$$

subject to linear or boundary constraints

$$Ax \geq b \quad \text{or} \quad l_j \leq x_j \leq u_j$$

where $x = (x_1, \dots, x_n)^T$, $g = (g_1, \dots, g_n)^T$, G is an $n \times n$ symmetric matrix, A is an $m \times n$ matrix of general linear constraints, and $b = (b_1, \dots, b_m)^T$. The value of c modifies only the value of the objective function, not its derivatives, and the location of the optimizer x^* does not depend on the value of the constant term c . For QUADAS or LICOMP, the objective function must be specified using the **MINQUAD** or **MAXQUAD** statement or using an **INQUAD=** data set.

In this case, derivatives do not need to be specified because the gradient vector

$$\nabla f(x) = Gx + g$$

and the $n \times n$ Hessian matrix

$$\nabla^2 f(x) = G$$

are easily obtained from the data input.

Simple boundary and general linear constraints can be specified using the **BOUNDS** or **LINCON** statement or an **INQUAD=** or **INEST=** data set.

General Quadratic Programming (QUADAS)

The QUADAS algorithm is an active set method that iteratively updates the QT decomposition of the matrix A_k of active linear constraints and the Cholesky factor of the projected Hessian $Z_k^T G Z_k$ simultaneously. The update of active boundary and linear constraints is done separately; refer to Gill et al. (1984). Here Q is an $n_{free} \times n_{free}$ orthogonal matrix composed of vectors spanning the null space Z of A_k in its first $n_{free} - n_{alc}$ columns and range space Y in its last n_{alc} columns; T is an $n_{alc} \times n_{alc}$ triangular matrix of special form, $t_{ij} = 0$ for $i < n - j$, where n_{free} is the number of free parameters (n minus the number of active boundary constraints), and n_{alc} is the number of active linear constraints. The Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ and the QT decomposition are updated simultaneously when the active set changes.

Linear Complementarity (LICOMP)

The LICOMP technique solves a quadratic problem as a linear complementarity problem. It can be used only if G is positive (negative) semidefinite for minimization (maximization) and if the parameters are restricted to be positive.

This technique finds a point that meets the Karush-Kuhn-Tucker conditions by solving the linear complementarity problem

$$w = Mz + q$$

with constraints

$$w^T z \geq 0, \quad w \geq 0, \quad z \geq 0,$$

where

$$z = \begin{bmatrix} x \\ \lambda \end{bmatrix} \quad M = \begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix} \quad q = \begin{bmatrix} g \\ -b \end{bmatrix}$$

Only the `LCEPSILON=` option can be used to specify a tolerance used in computations.

General Nonlinear Optimization

Trust-Region Optimization (TRUREG)

The trust region method uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function $f(x)$ have continuous first- and second-order derivatives inside the feasible region.

The trust region method iteratively optimizes a quadratic approximation to the nonlinear objective function within a hyperelliptic trust region with radius Δ that constrains the step length corresponding to the quality of the quadratic approximation. The trust region method is implemented using Dennis, Gay, and Welsch (1981), Gay (1983).

The trust region method performs well for small to medium problems and does not require many function, gradient, and Hessian calls. If the computation of the Hessian matrix is computationally expensive, use the `UPDATE=` option for update formulas (that gradually build the second-order information in the Hessian). For larger problems, the conjugate gradient algorithm may be more appropriate.

Newton-Raphson Optimization With Line-Search (NEWRAP)

The NEWRAP technique uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function have continuous first- and second-order derivatives inside the feasible region. If second-order derivatives are computed efficiently and precisely, the NEWRAP method may perform well for medium to large problems, and it does not need many function, gradient, and Hessian calls.

This algorithm uses a pure Newton step when the Hessian is positive definite and when the Newton step reduces the value of the objective function successfully. Otherwise, a combination of ridging and line search is done to compute successful steps. If the Hessian is not positive definite, a multiple of the identity matrix is added to the Hessian matrix to make it positive definite (Eskow and Schnabel 1991).

In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation (`LIS=2`).

Newton-Raphson Ridge Optimization (NRRIDG)

The NRRIDG technique uses the gradient $g(x^{(k)})$ and Hessian matrix $G(x^{(k)})$ and thus requires that the objective function have continuous first- and second-order derivatives inside the feasible region.

This algorithm uses a pure Newton step when the Hessian is positive definite and when the Newton step reduces the value of the objective function successfully. If at least one of these two conditions is not satisfied, a multiple of the identity matrix is added to the Hessian matrix. If this algorithm is used for least squares problems, it performs a ridged Gauss-Newton minimization.

The NRRIDG method performs well for small to medium problems and does not need many function, gradient, and Hessian calls. However, if the computation of the Hessian matrix is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

Since NRRIDG uses an orthogonal decomposition of the approximate Hessian, each iteration of NRRIDG can be slower than that of NEWRAP, which works with Cholesky decomposition. However, usually NRRIDG needs fewer iterations than NEWRAP.

Quasi-Newton Optimization (QUANEW)

The (dual) quasi-Newton method uses the gradient $g(x^{(k)})$ and does not need to compute second-order derivatives since they are approximated. It works well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian, but in general it requires more iterations than the techniques TRUREG, NEWRAP, and NRRIDG, which compute second-order derivatives.

The QUANEW algorithm depends on whether or not there are nonlinear constraints.

Unconstrained or Linearly Constrained Problems If there are no nonlinear constraints, QUANEW is either

- the original quasi-Newton algorithm that updates an approximation of the inverse Hessian, or
- the dual quasi-Newton algorithm that updates the Cholesky factor of an approximate Hessian (default),

depending on the value of the `UPDATE=` option. For problems with general linear inequality constraints, the dual quasi-Newton methods can be more efficient than the original ones.

Four update formulas can be specified with the `UPDATE=` option:

DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.
DDFP	performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.
BFGS	performs the original BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the inverse Hessian matrix.
DFP	performs the original DFP (Davidon, Fletcher, & Powell) update of the inverse Hessian matrix.

In each iteration, a line search is done along the search direction to find an approximate optimum. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step length α satisfying the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit of the step length. Violating the left-side Goldstein condition can affect the positive definiteness of the quasi-Newton update. In those cases, either the update is skipped or the iterations are restarted with an identity matrix resulting in the steepest descent or ascent search direction. Line-search algorithms other than the default one can be specified with the `LINESEARCH=` option.

Nonlinearly Constrained Problems The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's (1978a, 1982b) *Variable Metric Constrained WatchDog* (VMCWD) algorithm. A similar but older algorithm (VF02AD) is part of the Harwell library. Both VMCWD and VF02AD use Fletcher's VE02AD algorithm (part of the Harwell library) for positive-definite quadratic programming. The PROC NLP QUANEW implementation uses a quadratic programming subroutine that updates and downdates the approximation of the Cholesky factor when the active set changes. The nonlinear QUANEW algorithm is not a feasible-point algorithm, and the value of the objective function need not decrease (minimization) or increase (maximization) monotonically. Instead, the algorithm tries to reduce a linear combination of the objective function and constraint violations, called the *merit function*.

The following are similarities and differences between this algorithm and the VMCWD algorithm:

- A modification of this algorithm can be performed by specifying **VERSION=1**, which replaces the update of the Lagrange vector μ with the original update of Powell (1978a, b) that is used in VF02AD. This can be helpful for some applications with linearly dependent active constraints.
- If the **VERSION** option is not specified or if **VERSION=2** is specified, the evaluation of the Lagrange vector μ is performed in the same way as Powell (1982b) describes.
- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS (or DFP) update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, a restart is done with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.
- The Cholesky factor is loaded into the quadratic programming subroutine, automatically ensuring positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ and the QT decomposition are updated simultaneously when the active set changes. Refer to Gill et al. (1984) for more information.
- The line-search strategy is very similar to that of Powell (1982b). However, this algorithm does not call for derivatives during the line search, so the algorithm generally needs fewer derivative calls than function calls. VMCWD always requires the same number of derivative and function calls. Sometimes Powell's line-search method uses steps that are too long. In these cases, use the **INSTEP=** option to restrict the step length α .
- The watchdog strategy is similar to that of Powell (1982b); however, it does not return automatically after a fixed number of iterations to a former better point. A return here is further delayed if the observed function reduction is close to the expected function reduction of the quadratic model.
- The Powell termination criterion still is used (as **FCONV2**) but the QUANEW implementation uses two additional termination criteria (**GCONV** and **ABSGCONV**).

The nonlinear QUANEW algorithm needs the Jacobian matrix of the first-order derivatives (constraints normals) of the constraints $CJ(x)$.

You can specify two update formulas with the **UPDATE=** option:

- | | |
|-------|--|
| DBFGS | performs the dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default. |
| DDFP | performs the dual DFP update of the Cholesky factor of the Hessian matrix. |

This algorithm uses its own line-search technique. No options or parameters (except the **INSTEP=** option) controlling the line search in the other algorithms apply here. In several applications, large steps in the first iterations were troublesome. You can use the **INSTEP=** option to impose an upper bound for the step length α during the first five iterations. You may also use the **INHESIAN=** option to specify a different starting approximation for the Hessian. Choosing simply the **INHESIAN** option will use the Cholesky factor of a (possibly ridged) finite-difference approximation of the Hessian to initialize the quasi-Newton update process. The values of the **LCSINGULAR=**, **LCEPSILON=**, and **LCDEACT=** options, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the nonlinear constraints QUANEW algorithm.

Double Dogleg Optimization (DBLDOG)

The double dogleg optimization method combines the ideas of the quasi-Newton and trust region methods. The double dogleg algorithm computes in each iteration the step $s^{(k)}$ as a linear combination of the steepest descent or ascent search direction $s_1^{(k)}$ and a quasi-Newton search direction $s_2^{(k)}$:

$$s^{(k)} = \alpha_1 s_1^{(k)} + \alpha_2 s_2^{(k)}$$

The step is requested to remain within a prespecified trust region radius; refer to Fletcher (1987, p. 107). Thus, the DBLDOG subroutine uses the dual quasi-Newton update but does not perform a line search. Two update formulas can be specified with the **UPDATE=** option:

DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, & Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.
DDFP	performs the dual DFP (Davidon, Fletcher, & Powell) update of the Cholesky factor of the Hessian matrix.

The double dogleg optimization technique works well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian. The implementation is based on Dennis and Mei (1979) and Gay (1983) but is extended for dealing with boundary and linear constraints. DBLDOG generally needs more iterations than the techniques TRUREG, NEWRAP, or NRRIDG that need second-order derivatives, but each of the DBLDOG iterations is computationally cheap. Furthermore, DBLDOG needs only gradient calls for the update of the Cholesky factor of an approximate Hessian.

Conjugate Gradient Optimization (CONGRA)

Second-order derivatives are not used by CONGRA. The CONGRA algorithm can be expensive in function and gradient calls but needs only $O(n)$ memory for unconstrained optimization. In general, many iterations are needed to obtain a precise solution, but each of the CONGRA iterations is computationally cheap. Four different update formulas for generating the conjugate directions can be specified using the **UPDATE=** option:

PB	performs the automatic restart update method of Powell (1977) and Beale (1972). This is the default.
FR	performs the Fletcher-Reeves update (Fletcher 1987).
PR	performs the Polak-Ribiere update (Fletcher 1987).
CD	performs a conjugate-descent update of Fletcher (1987).

The default value is `UPDATE=PB`, since it behaved best in most test examples. You are advised to avoid the option `UPDATE=CD`, as it behaved worst in most test examples.

The CONGRA subroutine should be used for optimization problems with large n . For the unconstrained or boundary constrained case, CONGRA needs only $O(n)$ bytes of working memory, whereas all other optimization methods require order $O(n^2)$ bytes of working memory. During n successive iterations, uninterrupted by restarts or changes in the working set, the conjugate gradient algorithm computes a cycle of n conjugate search directions. In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step length α satisfying the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit for the step length. Other line-search algorithms can be specified with the `LINESEARCH=` option.

Nelder-Mead Simplex Optimization (NMSIMP)

The Nelder-Mead simplex method does not use any derivatives and does not assume that the objective function has continuous derivatives. The objective function itself needs to be continuous. This technique requires a large number of function evaluations. It is unlikely to give accurate results for $n \gg 40$.

Depending on the kind of constraints, one of the following Nelder-Mead simplex algorithms is used:

- unconstrained or only boundary constrained problems

The original Nelder-Mead simplex algorithm is implemented and extended to boundary constraints. This algorithm does not compute the objective for infeasible points. This algorithm is automatically invoked if the `LINCON` or `NLINCON` statement is not specified.

- general linearly constrained or nonlinearly constrained problems

A slightly modified version of Powell's (1992) COBYLA (Constrained Optimization BY Linear Approximations) implementation is used. This algorithm is automatically invoked if either the `LINCON` or the `NLINCON` statement is specified.

The original Nelder-Mead algorithm cannot be used for general linear or nonlinear constraints but can be faster for the unconstrained or boundary constrained case. The original Nelder-Mead algorithm changes the shape of the simplex adapting to the nonlinearities of the objective function which contributes to an increased speed of convergence. The two NMSIMP subroutines use special sets of termination criteria. For more details, refer to the section "Termination Criteria" on page 605.

Powell's COBYLA Algorithm (COBYLA)

Powell's COBYLA algorithm is a sequential trust region algorithm (originally with a monotonically decreasing radius ρ of a spherical trust region) that tries to maintain a regular-shaped simplex over the iterations. A small modification was made to the original algorithm that permits an increase of the trust region radius ρ in special situations. A sequence of iterations is performed with a constant trust region radius ρ until the computed objective function reduction is much less than the predicted reduction. Then, the trust region radius ρ is reduced. The trust region radius is increased only if the computed function reduction is relatively close to the predicted reduction and the simplex is well-shaped. The start radius ρ_{beg} and the final radius ρ_{end} can be specified using `ρ_{beg} =INSTEP` and `ρ_{end} =ABSXTOL`. The convergence to small values of ρ_{end} (high precision) may take many calls of the function and constraint modules and may result in numerical problems. There are two main reasons for the slow convergence of the COBYLA algorithm:

- Only linear approximations of the objective and constraint functions are used locally.
- Maintaining the regular-shaped simplex and not adapting its shape to nonlinearities yields very small simplices for highly nonlinear functions (for example, fourth-order polynomials).

Nonlinear Least Squares Optimization

Levenberg-Marquardt Least Squares Method (LEVMar)

The Levenberg-Marquardt method is a modification of the trust region method for nonlinear least squares problems and is implemented as in Moré (1978).

This is the recommended algorithm for small to medium least squares problems. Large least squares problems can be transformed into minimization problems, which can be processed with conjugate gradient or (dual) quasi-Newton techniques. In each iteration, LEVMAR solves a quadratically constrained quadratic minimization problem that restricts the step to stay at the surface of or inside an n -dimensional elliptical (or spherical) trust region. In each iteration, LEVMAR uses the crossproduct Jacobian matrix $J^T J$ as an approximate Hessian matrix.

Hybrid Quasi-Newton Least Squares Methods (HYQUAN)

In each iteration of one of the Fletcher and Xu (1987) (refer also to Al-Baali and Fletcher (1985,1986)) hybrid quasi-Newton methods, a criterion is used to decide whether a Gauss-Newton or a dual quasi-Newton search direction is appropriate. The **VERSION=** option can be used to choose one of three criteria (HY1, HY2, HY3) proposed by Fletcher and Xu (1987). The default is **VERSION=2**; that is, HY2. In each iteration, HYQUAN computes the crossproduct Jacobian (used for the Gauss-Newton step), updates the Cholesky factor of an approximate Hessian (used for the quasi-Newton step), and does a line search to compute an approximate minimum along the search direction. The default line-search technique used by HYQUAN is especially designed for least squares problems (refer to Lindström and Wedin (1984) and Al-Baali and Fletcher (1986)). Using the **LINESEARCH=** option you can choose a different line-search algorithm than the default one.

Two update formulas can be specified with the **UPDATE=** option:

DBFGS	performs the dual BFGS (Broyden, Fletcher, Goldfarb, and Shanno) update of the Cholesky factor of the Hessian matrix. This is the default.
DDFP	performs the dual DFP (Davidon, Fletcher, and Powell) update of the Cholesky factor of the Hessian matrix.

The HYQUAN subroutine needs about the same amount of working memory as the LEVMAR algorithm. In most applications, LEVMAR seems to be superior to HYQUAN, and using HYQUAN is recommended only when problems are experienced with the performance of LEVMAR.

Finite-Difference Approximations of Derivatives

The **FD=** and **FDHESSIAN=** options specify the use of finite-difference approximations of the derivatives. The **FD=** option specifies that all derivatives are approximated using function evaluations, and the **FDHESSIAN=** option specifies that second-order derivatives are approximated using gradient evaluations.

Computing derivatives by finite-difference approximations can be very time-consuming, especially for second-order derivatives based only on values of the objective function (**FD=** option). If analytical derivatives are difficult to obtain (for example, if a function is computed by an iterative process), you might consider one of the optimization techniques that uses first-order derivatives only (**TECH=QUANEW**, **TECH=DBLDOG**, or **TECH=CONGRA**).

Forward-Difference Approximations

The forward-difference derivative approximations consume less computer time but are usually not as precise as those using central-difference formulas.

- First-order derivatives: n additional function calls are needed:

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$$

- Second-order derivatives based on function calls only (Dennis and Schnabel 1983, p. 80, 104): for dense Hessian, $n(n + 3)/2$ additional function calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_j}$$

- Second-order derivatives based on gradient calls (Dennis and Schnabel 1983, p. 103): n additional gradient calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x + h_j e_j) - g_i(x)}{2h_j} + \frac{g_j(x + h_i e_i) - g_j(x)}{2h_i}$$

Central-Difference Approximations

- First-order derivatives: $2n$ additional function calls are needed:

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i}$$

- Second-order derivatives based on function calls only (Abramowitz and Stegun 1972, p. 884): for dense Hessian, $2n(n + 1)$ additional function calls are needed:

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}$$

- Second-order derivatives based on gradient: $2n$ additional gradient calls are needed:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x + h_j e_j) - g_i(x - h_j e_j)}{4h_j} + \frac{g_j(x + h_i e_i) - g_j(x - h_i e_i)}{4h_i}$$

The **FDIGITS=** and **CDIGITS=** options can be used for specifying the number of accurate digits in the evaluation of objective function and nonlinear constraints. These specifications are helpful in determining an appropriate interval length h to be used in the finite-difference formulas.

The **FDINT=** option specifies whether the finite-difference intervals h should be computed using an algorithm of Gill, Murray, Saunders, and Wright (1983) or based only on the information of the **FDIGITS=** and **CDIGITS=** options. For **FDINT=OBJ**, the interval h is based on the behavior of the objective function; for **FDINT=CON**, the interval h is based on the behavior of the nonlinear constraints functions; and for **FDINT=ALL**, the interval h is based on the behaviors of both the objective function and the nonlinear constraints functions. Note that the algorithm of Gill, Murray, Saunders, and Wright (1983) to compute the finite-difference intervals h_j can be very expensive in the number of function calls. If the **FDINT=** option is specified, it is currently performed twice, the first time before the optimization process starts and the second time after the optimization terminates.

If **FDINT=** is not specified, the step lengths h_j , $j = 1, \dots, n$, are defined as follows:

- for the forward-difference approximation of first-order derivatives using function calls and second-order derivatives using gradient calls: $h_j = \sqrt[2]{\eta_j}(1 + |x_j|)$,
- for the forward-difference approximation of second-order derivatives that use only function calls and all central-difference formulas: $h_j = \sqrt[3]{\eta_j}(1 + |x_j|)$,

where η is defined using the **FDIGITS=** option:

- If the number of accurate digits is specified with **FDIGITS=r**, η is set to 10^{-r} .
- If **FDIGITS=** is not specified, η is set to the machine precision ϵ .

For **FDINT=OBJ** and **FDINT=ALL**, the **FDIGITS=** specification is used in computing the forward and central finite-difference intervals.

If the problem has nonlinear constraints and the **FD=** option is specified, the first-order formulas are used to compute finite-difference approximations of the Jacobian matrix $JC(x)$. You can use the **CDIGITS=** option to specify the number of accurate digits in the constraint evaluations to define the step lengths h_j , $j = 1, \dots, n$. For **FDINT=CON** and **FDINT=ALL**, the **CDIGITS=** specification is used in computing the forward and central finite-difference intervals.

NOTE: If you are unable to specify analytic derivatives and the finite-difference approximations provided by PROC NLP are not good enough to solve your problem, you may program better finite-difference approximations using the **GRADIENT**, **JACOBIAN**, **CRPJAC**, or **HESSIAN** statement and the program statements.

Hessian and CRP Jacobian Scaling

The rows and columns of the Hessian and crossproduct Jacobian matrix can be scaled when using the trust region, Newton-Raphson, double dogleg, and Levenberg-Marquardt optimization techniques. Each element $G_{i,j}$, $i, j = 1, \dots, n$, is divided by the scaling factor $d_i \times d_j$, where the scaling vector $d = (d_1, \dots, d_n)$ is iteratively updated in a way specified by the **HESCAL=i** option, as follows:

$i = 0$ No scaling is done (equivalent to $d_i = 1$).

$i \neq 0$ First iteration and each restart iteration:

$$d_i^{(0)} = \sqrt{\max(|G_{i,i}^{(0)}|, \epsilon)}$$

$i = 1$ refer to Moré (1978):

$$d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

$i = 2$ refer to Dennis, Gay, and Welsch (1981):

$$d_i^{(k+1)} = \max\left(0.6d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$

$i = 3$ d_i is reset in each iteration:

$$d_i^{(k+1)} = \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}$$

where ϵ is the relative machine precision or, equivalently, the largest double precision value that when added to 1 results in 1.

Testing the Gradient Specification

There are three main ways to check the correctness of derivative specifications:

- Specify the `FD=` or `FDHESSIAN=` option in the `PROC NLP` statement to compute finite-difference approximations of first- and second-order derivatives. In many applications, the finite-difference approximations are computed with high precision and do not differ too much from the derivatives that are computed by specified formulas.
- Specify the `GRADCHECK` option in the `PROC NLP` statement to compute and display a test vector and a test matrix of the gradient values at the starting point $x^{(0)}$ by the method of Wolfe (1982). If you do not specify the `GRADCHECK` option, a fast derivative test identical to the `GRADCHECK=FAST` specification is done by default.
- If the default analytical derivative compiler is used or if derivatives are specified using the `GRADIENT` or `JACOBIAN` statement, the gradient or Jacobian computed at the initial point $x^{(0)}$ is tested by default using finite-difference approximations. In some examples, the relative test can show significant differences between the two forms of derivatives, resulting in a warning message indicating that the specified derivatives could be wrong, even if they are correct. This happens especially in cases where the magnitude of the gradient at the starting point $x^{(0)}$ is small.

The algorithm of Wolfe (1982) is used to check whether the gradient $g(x)$ specified by a `GRADIENT` statement (or indirectly by a `JACOBIAN` statement) is appropriate for the objective function $f(x)$ specified by the program statements.

Using function and gradient evaluations in the neighborhood of the starting point $x^{(0)}$, second derivatives are approximated by finite-difference formulas. Forward differences of gradient values are used to approximate the Hessian element G_{jk} ,

$$G_{jk} \approx H_{jk} = \frac{g_j(x + \delta e_k) - g_j(x)}{\delta}$$

where δ is a small step length and $e_k = (0, \dots, 0, 1, 0, \dots, 0)^T$ is the unit vector along the k th coordinate axis. The test vector s , with

$$s_j = H_{jj} - \frac{2}{\delta} \left\{ \frac{f(x + \delta e_j) - f(x)}{\delta} - g_j(x) \right\}$$

contains the differences between two sets of finite-difference approximations for the diagonal elements of the Hessian matrix

$$G_{jj} = \partial^2 f(x^{(0)}) / \partial x_j^2, \quad j = 1, \dots, n$$

The test matrix ΔH contains the absolute differences of symmetric elements in the approximate Hessian $|H_{jk} - H_{kj}|$, $j, k = 1, \dots, n$, generated by forward differences of the gradient elements.

If the specification of the first derivatives is correct, the elements of the test vector and test matrix should be relatively small. The location of large elements in the test matrix points to erroneous coordinates in the gradient specification. For very large optimization problems, this algorithm can be too expensive in terms of computer time and memory.

Termination Criteria

All optimization techniques stop iterating at $x^{(k)}$ if at least one of a set of termination criteria is satisfied. PROC NLP also terminates if the point $x^{(k)}$ is fully constrained by n linearly independent active linear or boundary constraints, and all Lagrange multiplier estimates of active inequality constraints are greater than a small negative tolerance.

Since the Nelder-Mead simplex algorithm does not use derivatives, no termination criterion is available based on the gradient of the objective function. Powell's COBYLA algorithm uses only one more termination criterion. COBYLA is a trust region algorithm that sequentially reduces the radius ρ of a spherical trust region from a start radius $\rho_{beg} = \text{INSTEP}$ to the final radius $\rho_{end} = \text{ABSXTOL}$. The default value is $\rho_{end} = 1\text{E}-4$. The convergence to small values of ρ_{end} (high precision) may take many calls of the function and constraint modules and may result in numerical problems.

In some applications, the small default value of the **ABSGCONV=** criterion is too difficult to satisfy for some of the optimization techniques. This occurs most often when finite-difference approximations of derivatives are used.

The default setting for the **GCONV=** option sometimes leads to early termination far from the location of the optimum. This is especially true for the special form of this criterion used in the CONGRA optimization.

The QUANEW algorithm for nonlinearly constrained optimization does not monotonically reduce the value of either the objective function or some kind of merit function which combines objective and constraint functions. Furthermore, the algorithm uses the watchdog technique with backtracking (Chamberlain et al. 1982). Therefore, no termination criteria were implemented that are based on the values (x or f) of successive

iterations. In addition to the criteria used by all optimization techniques, three more termination criteria are currently available. They are based on satisfying the Karush-Kuhn-Tucker conditions, which require that the gradient of the Lagrange function is zero at the optimal point (x^*, λ^*) :

$$\nabla_x L(x^*, \lambda^*) = 0$$

For more information, refer to the section “Criteria for Optimality” on page 589.

Active Set Methods

The parameter vector $x \in \mathcal{R}^n$ may be subject to a set of m linear equality and inequality constraints:

$$\begin{aligned} \sum_{j=1}^n a_{ij}x_j &= b_i, \quad i = 1, \dots, m_e \\ \sum_{j=1}^n a_{ij}x_j &\geq b_i, \quad i = m_e + 1, \dots, m \end{aligned}$$

The coefficients a_{ij} and right-hand sides b_i of the equality and inequality constraints are collected in the $m \times n$ matrix A and the m -vector b .

The m linear constraints define a feasible region \mathcal{G} in \mathcal{R}^n that must contain the point x^* that minimizes the problem. If the feasible region \mathcal{G} is empty, no solution to the optimization problem exists.

All optimization techniques in PROC NLP (except those processing nonlinear constraints) are *active set methods*. The iteration starts with a feasible point $x^{(0)}$, which either is provided by the user or can be computed by the Schittkowski and Stoer (1979) algorithm implemented in PROC NLP. The algorithm then moves from one feasible point $x^{(k-1)}$ to a better feasible point $x^{(k)}$ along a feasible search direction $s^{(k)}$:

$$x^{(k)} = x^{(k-1)} + \alpha^{(k)}s^{(k)}, \quad \alpha^{(k)} > 0$$

Theoretically, the path of points $x^{(k)}$ never leaves the feasible region \mathcal{G} of the optimization problem, but it can hit its boundaries. The active set $\mathcal{A}^{(k)}$ of point $x^{(k)}$ is defined as the index set of all linear equality constraints and those inequality constraints that are satisfied at $x^{(k)}$. If no constraint is active for $x^{(k)}$, the point is located in the interior of \mathcal{G} , and the active set $\mathcal{A}^{(k)}$ is empty. If the point $x^{(k)}$ in iteration k hits the boundary of inequality constraint i , this constraint i becomes active and is added to $\mathcal{A}^{(k)}$. Each equality or active inequality constraint reduces the dimension (degrees of freedom) of the optimization problem.

In practice, the active constraints can be satisfied only with finite precision. The `LCEPSILON=r` option specifies the range for active and violated linear constraints. If the point $x^{(k)}$ satisfies the condition

$$\left| \sum_{j=1}^n a_{ij}x_j^{(k)} - b_i \right| \leq t$$

where $t = r \times (|b_i| + 1)$, the constraint i is recognized as an active constraint. Otherwise, the constraint i is either an inactive inequality or a violated inequality or equality constraint. Due to rounding errors in computing the projected search direction, error can be accumulated so that an iterate $x^{(k)}$ steps out of the feasible region. In those cases, PROC NLP may try to pull the iterate $x^{(k)}$ into the feasible region. However, in some cases the algorithm needs to increase the feasible region by increasing the `LCEPSILON=r` value. If this happens it is indicated by a message displayed in the log output.

If you cannot expect an improvement in the value of the objective function by moving from an active constraint back into the interior of the feasible region, you use this inequality constraint as an equality constraint in the next iteration. That means the active set $\mathcal{A}^{(k+1)}$ still contains the constraint i . Otherwise you release the active inequality constraint and increase the dimension of the optimization problem in the next iteration.

A serious numerical problem can arise when some of the active constraints become (nearly) linearly dependent. Linearly dependent equality constraints are removed before entering the optimization. You can use the `LCSINGULAR=` option to specify a criterion r used in the update of the QR decomposition that decides whether an active constraint is linearly dependent relative to a set of other active constraints.

If the final parameter set x^* is subjected to n_{act} linear equality or active inequality constraints, the QR decomposition of the $n \times n_{act}$ matrix \hat{A}^T of the linear constraints is computed by $\hat{A}^T = QR$, where Q is an $n \times n$ orthogonal matrix and R is an $n \times n_{act}$ upper triangular matrix. The n columns of matrix Q can be separated into two matrices, $Q = [Y, Z]$, where Y contains the first n_{act} orthogonal columns of Q and Z contains the last $n - n_{act}$ orthogonal columns of Q . The $n \times (n - n_{act})$ column-orthogonal matrix Z is also called the nullspace matrix of the active linear constraints \hat{A}^T . The $n - n_{act}$ columns of the $n \times (n - n_{act})$ matrix Z form a basis orthogonal to the rows of the $n_{act} \times n$ matrix \hat{A} .

At the end of the iteration process, the PROC NLP can display the *projected gradient*

$$g_Z = Z^T g$$

In the case of boundary constrained optimization, the elements of the projected gradient correspond to the gradient elements of the free parameters. A necessary condition for x^* to be a local minimum of the optimization problem is

$$g_Z(x^*) = Z^T g(x^*) = 0$$

The symmetric $n_{act} \times n_{act}$ matrix

$$G_Z = Z^T G Z$$

is called a *projected Hessian matrix*. A second-order necessary condition for x^* to be a local minimizer requires that the projected Hessian matrix is positive semidefinite. If available, the projected gradient and projected Hessian matrix can be displayed and written in an `OUTEST=` data set.

Those elements of the n_{act} vector of first-order estimates of *Lagrange multipliers*

$$\lambda = (\hat{A}\hat{A}^T)^{-1} \hat{A} Z Z^T g$$

which correspond to active inequality constraints indicate whether an improvement of the objective function can be obtained by releasing this active constraint. For minimization (maximization), a significant negative (positive) Lagrange multiplier indicates that a possible reduction (increase) of the objective function can be obtained by releasing this active linear constraint. The `LCDEACT=r` option can be used to specify a threshold r for the Lagrange multiplier that decides whether an active inequality constraint remains active or can be deactivated. The Lagrange multipliers are displayed (and written in an `OUTEST=` data set) only if linear constraints are active at the solution x^* . (In the case of boundary-constrained optimization, the Lagrange multipliers for active lower (upper) constraints are the negative (positive) gradient elements corresponding to the active parameters.)

Feasible Starting Point

Two algorithms are used to obtain a feasible starting point.

- When only boundary constraints are specified:
 - If the parameter x_j , $1 \leq j \leq n$, violates a two-sided boundary constraint (or an equality constraint) $l_j \leq x_j \leq u_j$, the parameter is given a new value inside the feasible interval, as follows:

$$x_j = \begin{cases} l_j, & \text{if } u_j \leq l_j \\ l_j + \frac{1}{2}(u_j - l_j), & \text{if } u_j - l_j < 4 \\ l_j + \frac{1}{10}(u_j - l_j), & \text{if } u_j - l_j \geq 4 \end{cases}$$

- If the parameter x_j , $1 \leq j \leq n$, violates a one-sided boundary constraint $l_j \leq x_j$ or $x_j \leq u_j$, the parameter is given a new value near the violated boundary, as follows:

$$x_j = \begin{cases} l_j + \max(1, \frac{1}{10}l_j), & \text{if } x_j < l_j \\ u_j - \max(1, \frac{1}{10}u_j), & \text{if } x_j > u_j \end{cases}$$

- When general linear constraints are specified, the algorithm of Schittkowski and Stoer (1979) computes a feasible point, which may be quite far from a user-specified infeasible point.

Line-Search Methods

In each iteration k , the (dual) quasi-Newton, hybrid quasi-Newton, conjugate gradient, and Newton-Raphson minimization techniques use iterative line-search algorithms that try to optimize a linear, quadratic, or cubic approximation of f along a feasible descent search direction $s^{(k)}$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}, \quad \alpha^{(k)} > 0$$

by computing an approximately optimal scalar $\alpha^{(k)}$.

Therefore, a line-search algorithm is an iterative process that optimizes a nonlinear function $f = f(\alpha)$ of one parameter (α) within each iteration k of the optimization technique, which itself tries to optimize a linear or quadratic approximation of the nonlinear objective function $f = f(x)$ of n parameters x . Since the outside iteration process is based only on the approximation of the objective function, the inside iteration of the line-search algorithm does not have to be perfect. Usually, the choice of α significantly reduces (in a minimization) the objective function. Criteria often used for termination of line-search algorithms are the Goldstein conditions (refer to Fletcher (1987)).

Various line-search algorithms can be selected using the `LINESEARCH=` option. The line-search method `LINESEARCH=2` seems to be superior when function evaluation consumes significantly less computation time than gradient evaluation. Therefore, `LINESEARCH=2` is the default value for Newton-Raphson, (dual) quasi-Newton, and conjugate gradient optimizations.

A special default line-search algorithm for `TECH=HYQUAN` is useful only for least squares problems and cannot be chosen by the `LINESEARCH=` option. This method uses three columns of the $m \times n$ Jacobian matrix, which for large m can require more memory than using the algorithms designated by `LINESEARCH=1` through `LINESEARCH=8`.

The line-search methods `LINESEARCH=2` and `LINESEARCH=3` can be modified to exact line search by using the `LSPRECISION=` option (specifying the σ parameter in Fletcher (1987)). The line-search methods `LINESEARCH=1`, `LINESEARCH=2`, and `LINESEARCH=3` satisfy the left-hand-side and right-hand-side Goldstein conditions (refer to Fletcher (1987)). When derivatives are available, the line-search methods `LINESEARCH=6`, `LINESEARCH=7`, and `LINESEARCH=8` try to satisfy the right-hand-side Goldstein condition; if derivatives are not available, these line-search algorithms use only function calls.

Restricting the Step Length

Almost all line-search algorithms use iterative extrapolation techniques which can easily lead them to (feasible) points where the objective function f is no longer defined. (e.g., resulting in indefinite matrices for ML estimation) or difficult to compute (e.g., resulting in floating point overflows). Therefore, PROC NLP provides options restricting the step length α or trust region radius Δ , especially during the first main iterations.

The inner product $g^T s$ of the gradient g and the search direction s is the slope of $f(\alpha) = f(x + \alpha s)$ along the search direction s . The default starting value $\alpha^{(0)} = \alpha^{(k,0)}$ in each line-search algorithm ($\min_{\alpha > 0} f(x + \alpha s)$) during the main iteration k is computed in three steps:

1. The first step uses either the difference $df = |f^{(k)} - f^{(k-1)}|$ of the function values during the last two consecutive iterations or the final step length value α^- of the last iteration $k - 1$ to compute a first value of $\alpha_1^{(0)}$.

- Not using the `DAMPSTEP=r` option:

$$\alpha_1^{(0)} = \begin{cases} step, & \text{if } 0.1 \leq step \leq 10 \\ 10, & \text{if } step > 10 \\ 0.1, & \text{if } step < 0.1 \end{cases}$$

with

$$step = \begin{cases} df/|g^T s|, & \text{if } |g^T s| \geq \epsilon \max(100df, 1) \\ 1, & \text{otherwise} \end{cases}$$

This value of $\alpha_1^{(0)}$ can be too large and lead to a difficult or impossible function evaluation, especially for highly nonlinear functions such as the EXP function.

- Using the `DAMPSTEP=r` option:

$$\alpha_1^{(0)} = \min(1, r\alpha^-)$$

The initial value for the new step length can be no larger than r times the final step length α^- of the previous iteration. The default value is $r = 2$.

2. During the first five iterations, the second step enables you to reduce $\alpha_1^{(0)}$ to a smaller starting value $\alpha_2^{(0)}$ using the `INSTEP=r` option:

$$\alpha_2^{(0)} = \min(\alpha_1^{(0)}, r)$$

After more than five iterations, $\alpha_2^{(0)}$ is set to $\alpha_1^{(0)}$.

3. The third step can further reduce the step length by

$$\alpha_3^{(0)} = \min(\alpha_2^{(0)}, \min(10, u))$$

where u is the maximum length of a step inside the feasible region.

The `INSTEP=r` option lets you specify a smaller or larger radius Δ of the trust region used in the first iteration of the trust region, double dogleg, and Levenberg-Marquardt algorithms. The default initial trust region radius $\Delta^{(0)}$ is the length of the scaled gradient (Moré 1978). This step corresponds to the default radius factor of $r = 1$. In most practical applications of the TRUREG, DBLDOG, and LEVMAR algorithms, this choice is successful. However, for bad initial values and highly nonlinear objective functions (such as the EXP function), the default start radius can result in arithmetic overflows. If this happens, you may try decreasing values of `INSTEP=r`, $0 < r < 1$, until the iteration starts successfully. A small factor r also affects the trust region radius $\Delta^{(k+1)}$ of the next steps because the radius is changed in each iteration by a factor $0 < c \leq 4$, depending on the ratio ρ expressing the goodness of quadratic function approximation. Reducing the radius Δ corresponds to increasing the ridge parameter λ , producing smaller steps directed more closely toward the (negative) gradient direction.

Computational Problems

First Iteration Overflows

If you use bad initial values for the parameters, the computation of the value of the objective function (and its derivatives) can lead to arithmetic overflows in the first iteration. The line-search algorithms that work with cubic extrapolation are especially sensitive to arithmetic overflows. If an overflow occurs with an optimization technique that uses line search, you can use the `INSTEP=` option to reduce the length of the first trial step during the line search of the first five iterations or use the `DAMPSTEP` or `MAXSTEP=` option to restrict the step length of the initial α in subsequent iterations. If an arithmetic overflow occurs in the first iteration of the trust region, double dogleg, or Levenberg-Marquardt algorithm, you can use the `INSTEP=` option to reduce the default trust region radius of the first iteration. You can also change the minimization technique or the line-search method. If none of these methods helps, consider the following actions:

- scale the parameters
- provide better initial values
- use boundary constraints to avoid the region where overflows may happen
- change the algorithm (specified in program statements) which computes the objective function

Problems in Evaluating the Objective Function

The starting point $x^{(0)}$ must be a point that can be evaluated by all the functions involved in your problem. However, during optimization the optimizer may iterate to a point $x^{(k)}$ where the objective function or nonlinear constraint functions and their derivatives cannot be evaluated. If you can identify the problematic region, you can prevent the algorithm from reaching it by adding another constraint to the problem. Another possibility is a modification of the objective function that will produce a large, undesired function value. As a result, the optimization algorithm reduces the step length and stays closer to the point that has been evaluated successfully in the previous iteration. For more information, refer to the section “Missing Values in Program Statements” on page 626.

Problems with Quasi-Newton Methods for Nonlinear Constraints

The sequential quadratic programming algorithm in QUANEW, which is used for solving nonlinearly constrained problems, can have problems updating the Lagrange multiplier vector μ . This usually results in very high values of the Lagrangian function and in *watchdog* restarts indicated in the iteration history. If this happens, there are three actions you can try:

- By default, the Lagrange vector μ is evaluated in the same way as Powell (1982b) describes. This corresponds to `VERSION=2`. By specifying `VERSION=1`, a modification of this algorithm replaces the update of the Lagrange vector μ with the original update of Powell (1978a, b), which is used in VF02AD.
- You can use the `INSTEP=` option to impose an upper bound for the step length α during the first five iterations.
- You can use the `INHESIAN=` option to specify a different starting approximation for the Hessian. Choosing only the `INHESIAN` option will use the Cholesky factor of a (possibly ridged) finite-difference approximation of the Hessian to initialize the quasi-Newton update process.

Other Convergence Difficulties

There are a number of things to try if the optimizer fails to converge.

- Check the derivative specification:
If derivatives are specified by using the `GRADIENT`, `HESSIAN`, `JACOBIAN`, `CRPJAC`, or `JACNLC` statement, you can compare the specified derivatives with those computed by finite-difference approximations (specifying the `FD` and `FDHESSIAN` option). Use the `GRADCHECK` option to check if the gradient g is correct. For more information, refer to the section “Testing the Gradient Specification” on page 604.
- Forward-difference derivatives specified with the `FD=` or `FDHESSIAN=` option may not be precise enough to satisfy strong gradient termination criteria. You may need to specify the more expensive central-difference formulas or use analytical derivatives. The finite-difference intervals may be too small or too big and the finite-difference derivatives may be erroneous. You can specify the `FDINT=` option to compute better finite-difference intervals.

- Change the optimization technique:
For example, if you use the default `TECH=LEV MAR`, you can
 - change to `TECH=QUANEW` or to `TECH=NRRIDG`
 - run some iterations with `TECH=CONGRA`, write the results in an `OUTEST=` data set, and use them as initial values specified by an `INEST=` data set in a second run with a different `TECH=` technique
- Change or modify the update technique and the line-search algorithm:
This method applies only to `TECH=QUANEW`, `TECH=HYQUAN`, or `TECH=CONGRA`. For example, if you use the default update formula and the default line-search algorithm, you can
 - change the update formula with the `UPDATE=` option
 - change the line-search algorithm with the `LINESEARCH=` option
 - specify a more precise line search with the `LSPRECISION=` option, if you use `LINESEARCH=2` or `LINESEARCH=3`
- Change the initial values by using a grid search specification to obtain a set of good feasible starting values.

Convergence to Stationary Point

The (projected) gradient at a stationary point is zero and that results in a zero step length. The stopping criteria are satisfied.

There are two ways to avoid this situation:

- Use the `DECVAR` statement to specify a grid of feasible starting points.
- Use the `OPTCHECK=` option to avoid terminating at the stationary point.

The signs of the eigenvalues of the (reduced) Hessian matrix contain information regarding a stationary point:

- If all eigenvalues are positive, the Hessian matrix is positive definite and the point is a minimum point.
- If some of the eigenvalues are positive and all remaining eigenvalues are zero, the Hessian matrix is positive semidefinite and the point is a minimum or saddle point.
- If all eigenvalues are negative, the Hessian matrix is negative definite and the point is a maximum point.
- If some of the eigenvalues are negative and all remaining eigenvalues are zero, the Hessian matrix is negative semidefinite and the point is a maximum or saddle point.
- If all eigenvalues are zero, the point can be a minimum, maximum, or saddle point.

Precision of Solution

In some applications, PROC NLP may result in parameter estimates that are not precise enough. Usually this means that the procedure terminated too early at a point too far from the optimal point. The termination criteria define the size of the termination region around the optimal point. Any point inside this region can be accepted for terminating the optimization process. The default values of the termination criteria are set to satisfy a reasonable compromise between the computational effort (computer time) and the precision of the computed estimates for the most common applications. However, there are a number of circumstances where the default values of the termination criteria specify a region that is either too large or too small. If the termination region is too large, it can contain points with low precision. In such cases, you should inspect the log or list output to find the message stating which termination criterion terminated the optimization process. In many applications, you can obtain a solution with higher precision by simply using the old parameter estimates as starting values in a subsequent run where you specify a smaller value for the termination criterion that was satisfied at the previous run.

If the termination region is too small, the optimization process may take longer to find a point inside such a region or may not even find such a point due to rounding errors in function values and derivatives. This can easily happen in applications where finite-difference approximations of derivatives are used and the **GCONV** and **ABSGCONV** termination criteria are too small to respect rounding errors in the gradient values.

Covariance Matrix

The **COV=** option must be specified to compute an approximate covariance matrix for the parameter estimates under asymptotic theory for least squares, maximum-likelihood, or Bayesian estimation, with or without corrections for degrees of freedom as specified by the **VARDEF=** option.

Two groups of six different forms of covariance matrices (and therefore approximate standard errors) can be computed corresponding to the following two situations:

- The **LSQ** statement is specified, which means that least squares estimates are being computed:

$$\min f(x) = \sum_{i=1}^m f_i^2(x)$$

- The **MIN** or **MAX** statement is specified, which means that maximum-likelihood or Bayesian estimates are being computed:

$$\text{opt } f(x) = \sum_{i=1}^m f_i(x)$$

where opt is either min or max.

In either case, the following matrices are used:

$$G = \nabla^2 f(x)$$

$$J(f) = (\nabla f_1, \dots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

$$JJ(f) = J(f)^T J(f)$$

$$V = J(f)^T \text{diag}(f_i^2) J(f)$$

$$W = J(f)^T \text{diag}(f_i^\dagger) J(f)$$

where

$$f_i^\dagger = \begin{cases} 0, & \text{if } f_i = 0 \\ 1/f_i, & \text{otherwise} \end{cases}$$

For unconstrained minimization, or when none of the final parameter estimates are subjected to linear equality or active inequality constraints, the formulas of the six types of covariance matrices are as follows:

Table 6.3 Central-Difference Approximations

	COV	MIN or MAX Statement	LSQ Statement
1	M	$(_NOBS_/d)G^{-1}JJ(f)G^{-1}$	$(_NOBS_/d)G^{-1}VG^{-1}$
2	H	$(_NOBS_/d)G^{-1}$	σ^2G^{-1}
3	J	$(1/d)W^{-1}$	$\sigma^2JJ(f)^{-1}$
4	B	$(1/d)G^{-1}WG^{-1}$	$\sigma^2G^{-1}JJ(f)G^{-1}$
5	E	$(_NOBS_/d)JJ(f)^{-1}$	$(1/d)V^{-1}$
6	U	$(_NOBS_/d)W^{-1}JJ(f)W^{-1}$	$(_NOBS_/d)JJ(f)^{-1}VJJ(f)^{-1}$

The value of d depends on the `VARDEF=` option and on the value of the `_NOBS_` variable:

$$d = \begin{cases} \max(1, _NOBS_ - _DF_), & \text{for VARDEF=DF} \\ _NOBS_, & \text{for VARDEF=N} \end{cases}$$

where `_DF_` is either set in the program statements or set by default to n (the number of parameters) and `_NOBS_` is either set in the program statements or set by default to $nobs \times mfun$, where $nobs$ is the number of observations in the data set and $mfun$ is the number of functions listed in the `LSQ`, `MIN`, or `MAX` statement.

The value σ^2 depends on the specification of the `SIGSQ=` option and on the value of d :

$$\sigma^2 = \begin{cases} sq \times _NOBS_/d, & \text{if SIGSQ=sq is specified} \\ 2f(x^*)/d, & \text{if SIGSQ= is not specified} \end{cases}$$

where $f(x^*)$ is the value of the objective function at the optimal parameter estimates x^* .

The two groups of formulas distinguish between two situations:

- For least squares estimates, the error variance can be estimated from the objective function value and is used in three of the six different forms of covariance matrices. If you have an independent estimate of the error variance, you can specify it with the `SIGSQ=` option.
- For maximum-likelihood or Bayesian estimates, the objective function should be the logarithm of the likelihood or of the posterior density when using the `MAX` statement.

For minimization, the inversion of the matrices in these formulas is done so that negative eigenvalues are considered zero, resulting always in a positive semidefinite covariance matrix.

In small samples, estimates of the covariance matrix based on asymptotic theory are often too small and should be used with caution.

If the final parameter estimates are subjected to $n_{act} > 0$ linear equality or active linear inequality constraints, the formulas of the covariance matrices are modified similar to Gallant (1987) and Cramer (1986, p. 38) and additionally generalized for applications with singular matrices. In the constrained case, the value of d used in the scalar factor σ^2 is defined by

$$d = \begin{cases} \max(1, _NOBS_ - _DF_ + n_{act}), & \text{for VARDEF=DF} \\ _NOBS_, & \text{for VARDEF=N} \end{cases}$$

where n_{act} is the number of active constraints and $_NOBS_$ is set as in the unconstrained case.

For minimization, the covariance matrix should be positive definite; for maximization it should be negative definite. There are several options available to check for a rank deficiency of the covariance matrix:

- The **ASINGULAR=**, **MSINGULAR=**, and **VSINGULAR=** options can be used to set three singularity criteria for the inversion of the matrix A needed to compute the covariance matrix, when A is either the Hessian or one of the crossproduct Jacobian matrices. The singularity criterion used for the inversion is

$$|d_{j,j}| \leq \max(\text{ASING}, \text{VSING} \times |A_{j,j}|, \text{MSING} \times \max(|A_{1,1}|, \dots, |A_{n,n}|))$$

where $d_{j,j}$ is the diagonal pivot of the matrix A , and **ASING**, **VSING** and **MSING** are the specified values of the **ASINGULAR=**, **VSINGULAR=**, and **MSINGULAR=** options. The default values are

- **ASING**: the square root of the smallest positive double precision value
- **MSING**: $1\text{E}-12$ if the **SINGULAR=** option is not specified and $\max(10 \times \epsilon, 1\text{E} - 4 \times \text{SINGULAR})$ otherwise, where ϵ is the machine precision
- **VSING**: $1\text{E}-8$ if the **SINGULAR=** option is not specified and the value of **SINGULAR** otherwise

NOTE: In many cases, a normalized matrix $D^{-1}AD^{-1}$ is decomposed and the singularity criteria are modified correspondingly.

- If the matrix A is found singular in the first step, a generalized inverse is computed. Depending on the **G4=** option, a generalized inverse is computed that satisfies either all four or only two Moore-Penrose conditions. If the number of parameters n of the application is less than or equal to **G4=i**, a **G4** inverse is computed; otherwise only a **G2** inverse is computed. The **G4** inverse is computed by (the computationally very expensive but numerically stable) eigenvalue decomposition; the **G2** inverse is computed by Gauss transformation. The **G4** inverse is computed using the eigenvalue decomposition $A = Z\Lambda Z^T$, where Z is the orthogonal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues, $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$. If the **PEIGVAL** option is specified, the eigenvalues λ_i are displayed. The **G4** inverse of A is set to

$$A^- = Z\Lambda^-Z^T$$

where the diagonal matrix $\Lambda^- = \text{diag}(\lambda_1^-, \dots, \lambda_n^-)$ is defined using the **COVSING=** option:

$$\lambda_i^- = \begin{cases} 1/\lambda_i, & \text{if } |\lambda_i| > \text{COVSING} \\ 0, & \text{if } |\lambda_i| \leq \text{COVSING} \end{cases}$$

If the **COVSING=** option is not specified, the nr smallest eigenvalues are set to zero, where nr is the number of rank deficiencies found in the first step.

For optimization techniques that do not use second-order derivatives, the covariance matrix is usually computed using finite-difference approximations of the derivatives. By specifying **TECH=NONE**, any of the covariance matrices can be computed using analytical derivatives. The covariance matrix specified by the **COV=** option can be displayed (using the **PCOV** option) and is written to the **OUTEST=** data set.

Input and Output Data Sets

DATA= Input Data Set

The **DATA=** data set is used only to specify an objective function f that is a combination of m other functions f_i . For each function f_i , $i = 1, \dots, m$, listed in a **MAX**, **MIN**, or **LSQ** statement, each observation l , $l = 1, \dots, nobs$, in the **DATA=** data set defines a specific function f_{il} that is evaluated by substituting the values of the variables of this observation into the program statements. If the **MAX** or **MIN** statement is used, the $m \times nobs$ specific functions f_{il} are added to a single objective function f . If the **LSQ** statement is used, the sum-of-squares f of the $m \times nobs$ specific functions f_{il} is minimized. The **NOMISS** option causes observations with missing values to be skipped.

INEST= Input Data Set

The **INEST=** (or **INVAR=**, or **ESTDATA=**) input data set can be used to specify the initial values of the parameters defined in a **DECVAR** statement as well as boundary constraints and the more general linear constraints which could be imposed on these parameters. This form of input is similar to the dense format input used in PROC LP.

The variables of the **INEST=** data set are

- a character variable **_TYPE_** that indicates the type of the observation
- n numeric variables with the parameter names used in the **DECVAR** statement
- the **BY** variables that are used in a **DATA=** input data set
- a numeric variable **_RHS_** specifying the right-hand-side constants (needed only if linear constraints are used)
- additional variables with names corresponding to constants used in the program statements

The content of the **_TYPE_** variable defines the meaning of the observation of the **INEST=** data set. PROC NLP recognizes the following **_TYPE_** values:

- **PARMS**, which specifies initial values for parameters. Additional variables can contain the values of constants that are referred to in program statements. The values of the constants in the **PARMS** observation initialize the constants in the program statements.
- **UPPERBD | UB**, which specifies upper bounds. A missing value indicates that no upper bound is specified for the parameter.
- **LOWERBD | LB**, which specifies lower bounds. A missing value indicates that no lower bound is specified for the parameter.

- LE | <= | <, which specifies linear constraint $\sum_j a_{ij} x_j \leq b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.
- GE | >= | >, which specifies linear constraint $\sum_j a_{ij} x_j \geq b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.
- EQ | =, which specifies linear constraint $\sum_j a_{ij} x_j = b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.

The constraints specified in an `INEST=` data set are added to the constraints specified in the `BOUNDS` and `LINCON` statements. You can use an `OUTEST=` data set as an `INEST=` data set in a subsequent run of PROC NLP. However, be aware that the `OUTEST=` data set also contains the boundary and general linear constraints specified in the previous run of PROC NLP. When you are using this `OUTEST=` data set without changes as an `INEST=` data set, PROC NLP adds the constraints from the data set to the constraints specified by a `BOUNDS` and `LINCON` statement. Although PROC NLP automatically eliminates multiple identical constraints you should avoid specifying the same constraint twice.

INQUAD= Input Data Set

Two types of `INQUAD=` data sets can be used to specify the objective function of a quadratic programming problem for `TECH=QUADAS` or `TECH=LICOMP`,

$$f(x) = \frac{1}{2}x^T Gx + g^T x + c, \quad \text{with } G^T = G$$

The *dense* `INQUAD=` data set must contain all numerical values of the symmetric matrix G , the vector g , and the scalar c . Using the *sparse* `INQUAD=` data set enables you to specify only the nonzero positions in matrix G and vector g . Those locations that are not set by the *sparse* `INQUAD=` data set are assumed to be zero.

Dense INQUAD= Data Set

A dense `INQUAD=` data set must contain two character variables, `_TYPE_` and `_NAME_`, and at least n numeric variables whose names are the parameter names. The `_TYPE_` variable takes the following values:

- QUAD lists the n values of the row of the G matrix that is defined by the parameter name used in the `_NAME_` variable.
- LINEAR lists the n values of the g vector.
- CONST sets the value of the scalar c and cannot contain different numerical values; however, it could contain up to $n - 1$ missing values.
- PARMS specifies initial values for parameters.
- UPPERBD | UB specifies upper bounds. A missing value indicates that no upper bound is specified.
- LOWERBD | LB specifies lower bounds. A missing value indicates that no lower bound is specified.
- LE | <= | < specifies linear constraint $\sum_j a_{ij} x_j \leq b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.
- GE | >= | > specifies linear constraint $\sum_j a_{ij} x_j \geq b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.

- `EQ | =` specifies linear constraint $\sum_j a_{ij}x_j = b_i$. The n parameter values contain the coefficients a_{ij} , and the `_RHS_` variable contains the right-hand side b_i . Missing values indicate zeros.

Constraints specified in a dense `INQUAD=` data set are added to the constraints specified in `BOUNDS` and `LINCON` statements.

Sparse INQUAD= Data Set

A sparse `INQUAD=` data set must contain three character variables `_TYPE_`, `_ROW_`, and `_COL_`, and one numeric variable `_VALUE_`. The `_TYPE_` variable can assume two values:

- `QUAD` specifies that the `_ROW_` and `_COL_` variables define the row and column locations of the values in the G matrix.
- `LINEAR` specifies that the `_ROW_` variable defines the row locations of the values in the g vector. The `_COL_` variable is not used.

Using both the `MODEL=` option and the `INCLUDE` statement with the same model file will include the file twice (erroneous in most cases).

OUT= Output Data Set

The `OUT=` data set contains those variables of a `DATA=` input data set that are referred to in the program statements and additional variables computed by the program statements for the objective function. Specifying the `NOMISS` option enables you to skip observations with missing values in variables used in the program statements. The `OUT=` data set can also contain first- and second-order derivatives of these variables if the `OUTDER=` option is specified. The variables and derivatives are the final parameter estimates x^* or (for `TECH=NONE`) the initial value x^0 .

The variables of the `OUT=` data set are

- the `BY` variables and all other variables that are used in a `DATA=` input data set and referred to in the program code
- a variable `_OBS_` containing the number of observations read from a `DATA=` input data set, where the counting is restarted with the start of each `BY` group. If there is no `DATA=` input data set, then `_OBS_=1`.
- a character variable `_TYPE_` naming the type of the observation
- the parameter variables listed in the `DECVAR` statement
- the function variables listed in the `MIN`, `MAX`, or `LSQ` statement
- all other variables computed in the program statements
- the character variable `_WRT_` (if `OUTDER=1`) containing the *with respect to* variable for which the first-order derivatives are written in the function variables
- the two character variables `_WRT1_` and `_WRT2_` (if `OUTDER=2`) containing the two *with respect to* variables for which the first- and second-order derivatives are written in the function variables

OUTEST= Output Data Set

The `OUTEST=` or `OUTVAR=` output data set saves the optimization solution of PROC NLP. You can use the `OUTEST=` or `OUTVAR=` data set as follows:

- to save the values of the objective function on grid points to examine, for example, surface plots using PROC G3D (use the `OUTGRID` option)
- to avoid any costly computation of analytical (first- or second-order) derivatives during optimization when they are needed only upon termination. In this case a two-step approach is recommended:
 1. In a first execution, the optimization is done; that is, optimal parameter estimates are computed, and the results are saved in an `OUTEST=` data set.
 2. In a subsequent execution, the optimal parameter estimates in the previous `OUTEST=` data set are read in an `INEST=` data set and used with `TECH=NONE` to compute further results, such as analytical second-order derivatives or some kind of covariance matrix.
- to restart the procedure using parameter estimates as initial values
- to split a time-consuming optimization problem into a series of smaller problems using intermediate results as initial values in subsequent runs. (Refer to the `MAXTIME=`, `MAXIT=`, and `MAXFUNC=` options to trigger stopping.)
- to write the value of the objective function, the parameter estimates, the time in seconds starting at the beginning of the optimization process and (if available) the gradient to the `OUTEST=` data set during the iterations. After the PROC NLP run is completed, the convergence progress can be inspected by graphically displaying the iterative information. (Refer to the `OUTITER` option.)

The variables of the `OUTEST=` data set are

- the BY variables that are used in a `DATA=` input data set
- a character variable `_TECH_` naming the optimization technique used
- a character variable `_TYPE_` specifying the type of the observation
- a character variable `_NAME_` naming the observation. For a linear constraint, the `_NAME_` variable indicates whether the constraint is active at the solution. For the initial observations, the `_NAME_` variable indicates if the number in the `_RHS_` variable corresponds to the number of positive, negative, or zero eigenvalues.
- n numeric variables with the parameter names used in the `DECVAR` statement. These variables contain a point x of the parameter space, lower or upper bound constraints, or the coefficients of linear constraints.
- a numeric variable `_RHS_` (right-hand side) that is used for the right-hand-side value b_i of a linear constraint or for the value $f = f(x)$ of the objective function at a point x of the parameter space
- a numeric variable `_ITER_` that is zero for initial values, equal to the iteration number for the `OUTITER` output, and missing for the result output

The `_TYPE_` variable identifies how to interpret the observation. If `_TYPE_` is

- `PARMS` then parameter-named variables contain the coordinates of the resulting point x^* . The `_RHS_` variable contains $f(x^*)$.
- `INITIAL` then parameter-named variables contain the feasible starting point $x^{(0)}$. The `_RHS_` variable contains $f(x^{(0)})$.
- `GRIDPNT` then (if the `OUTGRID` option is specified) parameter-named variables contain the coordinates of any point $x^{(k)}$ used in the grid search. The `_RHS_` variable contains $f(x^{(k)})$.
- `GRAD` then parameter-named variables contain the gradient at the initial or final estimates.
- `STDERR` then parameter-named variables contain the approximate standard errors (square roots of the diagonal elements of the covariance matrix) if the `COV=` option is specified.
- `_NOBS_` then (if the `COV=` option is specified) all parameter variables contain the value of `_NOBS_` used in computing the σ^2 value in the formula of the covariance matrix.
- `UPPERBD | UB` then (if there are boundary constraints) the parameter variables contain the upper bounds.
- `LOWERBD | LB` then (if there are boundary constraints) the parameter variables contain the lower bounds.
- `NACTBC` then all parameter variables contain the number n_{abc} of active boundary constraints at the solution x^* .
- `ACTBC` then (if there are active boundary constraints) the observation indicate which parameters are actively constrained, as follows:
 - `_NAME_=GE` the active lower bounds
 - `_NAME_=LE` the active upper bounds
 - `_NAME_=EQ` the active equality constraints
- `NACTLC` then all parameter variables contain the number n_{alc} of active linear constraints that are recognized as linearly independent.
- `NLDACTLC` then all parameter variables contain the number of active linear constraints that are recognized as linearly dependent.
- `LE` then (if there are linear constraints) the observation contains the i th linear constraint $\sum_j a_{ij}x_j \leq b_i$. The parameter variables contain the coefficients a_{ij} , $j = 1, \dots, n$, and the `_RHS_` variable contains b_i . If the constraint i is active at the solution x^* , then `_NAME_=ACTLC` or `_NAME_=LDACTLC`.
- `GE` then (if there are linear constraints) the observation contains the i th linear constraint $\sum_j a_{ij}x_j \geq b_i$. The parameter variables contain the coefficients a_{ij} , $j = 1, \dots, n$, and the `_RHS_` variable contains b_i . If the constraint i is active at the solution x^* , then `_NAME_=ACTLC` or `_NAME_=LDACTLC`.
- `EQ` then (if there are linear constraints) the observation contains the i th linear constraint $\sum_j a_{ij}x_j = b_i$. The parameter variables contain the coefficients a_{ij} , $j = 1, \dots, n$, the `_RHS_` variable contains b_i , and `_NAME_=ACTLC` or `_NAME_=LDACTLC`.

- LAGRANGE then (if at least one of the linear constraints is an equality constraint or an active inequality constraint) the observation contains the vector of Lagrange multipliers. The Lagrange multipliers of active boundary constraints are listed first followed by those of active linear constraints and those of active nonlinear constraints. Lagrange multipliers are available only for the set of linearly independent active constraints.
- PROJGRAD then (if there are linear constraints) the observation contains the $n - n_{act}$ values of the projected gradient $g_Z = Z^T g$ in the variables corresponding to the first $n - n_{act}$ parameters.
- JACOBIAN then (if the PJACOBI or OUTJAC option is specified) the m observations contain the m rows of the $m \times n$ Jacobian matrix. The _RHS_ variable contains the row number l , $l = 1, \dots, m$.
- HESSIAN then the first n observations contain the n rows of the (symmetric) Hessian matrix. The _RHS_ variable contains the row number j , $j = 1, \dots, n$, and the _NAME_ variable contains the corresponding parameter name.
- PROJHESS then the first $n - n_{act}$ observations contain the $n - n_{act}$ rows of the projected Hessian matrix $Z^T G Z$. The _RHS_ variable contains the row number j , $j = 1, \dots, n - n_{act}$, and the _NAME_ variable is blank.
- CRPJAC then the first n observations contain the n rows of the (symmetric) crossproduct Jacobian matrix at the solution. The _RHS_ variable contains the row number j , $j = 1, \dots, n$, and the _NAME_ variable contains the corresponding parameter name.
- PROJCRPJ then the first $n - n_{act}$ observations contain the $n - n_{act}$ rows of the projected crossproduct Jacobian matrix $Z^T (J^T J) Z$. The _RHS_ variable contains the row number j , $j = 1, \dots, n - n_{act}$, and the _NAME_ variable is blank.
- COV1, COV2, COV3, COV4, COV5, or COV6 then (depending on the COV= option) the first n observations contain the n rows of the (symmetric) covariance matrix of the parameter estimates. The _RHS_ variable contains the row number j , $j = 1, \dots, n$, and the _NAME_ variable contains the corresponding parameter name.
- DETERMIN contains the determinant $det = a \times 10^b$ of the matrix specified by the value of the _NAME_ variable where a is the value of the first variable in the DECVAR statement and b is in _RHS_.
- NEIGPOS, NEIGNEG, or NEIGZER then the _RHS_ variable contains the number of positive, negative, or zero eigenvalues of the matrix specified by the value of the _NAME_ variable.
- COVRANK then the _RHS_ variable contains the rank of the covariance matrix.
- SIGSQ then the _RHS_ variable contains the scalar factor of the covariance matrix.
- _TIME_ then (if the OUTITER option is specified) the _RHS_ variable contains the number of seconds passed since the start of the optimization.
- TERMINAT then if optimization terminated at a point satisfying one of the termination criteria, an abbreviation of the corresponding criteria is given to the _NAME_ variable. Otherwise _NAME_=PROBLEMS.

If for some reason the procedure does not terminate successfully (for example, no feasible initial values can be computed or the function value or derivatives at the starting point cannot be computed), the `OUTEST=` data set may contain only part of the observations (usually only the `PARMS` and `GRAD` observation).

NOTE: Generally you can use an `OUTEST=` data set as an `INEST=` data set in a further run of PROC NLP. However, be aware that the `OUTEST=` data set also contains the boundary and general linear constraints specified in the previous run of PROC NLP. When you are using this `OUTEST=` data set without changes as an `INEST=` data set, PROC NLP adds the constraints from the data set to the constraints specified by a `BOUNDS` or `LINCON` statement. Although PROC NLP automatically eliminates multiple identical constraints you should avoid specifying the same constraint twice.

Output of Profiles

The following observations are written to the `OUTEST=` data set only when the `PROFILE` statement or `CLPARM` option is specified.

Table 6.4 Output of Profiles

<code>_TYPE_</code>	<code>_NAME_</code>	<code>_RHS_</code>	Meaning of Observation
<code>PLC_LOW</code>	<code>parname</code>	y value	coordinates of lower CL for α
<code>PLC_UPP</code>	<code>parname</code>	y value	coordinates of upper CL for α
<code>WALD_CL</code>	<code>LOWER</code>	y value	lower Wald CL for α in <code>_ALPHA_</code>
<code>WALD_CL</code>	<code>UPPER</code>	y value	upper Wald CL for α in <code>_ALPHA_</code>
<code>PL_CL</code>	<code>LOWER</code>	y value	lower PL CL for α in <code>_ALPHA_</code>
<code>PL_CL</code>	<code>UPPER</code>	y value	upper PL CL for α in <code>_ALPHA_</code>
<code>PROFILE</code>	<code>L(THETA)</code>	missing	y value corresponding to x in following <code>_NAME_=THETA</code>
<code>PROFILE</code>	<code>THETA</code>	missing	x value corresponding to y in previous <code>_NAME_=L(THETA)</code>

Assume that the `PROFILE` statement specifies n_p parameters and n_α confidence levels. For `CLPARM`, $n_p = n$ and $n_\alpha = 4$.

- `_TYPE_=PLC_LOW` and `_TYPE_=PLC_UPP`:

If the `CLPARM=` option or the `PROFILE` statement with the `OUTTABLE` option is specified, then the complete set θ of parameter estimates (rather than only the confidence limit $x = \theta_j$) is written to the `OUTEST=` data set for each side of the confidence interval. This output may be helpful for further analyses on how small changes in $x = \theta_j$ affect the changes in the other $\theta_i, i \neq j$. The `_ALPHA_` variable contains the corresponding value of α . There should be no more than $2n_\alpha n_p$ observations. If the confidence limit cannot be computed, the corresponding observation is not available.

- `_TYPE_=WALD_CL`:

If `CLPARM=WALD`, `CLPARM=BOTH`, or the `PROFILE` statement with α values is specified, then the Wald confidence limits are written to the `OUTEST=` data set for each of the default or specified values of α . The `_ALPHA_` variable contains the corresponding value of α . There should be $2n_\alpha$ observations.

- **_TYPE_=PL_CL:**
If **CLPARM=PL**, **CLPARM=BOTH**, or the **PROFILE** statement with α values is specified, then the PL confidence limits are written to the **OUTEST=** data set for each of the default or specified values of α . The **_ALPHA_** variable contains the corresponding values of α . There should be $2n_\alpha$ observations; some observations may have missing values.
- **_TYPE_=PROFILE:**
If **CLPARM=PL**, **CLPARM=BOTH**, or the **CLPARM=** statement with or without α values is specified, then a set of (x, y) point coordinates in two adjacent observations with **_NAME_=L(THETA)** (y value) and **_NAME_=THETA** (x value) is written to the **OUTEST=** data set. The **_RHS_** and **_ALPHA_** variables are not used (are set to missing). The number of observations depends on the difficulty of the optimization problems.

OUTMODEL= Output Data Set

The program statements for objective functions, nonlinear constraints, and derivatives can be saved into an **OUTMODEL=** output data set. This data set can be used in an **INCLUDE** program statement or as a **MODEL=** input data set in subsequent calls of PROC NLP. The **OUTMODEL=** option is similar to the option used in PROC MODEL in SAS/ETS software.

Storing Programs in Model Files

Models can be saved to and recalled from SAS catalog files. SAS catalogs are special files which can store many kinds of data structures as separate units in one SAS file. Each separate unit is called an entry, and each entry has an entry type that identifies its structure to the SAS system.

In general, to save a model, use the **OUTMODEL=name** option in the PROC NLP statement, where *name* is specified as *libref.catalog.entry*, *libref.entry*, or *entry*. The *libref*, *catalog*, and *entry* names must be valid SAS names no more than 8 characters long. The *catalog* name is restricted to 7 characters on the CMS operating system. If not given, the *catalog* name defaults to MODELS, and the *libref* defaults to WORK. The entry type is always MODEL. Thus, **OUTMODEL=X** writes the model to the file WORK.MODELS.X.MODEL.

The **MODEL=** option is used to read in a model. A list of model files can be specified in the **MODEL=** option, and a range of names with numeric suffixes can be given, as in **MODEL=(MODEL1-MODEL10)**. When more than one model file is given, the list must be placed in parentheses, as in **MODEL=(A B C)**. If more than one model file is specified, the files are combined in the order listed in the **MODEL=** option.

When the **MODEL=** option is specified in the PROC NLP statement and model definition statements are also given later in the PROC NLP step, the model files are read in first, in the order listed, and the model program specified in the PROC NLP step is appended after the model program read from the **MODEL=** files.

The **INCLUDE** statement can be used to append model code to the current model code. The contents of the model files are inserted into the current model at the position where the **INCLUDE** statement appears.

Note that the following statements are not part of the program code that is written to an **OUTMODEL=** data set: **MIN**, **MAX**, **LSQ**, **MINQUAD**, **MAXQUAD**, **DECVAR**, **BOUNDS**, **BY**, **CRPJAC**, **GRADIENT**, **HESSIAN**, **JACNLC**, **JACOBIAN**, **LABEL**, **LINCON**, **MATRIX**, and **NLINCON**.

Displayed Output

Procedure Initialization

After the procedure has processed the problem, it displays summary information about the problem and the options that you have selected. It may also display a list of linearly dependent constraints and other information about the constraints and parameters.

Optimization Start

At the start of optimization the procedure displays

- the number of constraints that are active at the starting point, or more precisely, the number of constraints that are currently members of the working set. If this number is followed by a plus sign, there are more active constraints, of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the starting point
- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- for the TRUREG and LEVMAR subroutines, the initial radius of the trust region around the starting point

Iteration History

In general, the iteration history consists of one line of output containing the most important information for each iteration. The iteration-extensive Nelder-Mead simplex method, however, displays only one line for several internal iterations. This technique skips the output for some iterations because

- some of the termination tests (size and standard deviation) are rather time-consuming compared to the simplex operations and are done once every five simplex operations
- the resulting history output is smaller

The `_LIST_` variable (refer to the section “[Program Statements](#)” on page 584) also enables you to display the parameter estimates $x^{(k)}$ and the gradient $g^{(k)}$ in all or some selected iterations k .

The iteration history always includes the following (the words in parentheses indicate the column header output):

- the iteration number (iter)
- the number of iteration restarts (nrest)
- the number of function calls (nfun)
- the number of active constraints (act)
- the value of the optimization criterion (optcrit)
- the difference between adjacent function values (difcrit)

- the maximum of the absolute (projected) gradient components (maxgrad)

An apostrophe trailing the number of active constraints indicates that at least one of the active constraints was released from the active set due to a significant Lagrange multiplier.

The optimization history is displayed by default because it is important to check for possible convergence problems.

Optimization Termination

The output of the optimization history ends with a short output of information concerning the optimization result:

- the number of constraints that are active at the final point, or more precisely, the number of constraints that are currently members of the working set. When this number is followed by a plus sign, it indicates that there are more active constraints of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the final point
- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- other information that is specific for the optimization technique

The **NOPRINT** option suppresses all output to the list file and only errors, warnings, and notes are displayed to the log file. The **PALL** option sets a large group of some of the commonly used specific displaying options, the **PSHORT** option suppresses some, and the **PSUMMARY** option suppresses almost all of the default output. The following table summarizes the correspondence between the general and the specific print options.

Table 6.5 Optimization Termination

Output Options	PALL	default	PSHORT	PSUMMARY	Output
	y	y	y	y	Summary of optimization
	y	y	y	n	Parameter estimates
	y	y	y	n	Gradient of objective func
PHISTORY	y	y	y	n	Iteration history
PINIT	y	y	n	n	Setting of initial values
	y	y	n	n	Listing of constraints
PGRID	y	n	n	n	Results of grid search
PNLCJAC	y	n	n	n	Jacobian nonlin. constr.
PFUNCTION	y	n	n	n	Values of functions
PEIGVAL	y	n	n	n	Eigenvalue distribution
PCRJAC	y	n	n	n	Crossproduct Jacobian
PHESSIAN	y	n	n	n	Hessian matrix
PSTDERR	y	n	n	n	Approx. standard errors
PCOV	y	n	n	n	Covariance matrices
PJACOBI	n	n	n	n	Jacobian
LIST	n	n	n	n	Model program, variables
LISTCODE	n	n	n	n	Compiled model program

Convergence Status

Upon termination, the NLP procedure creates an ODS table called “ConvergenceStatus.” You can use this name to reference the table when using the Output Delivery System (ODS) to select tables and create output data sets. Within the “ConvergenceStatus” table there are two variables, “Status” and “Reason,” which contain the status of the optimization run. For the “Status” variable, a value of zero indicates that one of the convergence criteria is satisfied; a nonzero value indicates otherwise. In all cases, an explicit interpretation of the status code is displayed as a string stored in the “Reason” variable. For more information about ODS, see *SAS Output Delivery System: User’s Guide*.

Missing Values

Missing Values in Program Statements

There is one very important reason for using missing values in program statements specifying the values of the objective functions and derivatives: it may not be possible to evaluate the program statements for a particular point x . For example, the extrapolation formula of one of the line-search algorithms may generate large x values for which the EXP function cannot be evaluated without floating point overflow. The compiler of the program statements may check for such situations automatically, but it would be safer if you check the feasibility of your program statements. In some cases, the specification of boundary or linear constraints for parameters can avoid such situations. In many other cases, you can indicate that x is a *bad* point simply by returning a missing value for the objective function. In such cases the optimization algorithms in PROC NLP shorten the step length α or reduce the trust region radius so that the next point will be closer to the point that was already successfully evaluated at the last iteration. Note that the starting point $x^{(0)}$ must be a point for which the program statements can be evaluated.

Missing Values in Input Data Sets

Observations with missing values in the `DATA=` data set for variables used in the objective function can lead to a missing value of the objective function implying that the corresponding `BY` group of data is not processed. The `NOMISS` option can be used to skip those observations of the `DATA=` data set for which relevant variables have missing values. Relevant variables are those that are referred to in program statements.

There can be different reasons to include observations with missing values in the `INEST=` data set. The value of the `_RHS_` variable is not used in some cases and can be missing. Missing values for the variables corresponding to parameters in the `_TYPE_` variable are as follows:

- `PARMS` observations cause those parameters to have initial values assigned by the `DECVAR` statement or by the `RANDOM=` or `INITIAL=` option.
- `UPPERBD` or `LOWERBD` observations cause those parameters to be unconstrained by upper or lower bounds.
- `LE`, `GE`, or `EQ` observations cause those parameters to have zero values in the constraint.

In general, missing values are treated as zeros.

Computational Resources

Since nonlinear optimization is an iterative process that depends on many factors, it is difficult to estimate how much computer time is necessary to compute an optimal solution satisfying one of the termination criteria. The `MAXTIME=`, `MAXITER=`, and `MAXFUNC=` options can be used to restrict the amount of real time, the number of iterations, and the number of function calls in a single run of PROC NLP.

In each iteration k , the NRRIDG and LEVMAR techniques use symmetric Householder transformations to decompose the $n \times n$ Hessian (crossproduct Jacobian) matrix G ,

$$G = V^T T V, \quad V \text{ orthogonal, } T \text{ tridiagonal}$$

to compute the (Newton) search direction s :

$$s^{(k)} = -G^{(k-1)} g^{(k)}, \quad k = 1, 2, 3, \dots$$

The QUADAS, TRUREG, NEWRAP, and HYQUAN techniques use the Cholesky decomposition to solve the same linear system while computing the search direction. The QUANEW, DBLDOG, CONGRA, and NMSIMP techniques do not need to invert or decompose a Hessian or crossproduct Jacobian matrix and thus require fewer computational resources than the first group of techniques.

The larger the problem, the more time is spent computing function values and derivatives. Therefore, many researchers compare optimization techniques by counting and comparing the respective numbers of function, gradient, and Hessian (crossproduct Jacobian) evaluations. You can save computer time and memory by specifying derivatives (using the `GRADIENT`, `JACOBIAN`, `CRPJAC`, or `HESSIAN` statement) since you will typically produce a more efficient representation than the internal derivative compiler.

Finite-difference approximations of the derivatives are expensive since they require additional function or gradient calls.

- Forward-difference formulas:
 - First-order derivatives: n additional function calls are needed.
 - Second-order derivatives based on function calls only: for a dense Hessian, $n(n + 3)/2$ additional function calls are needed.
 - Second-order derivatives based on gradient calls: n additional gradient calls are needed.
- Central-difference formulas:
 - First-order derivatives: $2n$ additional function calls are needed.
 - Second-order derivatives based on function calls only: for a dense Hessian, $2n(n + 1)$ additional function calls are needed.
 - Second-order derivatives based on gradient: $2n$ additional gradient calls are needed.

Many applications need considerably more time for computing second-order derivatives (Hessian matrix) than for first-order derivatives (gradient). In such cases, a (dual) quasi-Newton or conjugate gradient technique is recommended, which does not require second-order derivatives.

The following table shows for each optimization technique which derivatives are needed (FOD: first-order derivatives; SOD: second-order derivatives), what kinds of constraints are supported (BC: boundary constraints; LIC: linear constraints), and the minimal memory (number of double floating point numbers) required. For various reasons, there are additionally about $7n + m$ double floating point numbers needed.

Quadratic Programming	FOD	SOD	BC	LIC	Memory
LICOMP	-	-	x	x	$18n + 3nn$
QUADAS	-	-	x	x	$1n + 2nn/2$
General Optimization	FOD	SOD	BC	LIC	Memory
TRUREG	x	x	x	x	$4n + 2nn/2$
NEWRAP	x	x	x	x	$2n + 2nn/2$
NRRIDG	x	x	x	x	$6n + nn/2$
QUANEW	x	-	x	x	$1n + nn/2$
DBLDOG	x	-	x	x	$7n + nn/2$
CONGRA	x	-	x	x	$3n$
NMSIMP	-	-	x	x	$4n + nn$
Least Squares	FOD	SOD	BC	LIC	Memory
LEVMAR	x	-	x	x	$6n + nn/2$
HYQUAN	x	-	x	x	$2n + nn/2 + 3m$

Notes:

- Here, n denotes the number of parameters, nn the squared number of parameters, and $nn/2 := n(n + 1)/2$.
- The value of m is the product of the number of functions specified in the **MIN**, **MAX**, or **LSQ** statement and the maximum number of observations in each **BY** group of a **DATA=** input data set. The following table also contains the number v of variables in the **DATA=** data set that are used in the program statements.
- For a diagonal Hessian matrix, the $nn/2$ term in QUADAS, TRUREG, NEWRAP, and NRRIDG is replaced by n .
- If the TRUREG, NRRIDG, or NEWRAP method is used to minimize a least squares problem, the second derivatives are replaced by the crossproduct Jacobian matrix.
- The memory needed by the **TECH=NONE** specification depends on the output specifications (typically, it needs $3n + nn/2$ double floating point numbers and an additional mn if the Jacobian matrix is required).

The total amount of memory needed to run an optimization technique consists of the technique-specific memory listed in the preceding table, plus additional blocks of memory as shown in the following table.

	double	int	long	8byte
Basic Requirement	$7n + m$	n	$3n$	$n + m$
DATA= data set	v	-	-	v
JACOBIAN statement	$m(n + 2)$	-	-	-
CRPJAC statement	$nn/2$	-	-	-
HESSIAN statement	$nn/2$	-	-	-
COV= option	$(2*)nn/2 + n$	-	-	-
Scaling vector	n	-	-	-
BOUNDS statement	$2n$	n	-	-
Bounds in INEST=	$2n$	-	-	-
LINCON and TRUREG	$c(n + 1) + nn + nn/2 + 4n$	$3c$	-	-
LINCON and other	$c(n + 1) + nn + 2nn/2 + 4n$	$3c$	-	-

Notes:

- For **TECH=LICOMP**, the total amount of memory needed for the linear or boundary constrained case is $18(n + c) + 3(n + c)(n + c)$, where c is the number of constraints.
- The amount of memory needed to specify derivatives with a **GRADIENT**, **JACOBIAN**, **CRPJAC**, or **HESSIAN** statement (shown in this table) is small compared to that needed for using the internal function compiler to compute the derivatives. This is especially so for second-order derivatives.
- If the CONGRA technique is used, specifying the **GRADCHECK=DETAIL** option requires an additional $nn/2$ double floating point numbers to store the finite-difference Hessian matrix.

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory used by the SAS System. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit, which is sufficient for many applications. However, to solve most realistic optimization problems, the NLP procedure might require more memory. Increasing the memory limit can reduce the chance of an out-of-memory condition.

NOTE: The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify **-MEMSIZE 0** to indicate all available memory should be used, but this setting should be used with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify **-MEMSIZE 0**. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, **-MEMSIZE 500M** might be sufficient to enable the procedure to run without an out-of-memory condition. When problems have millions of variables, **-MEMSIZE 1000M** or higher might be needed. These are “rules of thumb”—problems with atypical structure, density, or other characteristics can increase the optimizer’s memory requirements.

The MEMSIZE option can be specified at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the SAS Companion for your operating environment.

To report a procedure’s memory consumption, you can use the FULLSTIMER option. The syntax is described in the SAS Companion for your operating environment.

Rewriting NLP Models for PROC OPTMODEL

This section covers techniques for converting NLP procedure models to OPTMODEL procedure models. For information about the OPTMODEL procedure, see Chapter 5, “The OPTMODEL Procedure” (*SAS/OR User’s Guide: Mathematical Programming*).

To illustrate the basics, consider the following first version of the NLP model for [Example 6.7](#):

```

/*****
/* Rewriting NLP Models for PROC OPTMODEL */
/*****

proc nlp all;
  parms amountx amounty amounta amountb amountc
    pooltox pooltoy ctox ctoy pools = 1;
  bounds 0 <= amountx amounty amounta amountb amountc,
    amountx <= 100,
    amounty <= 200,
    0 <= pooltox pooltoy ctox ctoy,
    1 <= pools <= 3;
  lincon amounta + amountb = pooltox + pooltoy,
    pooltox + ctox = amountx,
    pooltoy + ctoy = amounty,
    ctox + ctoy = amountc;
  nlincon nlc1-nlc2 >= 0.,
    nlc3 = 0.;
  max f;
  costa = 6; costb = 16; costc = 10;
  costx = 9; costy = 15;
  f = costx * amountx + costy * amounty
    - costa * amounta - costb * amountb - costc * amountc;
  nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
  nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
  nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;

```

These statements define a model that has bounds, linear constraints, nonlinear constraints, and a simple objective function. The following statements are a straightforward conversion of the PROC NLP statements to PROC OPTMODEL form:

```

proc optmodel;
  var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
  var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
  var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
  var ctox init 1 >= 0,
    ctoy init 1 >= 0;
  var pools init 1 >= 1 <= 3;
  con amounta + amountb = pooltox + pooltoy,
    pooltox + ctox = amountx,
    pooltoy + ctoy = amounty,
    ctox + ctoy = amountc;
  number costa, costb, costc, costx, costy;
  costa = 6; costb = 16; costc = 10;
  costx = 9; costy = 15;
  max f = costx * amountx + costy * amounty
    - costa * amounta - costb * amountb - costc * amountc;
  con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,

```

```

nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
      = 0;
solve;
print amountx amounty amounta amountb amountc
      pooltox pooltoy ctox ctoy pools;

```

The PROC OPTMODEL variable declarations are split into individual declarations because PROC OPTMODEL does not permit name lists in its declarations. In the OPTMODEL procedure, variable bounds are part of the variable declaration instead of a separate BOUNDS statement. The PROC NLP statements are as follows:

```

parms amountx amounty amounta amountb amountc
      pooltox pooltoy ctox ctoy pools = 1;
bounds 0 <= amountx amounty amounta amountb amountc,
       amountx <= 100,
       amounty <= 200,
       0 <= pooltox pooltoy ctox ctoy,
       1 <= pools <= 3;

```

The following PROC OPTMODEL statements are equivalent to the PROC NLP statements:

```

var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
var ctox init 1 >= 0,
    ctoy init 1 >= 0;
var pools init 1 >= 1 <= 3;

```

The linear constraints are declared in the NLP model with the following statement:

```

lincon amounta + amountb = pooltox + pooltoy,
      pooltox + ctox = amountx,
      pooltoy + ctoy = amounty,
      ctox + ctoy      = amountc;

```

The following linear constraint declarations in the PROC OPTMODEL model are quite similar to the PROC NLP LINCON declarations:

```

con amounta + amountb = pooltox + pooltoy,
  pooltox + ctox = amountx,
  pooltoy + ctoy = amounty,
  ctox + ctoy   = amountc;

```

But PROC OPTMODEL provides much more flexibility in defining linear constraints. For example, coefficients can be named parameters or any other expression that evaluates to a constant.

The cost parameters are declared explicitly in the PROC OPTMODEL model. Unlike the DATA step or PROC NLP, PROC OPTMODEL requires names to be declared before they are used. There are multiple ways to set the values of these parameters. The preceding example used assignments. The values could have been made part of the declaration by using the INIT *expression* clause or the = *expression* clause. The values could also have been read from a data set with the READ DATA statement.

Note in the original NLP statements that the assignment to a parameter such as *costa* occurs every time the objective function is evaluated. However, the assignment occurs just once in the PROC OPTMODEL code, when the assignment statement is processed. This works because the values are constant. But the PROC OPTMODEL statements permit the parameters to be reassigned later to interactively modify the model.

The following statements define the objective *f* in the NLP model:

```
max f;
. . .
f = costx * amountx + costly * amounty
   - costa * amounta - costb * amountb - costc * amountc;
```

The PROC OPTMODEL version of the objective is defined with the same expression text, as follows:

```
max f = costx * amountx + costly * amounty
       - costa * amounta - costb * amountb - costc * amountc;
```

But in PROC OPTMODEL the MAX statement and the assignment to the name *f* in the PROC NLP statements are combined. There are advantages and disadvantages to this approach. The PROC OPTMODEL formulation is much closer to the mathematical formulation of the model. However, if there are multiple intermediate variables being used to structure the objective, then multiple IMPVAR declarations are required.

In the PROC NLP model the nonlinear constraints use the following syntax:

```
nlincon nlc1-nlc2 >= 0.,
        nlc3 = 0.;
. . .
nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
```

In the PROC OPTMODEL model the equivalent statements are as follows:

```
con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
        nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
        nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
              = 0;
```

The nonlinear constraints in PROC OPTMODEL use the same syntax as linear constraints. In fact, if the variable *pools* were declared as a parameter, then all the preceding constraints would be linear. The nonlinear

constraint in PROC OPTMODEL combines the NLINCON statement of PROC NLP with the assignment in the PROC NLP statements. As in objective expressions, objective names can be used in nonlinear constraint expressions to structure the formula.

The PROC OPTMODEL model does not use a RUN statement to invoke the solver. Instead the solver is invoked interactively by the SOLVE statement in PROC OPTMODEL. By default, the OPTMODEL procedure prints much less data about the optimization process. Generally these data consist of messages from the solver (such as the termination reason) in addition to a short status display. The PROC OPTMODEL statements add a PRINT statement in order to display the variable estimates from the solver.

The model for [Example 6.8](#) illustrates how to convert PROC NLP statements that handle arrays into PROC OPTMODEL form. The PROC NLP model is as follows:

```
proc nlp tech=tr pall;
  array c[10] -6.089 -17.164 -34.054 -5.914 -24.721
           -14.986 -24.100 -10.708 -26.662 -22.179;
  array x[10] x1-x10;
  min y;
  parms x1-x10 = .1;
  bounds 1.e-6 <= x1-x10;
  lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
         1. = x4 + 2. * x5 + x6 + x7,
         1. = x3 + x7 + x8 + 2. * x9 + x10;
  s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
  y = 0.;
  do j = 1 to 10;
    y = y + x[j] * (c[j] + log(x[j] / s));
  end;
run;
```

The model finds an equilibrium state for a mixture of chemicals. The following statements show a corresponding PROC OPTMODEL model:

```
proc optmodel;
  set CMP = 1..10;
  number c{CMP} = [-6.089 -17.164 -34.054 -5.914 -24.721
                 -14.986 -24.100 -10.708 -26.662 -22.179];
  var x{CMP} init 0.1 >= 1.e-6;
  con 2. = x[1] + 2. * x[2] + 2. * x[3] + x[6] + x[10],
       1. = x[4] + 2. * x[5] + x[6] + x[7],
       1. = x[3] + x[7] + x[8] + 2. * x[9] + x[10];
  /* replace the variable s in the NLP model */
  impvar s = sum{i in CMP} x[i];
  min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
  solve;
  print x y;
```

The PROC OPTMODEL model uses the set CMP to represent the set of compounds, which are numbered 1 to 10 in the example. The array c was initialized by using the equivalent PROC OPTMODEL syntax. The individual array locations could also have been initialized by assignment or READ DATA statements.

The VAR declaration for variable x combines the VAR and BOUNDS statements of the PROC NLP model. The index set of the array is based on the set of compounds CMP, to simplify changes to the model.

The linear constraints are similar in form to the PROC NLP model. However, the PROC OPTMODEL version uses the array form of the variable names because the OPTMODEL procedure treats arrays as distinct variables, not as aliases of lists of scalar variables.

The implicit variable `s` replaces the intermediate variable of the same name in the PROC NLP model. This is an example of translating an intermediate variable from the other models to PROC OPTMODEL. An alternative way is to use an additional constraint for every intermediate variable. In the preceding statements, instead of declaring objective `s`, you can use the following statements:

```
. . .
var s;
con s = sum{i in CMP} x[i];
. . .
```

Note that this alternative formulation passes an extra variable and constraint to the solver. This formulation can sometimes be solved more efficiently, depending on the characteristics of the model.

The PROC OPTMODEL version uses a SUM operator over the set `CMP`, which enhances the flexibility of the model to accommodate possible changes in the set of compounds.

In the PROC NLP model the objective function `y` is determined by an explicit loop. With PROC OPTMODEL, the DO loop is replaced by a SUM aggregation operation. The accumulation in the PROC NLP model is now performed by PROC OPTMODEL with the SUM operator.

This PROC OPTMODEL model can be further generalized. Note that the array initialization and constraints assume a fixed set of compounds. You can rewrite the model to handle an arbitrary number of compounds and chemical elements. The new model loads the linear constraint coefficients from a data set along with the objective coefficients for the parameter `c`, as follows:

```
data comp;
  input c a_1 a_2 a_3;
  datalines;
-6.089  1 0 0
-17.164 2 0 0
-34.054 2 0 1
-5.914  0 1 0
-24.721 0 2 0
-14.986 1 1 0
-24.100 0 1 1
-10.708 0 0 1
-26.662 0 0 2
-22.179 1 0 1
;

data atom;
  input b @@;
  datalines;
2. 1. 1.
;

proc optmodel;
  set CMP;
  set ELT;
  number c{CMP};
```

```

number a{ELT,CMP};
number b{ELT};
read data atom into ELT=[_n_] b;
read data comp into CMP=[_n_]
    c {i in ELT} < a[i,_n_]=col("a_"||i) >;
var x{CMP} init 0.1 >= 1.e-6;
con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
impvar s = sum{i in CMP} x[i];
min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
print a b;
solve;
print x;

```

This version adds coefficients for the linear constraints to the COMP data set. The data set variable a_n represents the number of atoms in the compound for element n . The READ DATA statement for COMP uses the iterated column syntax to read each of the data set variables a_n into the appropriate location in the array a . In this example the expanded data set variable names are a_1 , a_2 , and a_3 .

The preceding version also adds a new set, ELT, of chemical elements and a numeric parameter, b , that represents the left-hand side of the linear constraints. The data values for the parameters ELT and b are read from the data set ATOM. The model can handle varying sets of chemical elements because of this extra data set and the new parameters.

The linear constraints have been converted to a single indexed family of constraints. One constraint is applied for each chemical element in the set ELT. The constraint expression uses a simple form that applies generally to linear constraints. The following PRINT statement in the model shows the values read from the data sets to define the linear constraints:

```
print a b;
```

The PRINT statements in the model produce the results shown in [Output 6.11](#).

Figure 6.11 PROC OPTMODEL Output

The OPTMODEL Procedure

a										
	1	2	3	4	5	6	7	8	9	10
1	1	2	2	0	0	1	0	0	0	1
2	0	0	0	1	2	1	1	0	0	0
3	0	0	1	0	0	0	1	1	2	1

[1] b	
1	2
2	1
3	1

Figure 6.11 continued

[1]	x
1	0.04066848
2	0.14773067
3	0.78315260
4	0.00141459
5	0.48524616
6	0.00069358
7	0.02739955
8	0.01794757
9	0.03731444
10	0.09687143

In the preceding model the chemical elements and compounds are designated by numbers. So in the PRINT output, for example, the row that is labeled “3” represents the amount of the compound H₂O. PROC OPTMODEL is capable of using more symbolic strings to designate array indices. The following version of the model uses strings to index arrays:

```

data comp;
  input name $ c a_h a_n a_o;
  datalines;
H      -6.089   1 0 0
H2     -17.164  2 0 0
H2O    -34.054  2 0 1
N      -5.914   0 1 0
N2     -24.721  0 2 0
NH     -14.986  1 1 0
NO     -24.100  0 1 1
O      -10.708  0 0 1
O2     -26.662  0 0 2
OH     -22.179  1 0 1
;
data atom;
  input name $ b;
  datalines;
H 2.
N 1.
O 1.
;
proc optmodel;
  set<string> CMP;
  set<string> ELT;
  number c{CMP};
  number a{ELT,CMP};
  number b{ELT};
  read data atom into ELT=[name] b;
  read data comp into CMP=[name]
    c {i in ELT} < a[i,name]=col("a_"||i) >;
  var x{CMP} init 0.1 >= 1.e-6;
  con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
  impvar s = sum{i in CMP} x[i];

```

```

min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
solve;
print x;

```

In this model, the sets CMP and ELT are now sets of strings. The data sets provide the names of the compounds and elements. The names of the data set variables for atom counts in the data set COMP now include the chemical element symbol as part of their spelling. For example, the atom count for element H (hydrogen) is named `a_h`. Note that these changes did not require any modification to the specifications of the linear constraints or the objective.

The PRINT statement in the preceding statements produces the results shown in [Output 6.12](#). The indices of variable `x` are now strings that represent the actual compounds.

Figure 6.12 PROC OPTMODEL Output with Strings

The OPTMODEL Procedure

[1]	x
H	0.04066848
H2	0.14773067
H2O	0.78315260
N	0.00141459
N2	0.48524616
NH	0.00069358
NO	0.02739955
O	0.01794757
O2	0.03731444
OH	0.09687143

Examples: NLP Procedure

Example 6.1: Using the DATA= Option

This example illustrates the use of the `DATA=` option. The Bard function (refer to Moré, Garbow, and Hillstom (1981)) is a least squares problem with $n = 3$ parameters and $m = 15$ functions f_k :

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

where

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

with $u_k = k$, $v_k = 16 - k$, $w_k = \min(u_k, v_k)$, and

$$y = (.14, .18, .22, .25, .29, .32, .35, .39, .37, .58, .73, .96, 1.34, 2.10, 4.39)$$

The minimum function value $f(x^*) = 4.107\text{E-}3$ is at the point (0.08, 1.13, 2.34). The starting point $x^0 = (1, 1, 1)$ is used. The following is the naive way of specifying the objective function.

```
proc nlp tech=levmar;
  lsq y1-y15;
  parms x1-x3 = 1;
  tmp1 = 15 * x2 + min(1,15) * x3;
  y1 = 0.14 - (x1 + 1 / tmp1);
  tmp1 = 14 * x2 + min(2,14) * x3;
  y2 = 0.18 - (x1 + 2 / tmp1);
  tmp1 = 13 * x2 + min(3,13) * x3;
  y3 = 0.22 - (x1 + 3 / tmp1);
  tmp1 = 12 * x2 + min(4,12) * x3;
  y4 = 0.25 - (x1 + 4 / tmp1);
  tmp1 = 11 * x2 + min(5,11) * x3;
  y5 = 0.29 - (x1 + 5 / tmp1);
  tmp1 = 10 * x2 + min(6,10) * x3;
  y6 = 0.32 - (x1 + 6 / tmp1);
  tmp1 = 9 * x2 + min(7,9) * x3;
  y7 = 0.35 - (x1 + 7 / tmp1);
  tmp1 = 8 * x2 + min(8,8) * x3;
  y8 = 0.39 - (x1 + 8 / tmp1);
  tmp1 = 7 * x2 + min(9,7) * x3;
  y9 = 0.37 - (x1 + 9 / tmp1);
  tmp1 = 6 * x2 + min(10,6) * x3;
  y10 = 0.58 - (x1 + 10 / tmp1);
  tmp1 = 5 * x2 + min(11,5) * x3;
  y11 = 0.73 - (x1 + 11 / tmp1);
  tmp1 = 4 * x2 + min(12,4) * x3;
  y12 = 0.96 - (x1 + 12 / tmp1);
  tmp1 = 3 * x2 + min(13,3) * x3;
  y13 = 1.34 - (x1 + 13 / tmp1);
  tmp1 = 2 * x2 + min(14,2) * x3;
  y14 = 2.10 - (x1 + 14 / tmp1);
  tmp1 = 1 * x2 + min(15,1) * x3;
  y15 = 4.39 - (x1 + 15 / tmp1);
run;
```

A more economical way to program this problem uses the `DATA=` option to input the 15 terms in $f(x)$.

```
data bard;
  input r @@;
  w1 = 16. - _n_;
  w2 = min(_n_ , 16. - _n_);
  datalines;
.14 .18 .22 .25 .29 .32 .35 .39
.37 .58 .73 .96 1.34 2.10 4.39
;
```

```
proc nlp data=bard tech=levmar;
  lsq y;
  parms x1-x3 = 1.;
  y = r - (x1 + _obs_ / (w1 * x2 + w2 * x3));
run;
```

Another way you can specify the objective function uses the `ARRAY` statement and an explicit do loop, as in the following code.

```
proc nlp tech=levmar;
  array r[15] .14 .18 .22 .25 .29 .32 .35 .39 .37 .58
           .73 .96 1.34 2.10 4.39 ;
  array y[15] y1-y15;
  lsq y1-y15;
  parms x1-x3 = 1.;
  do i = 1 to 15;
    w1 = 16. - i;
    w2 = min(i , w1);
    w3 = w1 * x2 + w2 * x3;
    y[i] = r[i] - (x1 + i / w3);
  end;
run;
```

Example 6.2: Using the INQUAD= Option

This example illustrates the `INQUAD=` option for specifying a quadratic programming problem:

$$\min f(x) = \frac{1}{2}x^T Gx + g^T x + c, \quad \text{with } G^T = G$$

Suppose that $c = -100$, $G = \text{diag}(.4, 4)$ and $2 \leq x_1 \leq 50$, $-50 \leq x_2 \leq 50$, and $10 \leq 10x_1 - x_2$.

You specify the constant c and the Hessian G in the data set `QUAD1`. Notice that the `_TYPE_` variable contains the keywords that identify how the procedure should interpret the observations.

```
data quad1;
  input _type_ $ _name_ $ x1 x2;
  datalines;
const . -100 -100
quad x1 0.4 0
quad x2 0 4
;
```

You specify the QUAD1 data set with the `INQUAD=` option. Notice that the names of the variables in the QUAD1 data set and the `_NAME_` variable match the names of the parameters in the `PARMS` statement.

```
proc nlp inquad=quad1 all;
  min ;
  parms x1 x2 = -1;
  bounds 2 <= x1 <= 50,
         -50 <= x2 <= 50;
  lincon 10 <= 10 * x1 - x2;
run;
```

Alternatively, you can use a sparse format for specifying the G matrix, eliminating the zeros. You use the special variables `_ROW_`, `_COL_`, and `_VALUE_` to give the nonzero row and column names and value.

```
data quad2;
  input _type_ $ _row_ $ _col_ $ _value_;
  datalines;
const . . -100
quad x1 x1 0.4
quad x2 x2 4
;
```

You can also include the constraints in the QUAD data set. Notice how the `_TYPE_` variable contains keywords that identify how the procedure is to interpret the values in each observation.

```
data quad3;
  input _type_ $ _name_ $ x1 x2 _rhs_;
  datalines;
const . -100 -100 .
quad x1 0.02 0 .
quad x2 0.00 2 .
parms . -1 -1 .
lowerbd . 2 -50 .
upperbd . 50 50 .
ge . 10 -1 10
;
```

```
proc nlp inquad=quad3;
  min ;
  parms x1 x2;
run;
```

Example 6.3: Using the INEST=Option

This example illustrates the use of the `INEST=` option for specifying a starting point and linear constraints. You name a data set with the `INEST=` option. The format of this data set is similar to the format of the `QUAD` data set described in the previous example.

Consider the Hock and Schittkowski (1981) Problem # 24:

$$\min f(x) = \frac{((x_1 - 3)^2 - 9)x_2^3}{27\sqrt{3}}$$

subject to:

$$\begin{aligned} 0 &\leq x_1, x_2 \\ 0 &\leq .57735x_1 - x_2 \\ 0 &\leq x_1 + 1.732x_2 \\ 6 &\geq x_1 + 1.732x_2 \end{aligned}$$

with minimum function value $f(x^*) = -1$ at $x^* = (3, \sqrt{3})$. The feasible starting point is $x^0 = (1, .5)$.

You can specify this model in `PROC NLP` as follows:

```
proc nlp tech=trureg outest=res;
  min y;
  parms x1 = 1,
        x2 = .5;
  bounds 0 <= x1-x2;
  lincon .57735 * x1 -          x2 >= 0,
        x1 + 1.732 * x2 >= 0,
        -x1 - 1.732 * x2 >= -6;
  y = (((x1 - 3)**2 - 9.) * x2**3) / (27 * sqrt(3));
run;
```

Note that none of the data for this model are in a data set. Alternatively, you can save the starting point (1, .5) and the linear constraints in a data set. Notice that the `_TYPE_` variable contains keywords that identify how the procedure is to interpret each of the observations and that the parameters in the problems `X1` and `X2` are variables in the data set. The observation with `_TYPE_=LOWERBD` gives the lower bounds on the parameters. The observation with `_TYPE_=GE` gives the coefficients for the first constraint. Similarly, the subsequent observations contain specifications for the other constraints. Also notice that the special variable `_RHS_` contains the right-hand-side values.

```

data betts1(type=est);
  input _type_ $ x1 x2 _rhs_;
  datalines;
parms    1      .5      .
lowerbd  0      0      .
ge       .57735  -1     .
ge       1      1.732  .
le       1      1.732  6
;

```

Now you can solve this problem with the following code. Notice that you specify the objective function and the parameters.

```

proc nlp inest=betts1 tech=trureg;
  min y;
  parms x1 x2;
  y = ((x1 - 3)**2 - 9) * x2**3 / (27 * sqrt(3));
run;

```

You can even include any constants used in the program statements in the `INEST=` data set. In the following code the variables A, B, C, and D contain some of the constants used in calculating the objective function Y.

```

data betts2(type=est);
  input _type_ $ x1 x2 _rhs_ a b c d;
  datalines;
parms    1      .5      .      3  9 27 3
lowerbd  0      0      .      .  .  . .
ge       .57735  -1     0      .  .  . .
ge       1      1.732  0      .  .  . .
le       1      1.732  6      .  .  . .
;

```

Notice that in the program statement for calculating Y, the constants are replaced by the A, B, C, and D variables.

```

proc nlp inest=betts2 tech=trureg;
  min y;
  parms x1 x2;
  y = ((x1 - a)**2 - b) * x2**3 / (c * sqrt(d));
run;

```

Example 6.4: Restarting an Optimization

This example shows how you can restart an optimization problem using the `OUTEST=`, `INEST=`, `OUTMODEL=`, and `MODEL=` options and how to save output into an `OUT=` data set. The least squares solution of the Rosenbrock function using the trust region method is used.

The following code solves the problem and saves the model in the `MODEL` data set and the solution in the `EST` and `OUT1` data sets.

```
proc nlp tech=trureg outmodel=model outest=est out=out1;
  lsq y1 y2;
  parms x1 = -1.2 ,
        x2 = 1.;
  y1 = 10. * (x2 - x1 * x1);
  y2 = 1. - x1;
run;

proc print data=out1;
run;
```

The final parameter estimates $x^* = (1, 1)$ and the values of the functions $f_1 = Y1$ and $f_2 = Y2$ are written into an `OUT=` data set, shown in [Output 6.4.1](#). Since `OUTDER=0` is the default, the `OUT=` data set does not contain the Jacobian matrix.

Output 6.4.1 Solution in an `OUT=` Data Set

Obs	_OBS_	_TYPE_	y1	y2	x2	x1
1	1		0	3.3307E-16	1	1

Next, the procedure reads the optimal parameter estimates from the `EST` data set and the model from the `MODEL` data set. It does not do any optimization (`TECH=NONE`), but it saves the Jacobian matrix to the `OUT=OUT2` data set because of the option `OUTDER=1`. It also displays the Jacobian matrix because of the option `PJAC`; the Jacobian matrix is shown in [Output 6.4.2](#). [Output 6.4.3](#) shows the contents of the `OUT2` data set, which also contains the Jacobian matrix.

```
proc nlp tech=none model=model inest=est out=out2 outder=1 pjac PHISTORY;
  lsq y1 y2;
  parms x1 x2;
run;

proc print data=out2;
run;
```

Output 6.4.2 Jacobian Matrix Output
PROC NLP: Least Squares Minimization

Jacobian Matrix	
x1	x2
-20	10
-1	0

Output 6.4.3 Jacobian Matrix in an OUT= Data Set

Obs	_OBS_	_TYPE_	y1	y2	_WRT_	x2	x1
1	1		0	0		1	1
2	1	ANALYTIC	10	0	x2	1	1
3	1	ANALYTIC	-20	-1	x1	1	1

Example 6.5: Approximate Standard Errors

The NLP procedure provides a variety of ways for estimating parameters in nonlinear statistical models and for obtaining approximate standard errors and covariance matrices for the estimators. These methods are illustrated by estimating the mean of a random sample from a normal distribution with mean μ and standard deviation σ . The simplicity of the example makes it easy to compare the results of different methods in NLP with the usual estimator, the sample mean.

The following data step is used:

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

The standard error of the mean, computed with $n - 1$ degrees of freedom, is 1. The usual maximum-likelihood approximation to the standard error of the mean, using a variance divisor of n rather than $n - 1$, is 0.894427.

The sample mean is a least squares estimator, so it can be computed using an LSQ statement. Moreover, since this model is linear, the Hessian matrix and crossproduct Jacobian matrix are identical, and all three versions of the **COV=** option yield the same variance and standard error of the mean. Note that **COV=j** means that the crossproduct Jacobian is used. This is chosen because it requires the least computation.

```
proc nlp data=x cov=j pstderr pshort PHISTORY;
  lsq resid;
  parms mean=0;
  resid=x-mean;
run;
```

The results are the same as the usual estimates.

Output 6.5.1 Parameter Estimates**PROC NLP: Least Squares Minimization**

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	1.000000	4.000000	0.016130	0

Value of Objective Function = 10

PROC NLP can also compute maximum-likelihood estimates of μ and σ . In this case it is convenient to minimize the negative log likelihood. To get correct standard errors for maximum-likelihood estimators, the `SIGSQ=1` option is required. The following program shows `COV=1` but the output that follows has `COV=2` and `COV=3`.

```
proc nlp data=x cov=1 sigsq=1 pstderr phes pcov pshort;
  min nloglik;
  parms mean=0, sigma=1;
  bounds 1e-12 < sigma;
  nloglik=.5*((x-mean)/sigma)**2 + log(sigma);
run;
```

The variance divisor is n instead of $n - 1$, so the standard error of the mean is 0.894427 instead of 1. The standard error of the mean is the same with all six types of covariance matrix, but the standard error of the standard deviation varies. The sampling distribution of the standard deviation depends on the higher moments of the population distribution, so different methods of estimation can produce markedly different estimates of the standard error of the standard deviation.

Output 6.5.2 shows the output when COV=1, Output 6.5.3 shows the output when COV=2, and Output 6.5.4 shows the output when COV=3.

Output 6.5.2 Solution for COV=1

PROC NLP: Nonlinear Minimization

Optimization Results
Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	1.33149E-10
2	sigma	2.000000	0.458258	4.364358	0.007260	-5.606415E-9

Value of Objective Function = 5.9657359028

Hessian Matrix

	mean	sigma
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 1:
M = (NOBS/d) inv(G) JJ(f) inv(G)

	mean	sigma
mean	0.8	1.980107E-11
sigma	1.980107E-11	0.2099999991

Factor sigm = 1

Determinant = 0.1679999993

Matrix has Only Positive Eigenvalues

Output 6.5.3 Solution for COV=2

PROC NLP: Nonlinear Minimization

Optimization Results
Parameter Estimates

N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566	1.33149E-10
2	sigma	2.000000	0.632456	3.162278	0.025031	-5.606415E-9

Output 6.5.3 *continued*

Value of Objective Function = 5.9657359028

Hessian Matrix		
	mean	sigma
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	mean	sigma
mean	0.7999999982	4.260766E-11
sigma	4.260766E-11	0.3999999978

Factor sigm = 1

Determinant = 0.3199999975

Matrix has Only Positive Eigenvalues

Output 6.5.4 Solution for COV=3

PROC NLP: Nonlinear Minimization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	0.509136	7.856442	0.000537	1.301733E-10
2	sigma	2.000000	0.419936	4.762634	0.005048	-5.940302E-9

Value of Objective Function = 5.9657359028

Hessian Matrix		
	mean	sigma
mean	1.2500000028	-1.33149E-10
sigma	-1.33149E-10	2.500000014

Determinant = 3.1250000245

Matrix has Only Positive Eigenvalues

Covariance Matrix 3: J = (1/d) inv(W)		
	mean	sigma
mean	0.2592197879	1.062283E-11
sigma	1.062283E-11	0.1763460041

Output 6.5.4 *continued***Factor sigm = 0.2****Determinant = 0.0457123738****Matrix has Only Positive Eigenvalues**

Under normality, the maximum-likelihood estimators of μ and σ are independent, as indicated by the diagonal Hessian matrix in the previous example. Hence, the maximum-likelihood estimate of μ can be obtained by using any fixed value for σ , such as 1. However, if the fixed value of σ differs from the actual maximum-likelihood estimate (in this case 2), the model is misspecified and the standard errors obtained with **COV=2** or **COV=3** are incorrect. It is therefore necessary to use **COV=1**, which yields consistent estimates of the standard errors under a variety of forms of misspecification of the error distribution.

```
proc nlp data=x cov=1 sigsq=1 pstderr pcov pshort;
  min sqresid;
  parms mean=0;
  sqresid=.5*(x-mean)**2;
run;
```

This formulation produces the same standard error of the mean, 0.894427 (see [Output 6.5.5](#)).

Output 6.5.5 Solution for Fixed σ and **COV=1****PROC NLP: Nonlinear Minimization**

Optimization Results					
Parameter Estimates					
N	Parameter	Estimate	Approx Std Err	t Value	Gradient Approx Pr > t Objective Function
1	mean	4.000000	0.894427	4.472136	0.006566

Value of Objective Function = 10

Covariance
Matrix 1:
M = (NOBS/d)
inv(G) JJ(f)
inv(G)

mean	
mean	0.8

Factor sigm = 1

The maximum-likelihood formulation with fixed σ is actually a least squares problem. The objective function, parameter estimates, and Hessian matrix are the same as those in the first example in this section using the **LSQ** statement. However, the Jacobian matrix is different, each row being multiplied by twice the residual. To treat this formulation as a least squares problem, the **SIGSQ=1** option can be omitted. But since the Jacobian is not the same as in the formulation using the **LSQ** statement, the **COV=1 | M** and **COV=3 | J** options, which use the Jacobian, do not yield correct standard errors. The correct standard error is obtained with **COV=2 | H**, which uses only the Hessian matrix:

```
proc nlp data=x cov=2 pstderr pcov pshort;
  min sqresid;
  parms mean=0;
  sqresid=.5*(x-mean)**2;
run;
```

The results are the same as in the first example.

Output 6.5.6 Solution for Fixed σ and COV=2

PROC NLP: Nonlinear Minimization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	mean	4.000000	0.500000	8.000000	0.001324	0

Value of Objective Function = 10

Covariance Matrix 2: H = (NOBS/d) inv(G)	
	mean
mean	0.25

Factor $\text{sigm} = 1.25$

In summary, to obtain appropriate standard errors for least squares estimates, you can use the **LSQ** statement with any of the **COV=** options, or you can use the **MIN** statement with **COV=2**. To obtain appropriate standard errors for maximum-likelihood estimates, you can use the **MIN** statement with the negative log likelihood or the **MAX** statement with the log likelihood, and in either case you can use any of the **COV=** options provided that you specify **SIGSQ=1**. You can also use a log-likelihood function with a misspecified scale parameter provided that you use **SIGSQ=1** and **COV=1**. For nonlinear models, all of these methods yield approximations based on asymptotic theory, and should therefore be interpreted cautiously.

Example 6.6: Maximum Likelihood Weibull Estimation

Two-Parameter Weibull Estimation

The following data are taken from Lawless (1982, p. 193) and represent the number of days it took rats painted with a carcinogen to develop carcinoma. The last two observations are censored data from a group of 19 rats:

```
data pike;
  input days cens @@;
  datalines;
143 0 164 0 188 0 188 0
190 0 192 0 206 0 209 0
```

```

213  0  216  0  220  0  227  0
230  0  234  0  246  0  265  0
304  0  216  1  244  1
;

```

Suppose that you want to show how to compute the maximum likelihood estimates of the scale parameter σ (α in Lawless), the shape parameter c (β in Lawless), and the location parameter θ (μ in Lawless). The observed likelihood function of the three-parameter Weibull transformation (Lawless 1982, p. 191) is

$$L(\theta, \sigma, c) = \frac{c^m}{\sigma^m} \prod_{i \in D} \left(\frac{t_i - \theta}{\sigma} \right)^{c-1} \prod_{i=1}^p \exp \left(- \left(\frac{t_i - \theta}{\sigma} \right)^c \right)$$

and the log likelihood is

$$l(\theta, \sigma, c) = m \log c - mc \log \sigma + (c - 1) \sum_{i \in D} \log(t_i - \theta) - \sum_{i=1}^p \left(\frac{t_i - \theta}{\sigma} \right)^c$$

The log likelihood function can be evaluated only for $\sigma > 0$, $c > 0$, and $\theta < \min_i t_i$. In the estimation process, you must enforce these conditions using lower and upper boundary constraints. The three-parameter Weibull estimation can be numerically difficult, and it usually pays off to provide good initial estimates. Therefore, you first estimate σ and c of the two-parameter Weibull distribution for constant $\theta = 0$. You then use the optimal parameters $\hat{\sigma}$ and \hat{c} as starting values for the three-parameter Weibull estimation.

Although the use of an `INEST=` data set is not really necessary for this simple example, it illustrates how it is used to specify starting values and lower boundary constraints:

```

data par1(type=est);
  keep _type_ sig c theta;
  _type_='parms'; sig = .5;
    c = .5; theta = 0; output;
  _type_='lb'; sig = 1.0e-6;
    c = 1.0e-6; theta = .; output;
run;

```

The following `PROC NLP` call specifies the maximization of the log likelihood function for the two-parameter Weibull estimation for constant $\theta = 0$:

```

proc nlp data=pike tech=tr inest=par1 outest=opar1
  outmodel=model cov=2 vardef=n pcov phes;
  max logf;
  parms sig c;
  profile sig c / alpha = .9 to .1 by -.1 .09 to .01 by -.01;

  x_th = days - theta;
  s     = - (x_th / sig)**c;
  if cens=0 then s + log(c) - c*log(sig) + (c-1)*log(x_th);
  logf = s;
run;

```

After a few iterations you obtain the solution given in [Output 6.6.1](#).

Output 6.6.1 Optimization Results
PROC NLP: Nonlinear Maximization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	sig	234.318611	9.645908	24.292021	9.050475E-16	1.3372183E-9
2	c	6.083147	1.068229	5.694611	0.000017269	-7.859276E-9

Value of Objective Function = -88.23273515

Since the gradient has only small elements and the Hessian (shown in [Output 6.6.2](#)) is negative definite (has only negative eigenvalues), the solution defines an isolated maximum point.

Output 6.6.2 Hessian Matrix at x^*
PROC NLP: Nonlinear Maximization

Hessian Matrix		
	sig	c
sig	-0.011457556	0.0257527577
c	0.0257527577	-0.934221388

Determinant = 0.0100406894

Matrix has Only Negative Eigenvalues

The square roots of the diagonal elements of the approximate covariance matrix of parameter estimates are the approximate standard errors (ASE's). The covariance matrix is given in [Output 6.6.3](#).

Output 6.6.3 Covariance Matrix
PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)		
	sig	c
sig	93.043549863	2.5648395794
c	2.5648395794	1.141112488

Factor sigm = 1

Determinant = 99.594754608

Matrix has 2 Positive Eigenvalue(s)

The confidence limits in [Output 6.6.4](#) correspond to the α values in the `PROFILE` statement.

Output 6.6.4 Confidence Limits
PROC NLP: Nonlinear Maximization

Wald and PL Confidence Limits						
N	Parameter	Estimate	Alpha	Profile Likelihood Confidence Limits		Wald Confidence Limits
1	sig	234.318611	0.900000	233.111324	235.532695	233.106494 235.530729
1	sig	.	0.800000	231.886549	236.772876	231.874849 236.762374
1	sig	.	0.700000	230.623280	238.063824	230.601846 238.035377
1	sig	.	0.600000	229.292797	239.436639	229.260292 239.376931
1	sig	.	0.500000	227.855829	240.935290	227.812545 240.824678
1	sig	.	0.400000	226.251597	242.629201	226.200410 242.436813
1	sig	.	0.300000	224.372260	244.643392	224.321270 244.315953
1	sig	.	0.200000	221.984557	247.278423	221.956882 246.680341
1	sig	.	0.100000	218.390824	251.394102	218.452504 250.184719
1	sig	.	0.090000	217.884162	251.987489	217.964960 250.672263
1	sig	.	0.080000	217.326988	252.645278	217.431654 251.205569
1	sig	.	0.070000	216.708814	253.383546	216.841087 251.796136
1	sig	.	0.060000	216.008815	254.228034	216.176649 252.460574
1	sig	.	0.050000	215.199301	255.215496	215.412978 253.224245
1	sig	.	0.040000	214.230116	256.411041	214.508337 254.128885
1	sig	.	0.030000	213.020874	257.935686	213.386118 255.251105
1	sig	.	0.020000	211.369067	260.066128	211.878873 256.758350
1	sig	.	0.010000	208.671091	263.687174	209.472398 259.164825
2	c	6.083147	0.900000	5.950029	6.217752	5.948912 6.217382
2	c	.	0.800000	5.815559	6.355576	5.812514 6.353780
2	c	.	0.700000	5.677909	6.499187	5.671537 6.494757
2	c	.	0.600000	5.534275	6.651789	5.522967 6.643327
2	c	.	0.500000	5.380952	6.817880	5.362638 6.803656
2	c	.	0.400000	5.212344	7.004485	5.184103 6.982191
2	c	.	0.300000	5.018784	7.225733	4.975999 7.190295
2	c	.	0.200000	4.776379	7.506166	4.714157 7.452137
2	c	.	0.100000	4.431310	7.931669	4.326067 7.840227
2	c	.	0.090000	4.382687	7.991457	4.272075 7.894220
2	c	.	0.080000	4.327815	8.056628	4.213014 7.953280
2	c	.	0.070000	4.270773	8.129238	4.147612 8.018682
2	c	.	0.060000	4.207130	8.211221	4.074029 8.092265
2	c	.	0.050000	4.134675	8.306218	3.989457 8.176837
2	c	.	0.040000	4.049531	8.418782	3.889274 8.277021
2	c	.	0.030000	3.945037	8.559677	3.764994 8.401300
2	c	.	0.020000	3.805759	8.749130	3.598076 8.568219
2	c	.	0.010000	3.588814	9.056751	3.331572 8.834722

Three-Parameter Weibull Estimation

You now prepare for the three-parameter Weibull estimation by using PROC UNIVARIATE to obtain the smallest data value for the upper boundary constraint for θ . For this small problem, you can do this much more simply by just using a value slightly smaller than the minimum data value 143.

```

/* Calculate upper bound for theta parameter */
proc univariate data=pike noprint;
  var days;
  output out=stats n=nobs min=minx range=range;
run;

data stats;
  set stats;
  keep _type_ theta;

  /* 1. write parms observation */
  theta = minx - .1 * range;
  if theta < 0 then theta = 0;
  _type_ = 'parms';
  output;

  /* 2. write ub observation */
  theta = minx * (1 - 1e-4);
  _type_ = 'ub';
  output;
run;

```

The data set PAR2 specifies the starting values and the lower and upper bounds for the three-parameter Weibull problem:

```

proc sort data=opar1;
  by _type_;
run;

data par2(type=est);
  merge opar1(drop=theta) stats;
  by _type_;
  keep _type_ sig c theta;
  if _type_ in ('parms' 'lowerbd' 'ub');
run;

```

The following PROC NLP call uses the MODEL= input data set containing the log likelihood function that was saved during the two-parameter Weibull estimation:

```

proc nlp data=pike tech=tr inest=par2 outest=opar2
  model=model cov=2 vardef=n pcov phes;
  max logf;
  parms sig c theta;
  profile sig c theta / alpha = .5 .1 .05 .01;
run;

```

After a few iterations, you obtain the solution given in [Output 6.6.5](#).

Output 6.6.5 Optimization Results
PROC NLP: Nonlinear Maximization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Approx Std Err	t Value	Approx Pr > t	Gradient Objective Function
1	sig	108.382607	32.573174	3.327358	0.003540	-9.253755E-9
2	c	2.711474	1.058754	2.561005	0.019107	-0.000001433
3	theta	122.026060	28.692215	4.252933	0.000430	-0.000000199

Value of Objective Function = -87.32424712

From inspecting the first- and second-order derivatives at the optimal solution, you can verify that you have obtained an isolated maximum point. The Hessian matrix is shown in [Output 6.6.6](#).

Output 6.6.6 Hessian Matrix
PROC NLP: Nonlinear Maximization

Hessian Matrix			
	sig	c	theta
sig	-0.01063996	0.0453889075	-0.010033738
c	0.0453889075	-4.078731918	-0.083028454
theta	-0.010033738	-0.083028454	-0.014752273

Determinant = 0.0000502159

Matrix has Only Negative Eigenvalues

The square roots of the diagonal elements of the approximate covariance matrix of parameter estimates are the approximate standard errors. The covariance matrix is given in [Output 6.6.7](#).

Output 6.6.7 Covariance Matrix
PROC NLP: Nonlinear Maximization

Covariance Matrix 2: H = (NOBS/d) inv(G)			
	sig	c	theta
sig	1061.0211695	29.926121154	-890.0886743
c	29.926121154	1.1209670393	-26.66339253
theta	-890.0886743	-26.66339253	823.25165894

Factor sigm = 1

Determinant = 19915.602625

Matrix has 3 Positive Eigenvalue(s)

The difference between the Wald and profile CLs for parameter PHI2 are remarkable, especially for the upper 95% and 99% limits, as shown in [Output 6.6.8](#).

Output 6.6.8 Confidence Limits
PROC NLP: Nonlinear Maximization

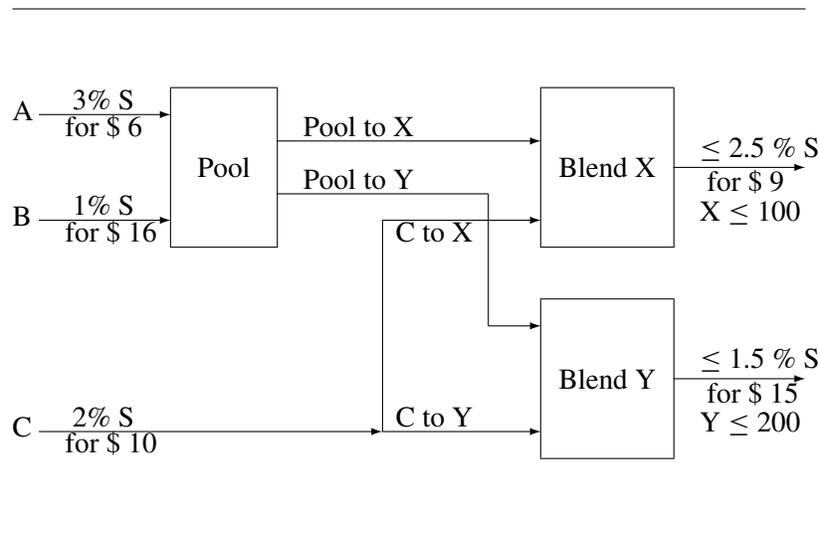
Wald and PL Confidence Limits							
N	Parameter	Estimate	Alpha	Profile Likelihood Confidence Limits		Wald Confidence Limits	
1	sig	108.382647	0.500000	91.811561	141.564605	86.412327	130.352967
1	sig	.	0.100000	76.502372	.	54.804427	161.960867
1	sig	.	0.050000	72.215850	.	44.540260	172.225033
1	sig	.	0.010000	64.262384	.	24.479529	192.285765
2	c	2.711475	0.500000	2.139297	3.704052	1.997355	3.425594
2	c	.	0.100000	1.574162	9.250073	0.969977	4.452972
2	c	.	0.050000	1.424853	19.518417	0.636352	4.786597
2	c	.	0.010000	1.163096	19.542924	-0.015699	5.438648
3	theta	122.026022	0.500000	91.027146	135.095454	102.673369	141.378675
3	theta	.	0.100000	.	141.833769	74.831411	142.985700
3	theta	.	0.050000	.	142.512603	65.790174	142.985700
3	theta	.	0.010000	.	142.967407	48.119590	142.985700

Example 6.7: Simple Pooling Problem

The following optimization problem is discussed in Haverly (1978) and in Liebman et al. (1986, pp. 127–128). Two liquid chemicals, *X* and *Y*, are produced by the pooling and blending of three input liquid chemicals, *A*, *B*, and *C*. You know the sulfur impurity amounts of the input chemicals, and you have to respect upper limits of the sulfur impurity amounts of the output chemicals. The sulfur concentrations and the prices of the input and output chemicals are:

- Chemical *A*: Concentration = 3%, Price= \$6
- Chemical *B*: Concentration = 1%, Price= \$16
- Chemical *C*: Concentration = 2%, Price= \$10
- Chemical *X*: Concentration \leq 2.5%, Price= \$9
- Chemical *Y*: Concentration \leq 1.5%, Price= \$15

The problem is complicated by the fact that the two input chemicals *A* and *B* are available only as a mixture (they are either shipped together or stored together). Because the amounts of *A* and *B* are unknown, the sulfur concentration of the mixture is also unknown.



You know customers will buy no more than 100 units of X and 200 units of Y. The problem is determining how to operate the pooling and blending of the chemicals to maximize the profit. The objective function for the profit is

$$\begin{aligned} \text{profit} &= \text{cost}(x) \times \text{amount}(x) + \text{cost}(y) \times \text{amount}(y) \\ &- \text{cost}(a) \times \text{amount}(a) - \text{cost}(b) \times \text{amount}(b) - \text{cost}(c) \times \text{amount}(c) \end{aligned}$$

There are three groups of constraints:

- The first group of constraint functions is the mass balance restrictions illustrated by the graph. These are four linear equality constraints:
 - $\text{amount}(a) + \text{amount}(b) = \text{pool_to_x} + \text{pool_to_y}$
 - $\text{pool_to_x} + \text{c_to_x} = \text{amount}(x)$
 - $\text{pool_to_y} + \text{c_to_y} = \text{amount}(y)$
 - $\text{amount}(c) = \text{c_to_x} + \text{c_to_y}$
- You introduce a new variable, pool_s , that represents the sulfur concentration of the pool. Using pool_s and the sulfur concentration of C (2%), you obtain two nonlinear inequality constraints for the sulfur concentrations of X and Y, one linear equality constraint for the sulfur balance, and lower and upper boundary restrictions for pool_s :
 - $\text{pool_s} \times \text{pool_to_x} + 2 \text{c_to_x} \leq 2.5 \text{amount}(x)$
 - $\text{pool_s} \times \text{pool_to_y} + 2 \text{c_to_y} \leq 1.5 \text{amount}(y)$
 - $3 \text{amount}(a) + 1 \text{amount}(b) = \text{pool_s} \times (\text{amount}(a) + \text{amount}(b))$
 - $1 \leq \text{pool_s} \leq 3$

3. The last group assembles the remaining boundary constraints. First, you do not want to produce more than you can sell; and finally, all variables must be nonnegative:

- $amount(x) \leq 100, \quad amount(y) \leq 200$
- $amount(a), amount(b), amount(c), amount(x), amount(y) \geq 0$
- $pool_to_x, pool_to_y, c_to_x, c_to_y \geq 0$

There exist several local optima to this problem that can be found by specifying different starting points. Using the starting point with all variables equal to 1 (specified with a **PARMS** statement), PROC NLP finds a solution with $profit = 400$:

```
proc nlp all;
  parms amountx amounty amounta amountb amountc
        pooltox pooltoy ctox ctoy pools = 1;
        bounds 0 <= amountx amounty amounta amountb amountc,
               amountx <= 100,
               amounty <= 200,
               0 <= pooltox pooltoy ctox ctoy,
               1 <= pools <= 3;
  lincon amounta + amountb = pooltox + pooltoy,
        pooltox + ctox = amountx,
        pooltoy + ctoy = amounty,
        ctox + ctoy = amountc;
  nlincon nlc1-nlc2 >= 0.,
        nlc3 = 0.;
  max f;
  costa = 6; costb = 16; costc = 10;
  costx = 9; costy = 15;
  f = costx * amountx + costy * amounty
      - costa * amounta - costb * amountb - costc * amountc;
  nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
  nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
  nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

The specified starting point was not feasible with respect to the linear equality constraints; therefore, a starting point is generated that satisfies linear and boundary constraints. [Output 6.7.1](#) gives the starting parameter estimates.

Output 6.7.1 Starting Estimates

PROC NLP: Nonlinear Maximization

Optimization Start						
Parameter Estimates						
N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Lower Bound Constraint	Upper Bound Constraint
1	amountx	1.363636	9.000000	-0.843698	0	100.000000
2	amounty	1.363636	15.000000	-0.111882	0	200.000000
3	amounta	0.818182	-6.000000	-0.430733	0	.
4	amountb	0.818182	-16.000000	-0.542615	0	.
5	amountc	1.090909	-10.000000	0.017768	0	.
6	pooltox	0.818182	0	-0.669628	0	.
7	pooltoy	0.818182	0	-0.303720	0	.
8	ctox	0.545455	0	-0.174070	0	.
9	ctoy	0.545455	0	0.191838	0	.
10	pools	2.000000	0	0.068372	1.000000	3.000000

Value of Objective Function = 3.8181818182

Value of Lagrange Function = -2.866739915

PROC NLP: Nonlinear Maximization

Optimization Results					
Parameter Estimates					
N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Active Bound Constraint
1	amountx	-1.40474E-11	9.000000	0	Lower BC
2	amounty	200.000000	15.000000	-8.88178E-16	Upper BC
3	amounta	5.561161E-16	-6.000000	0	Lower BC
4	amountb	100.000000	-16.000000	1.065814E-14	
5	amountc	100.000000	-10.000000	-1.77636E-15	
6	pooltox	7.024225E-12	0	0	Lower BC
7	pooltoy	100.000000	0	1.776357E-15	
8	ctox	-2.10716E-11	0	1.776357E-15	Lower BC LinDep
9	ctoy	100.000000	0	5.329071E-15	
10	pools	1.000000	0	4.973799E-14	Lower BC LinDep

The starting point satisfies the four equality constraints, as shown in [Output 6.7.2](#). The nonlinear constraints are given in [Output 6.7.3](#).

Output 6.7.2 Linear Constraints

PROC NLP: Nonlinear Maximization

Linear Constraints

- 1 2.2204E-16 : ACT 0 == + 1.0000 * amounta + 1.0000 * amountb - 1.0000 * pooltox - 1.0000 * pooltoy
- 2 0 : ACT 0 == - 1.0000 * amountx + 1.0000 * pooltox + 1.0000 * ctox
- 3 -1.11E-16 : ACT 0 == - 1.0000 * amounty + 1.0000 * pooltoy + 1.0000 * ctoy
- 4 -1.11E-16 : ACT 0 == - 1.0000 * amountc + 1.0000 * ctox + 1.0000 * ctoy

Output 6.7.3 Nonlinear Constraints
PROC NLP: Nonlinear Maximization

Values of Nonlinear Constraints

Constraint	Value	Residual	Lagrange Multiplier	
[5] nlc3	0	0	4.9441	Active NLEC
[6] nlc1_G	0.6818	0.6818	.	
[7] nlc2_G	-0.6818	-0.6818	-9.8046	Violat. NLIC

PROC NLP: Nonlinear Maximization

Values of Nonlinear Constraints

Constraint	Value	Residual	Lagrange Multiplier	
[5] nlc3	1.11E-15	1.11E-15	6.0000	Active NLEC
[6] nlc1_G	4.31E-16	4.31E-16	.	Active NLIC LinDep
[7] nlc2_G	0	0	-6.0000	Active NLIC

Output 6.7.4 shows the settings of some important PROC NLP options.

Output 6.7.4 Options

PROC NLP: Nonlinear Maximization

Minimum Iterations	0
Maximum Iterations	200
Maximum Function Calls	500
Iterations Reducing Constraint Violation	20
ABSGCONV Gradient Criterion	0.00001
GCONV Gradient Criterion	1E-8
ABSFCONV Function Criterion	0
FCONV Function Criterion	2.220446E-16
FCONV2 Function Criterion	1E-6
FSIZE Parameter	0
ABSXCONV Parameter Change Criterion	0
XCONV Parameter Change Criterion	0
XSIZE Parameter	0
ABSCONV Function Criterion	1.340781E154
Line Search Method	2
Starting Alpha for Line Search	1
Line Search Precision LSPRECISSION	0.4
DAMPSTEP Parameter for Line Search	.
FD Derivatives: Accurate Digits in Obj.F	15.653559775
FD Derivatives: Accurate Digits in NLCon	15.653559775
Singularity Tolerance (SINGULAR)	1E-8
Constraint Precision (LCEPS)	1E-8
Linearly Dependent Constraints (LCSING)	1E-8
Releasing Active Constraints (LCDEACT)	.

The iteration history, given in [Output 6.7.5](#), does not show any problems.

Output 6.7.5 Iteration History
PROC NLP: Nonlinear Maximization
Dual Quasi-Newton Optimization
Modified VMCWD Algorithm of Powell (1978, 1982)
Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
Lagrange Multiplier Update of Powell(1982)

Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Maximum Gradient Element of the Lagrange Function
1	0	19	-1.42400	0.00962	6.9131	1.000	0.783
2	0	20	2.77026	0.0166	5.3770	1.000	2.629
3	0	21	7.08706	0.1409	7.1965	1.000	9.452
4	0	22	11.41264	0.0583	15.5769	1.000	23.390
5	0	23	24.84607	1.78E-15	496.1	1.000	147.6
6	0	24	378.22829	147.4	3316.8	1.000	840.4
7	0	25	307.56787	50.9338	607.9	1.000	27.143
8	0	26	347.24475	1.8328	21.9882	1.000	28.482
9	0	27	349.49273	0.00915	7.1826	1.000	28.289
10	0	28	356.58291	0.1083	50.2554	1.000	27.479
11	0	29	388.70709	2.4280	24.7997	1.000	21.114
12	0	30	389.30094	0.0157	10.0473	1.000	18.647
13	0	31	399.19199	0.7996	11.1866	1.000	0.416
14	0	32	400.00000	0.0128	0.1534	1.000	0.00087
15	0	33	400.00000	7.38E-11	2.43E-10	1.000	365E-12

Optimization Results			
Iterations	15	Function Calls	34
Gradient Calls	18	Active Constraints	10
Objective Function	400	Maximum Constraint Violation	7.381118E-11
Maximum Projected Gradient	0	Value Lagrange Function	-400
Maximum Gradient of the Lagran Func	4.973799E-14	Slope of Search Direction	-2.43334E-10

FCONV2 convergence criterion satisfied.

The optimal solution in [Output 6.7.6](#) shows that to obtain the maximum profit of \$400, you need only to produce the maximum 200 units of blending *Y* and no units of blending *X*.

Output 6.7.6 Optimization Solution

PROC NLP: Nonlinear Maximization

Optimization Results						
Parameter Estimates						
N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function	Active Bound	Constraint
1	amountx	-1.40474E-11	9.000000	0	Lower BC	
2	amounty	200.000000	15.000000	-8.88178E-16	Upper BC	
3	amounta	5.561161E-16	-6.000000	0	Lower BC	
4	amountb	100.000000	-16.000000	1.065814E-14		
5	amountc	100.000000	-10.000000	-1.77636E-15		
6	pooltox	7.024225E-12	0	0	Lower BC	
7	pooltoy	100.000000	0	1.776357E-15		
8	ctox	-2.10716E-11	0	1.776357E-15	Lower BC	LinDep
9	ctoy	100.000000	0	5.329071E-15		
10	pools	1.000000	0	4.973799E-14	Lower BC	LinDep

Value of Objective Function = 400

Value of Lagrange Function = 400

The constraints are satisfied at the solution, as shown in [Output 6.7.7](#)

Output 6.7.7 Linear and Nonlinear Constraints at the Solution

PROC NLP: Nonlinear Maximization

Linear Constraints Evaluated at Solution

1	ACT	$0 = 0 + 1.0000 * \text{amounta} + 1.0000 * \text{amountb} - 1.0000 * \text{pooltox} - 1.0000 * \text{pooltoy}$
2	ACT	$3.8975E-17 = 0 - 1.0000 * \text{amountx} + 1.0000 * \text{pooltox} + 1.0000 * \text{ctox}$
3	ACT	$0 = 0 - 1.0000 * \text{amounty} + 1.0000 * \text{pooltoy} + 1.0000 * \text{ctoy}$
4	ACT	$0 = 0 - 1.0000 * \text{amountc} + 1.0000 * \text{ctox} + 1.0000 * \text{ctoy}$

Values of Nonlinear Constraints

Constraint	Value	Residual	Lagrange Multiplier	
[5] nlc3	1.11E-15	1.11E-15	6.0000	Active NLEC
[6] nlc1_G	4.31E-16	4.31E-16	.	Active NLIC LinDep
[7] nlc2_G	0	0	-6.0000	Active NLIC

Linearly Dependent Active Boundary Constraints

Parameter	N	Kind
ctox	8	Lower BC
pools	10	Lower BC

Output 6.7.7 *continued*

Linearly Dependent Gradients of Active Nonlinear Constraints	
Parameter	N
nlc3	6

The same problem can be specified in many different ways. For example, the following specification uses an `INEST=` data set containing the values of the starting point and of the constants `COST`, `COSTB`, `COSTC`, `COSTX`, `COSTY`, `CA`, `CB`, `CC`, and `CD`:

```
data init1(type=est);
  input _type_ $ amountx amounty amounta amountb
         amountc pooltox pooltoy cttox cttoy pools
         _rhs_ costa costb costc costx costy
         ca cb cc cd;
  datalines;
parms 1 1 1 1 1 1 1 1 1 1
      . 6 16 10 9 15 2.5 1.5 2. 3.
;

proc nlp inest=init1 all;
  parms amountx amounty amounta amountb amountc
         pooltox pooltoy cttox cttoy pools;
  bounds 0 <= amountx amounty amounta amountb amountc,
         amountx <= 100,
         amounty <= 200,
         0 <= pooltox pooltoy cttox cttoy,
         1 <= pools <= 3;
  lincon amounta + amountb = pooltox + pooltoy,
         pooltox + cttox = amountx,
         pooltoy + cttoy = amounty,
         cttox + cttoy = amountc;
  nlincon nlc1-nlc2 >= 0.,
         nlc3 = 0.;
  max f;
  f = costx * amountx + costy * amounty
      - costa * amounta - costb * amountb - costc * amountc;
  nlc1 = ca * amountx - pools * pooltox - cc * cttox;
  nlc2 = cb * amounty - pools * pooltoy - cc * cttoy;
  nlc3 = cd * amounta + amountb - pools * (amounta + amountb);
run;
```

The third specification uses an `INEST=` data set containing the boundary and linear constraints in addition to the values of the starting point and of the constants. This specification also writes the model specification into an `OUTMOD=` data set:

```
data init2(type=est);
  input _type_ $ amountx amounty amounta amountb amountc
         pooltox pooltoy cttox cttoy pools
         _rhs_ costa costb costc costx costy;
```

```

datalines;
parms      1   1   1   1   1   1   1   1   1   1
           .   6  16  10  9   15  2.5  1.5  2   3
lowerbd    0   0   0   0   0   0   0   0   0   1
           .   .   .   .   .   .   .   .   .   .
upperbd   100 200 .   .   .   .   .   .   .   3
           .   .   .   .   .   .   .   .   .   .
eq         .   .   1   1   .   -1  -1   .   .   .
           0   .   .   .   .   .   .   .   .   .
eq         1   .   .   .   .   -1   .   -1   .   .
           0   .   .   .   .   .   .   .   .   .
eq         .   1   .   .   .   .   -1   .   -1   .
           0   .   .   .   .   .   .   .   .   .
eq         .   .   .   .   1   .   .   -1  -1   .
           0   .   .   .   .   .   .   .   .   .
;

proc nlp inest=init2 outmod=model all;
  parms amountx amounty amounta amountb amountc
        pooltox pooltoy ctox ctoy pools;
  nlincon nlc1-nlc2 >= 0.,
          nlc3 = 0.;
  max f;
  f = costx * amountx + costy * amounty
      - costa * amounta - costb * amountb - costc * amountc;
  nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
  nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
  nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;

```

The fourth specification not only reads the `INEST=INIT2` data set, it also uses the model specification from the `MODEL` data set that was generated in the last specification. The `PROC NLP` call now contains only the defining variable statements:

```

proc nlp inest=init2 model=model all;
  parms amountx amounty amounta amountb amountc
        pooltox pooltoy ctox ctoy pools;
  nlincon nlc1-nlc2 >= 0.,
          nlc3 = 0.;
  max f;
run;

```

All four specifications start with the same starting point of all variables equal to 1 and generate the same results. However, there exist several local optima to this problem, as is pointed out in Liebman et al. (1986, p. 130).

```

proc nlp inest=init2 model=model all;
  parms amountx amounty amounta amountb amountc
        pooltox pooltoy ctox ctoy = 0,
        pools = 2;
  nlincon nlc1-nlc2 >= 0.,
          nlc3 = 0.;
  max f;
run;

```

This starting point with all variables equal to 0 is accepted as a local solution with $profit = 0$, which minimizes rather than maximizes the profit.

Example 6.8: Chemical Equilibrium

The following example is used in many test libraries for nonlinear programming and was taken originally from Bracken and McCormick (1968).

The problem is to determine the composition of a mixture of various chemicals satisfying its chemical equilibrium state. The second law of thermodynamics implies that a mixture of chemicals satisfies its chemical equilibrium state (at a constant temperature and pressure) when the free energy of the mixture is reduced to a minimum. Therefore the composition of the chemicals satisfying its chemical equilibrium state can be found by minimizing the function of the free energy of the mixture.

Notation:

- m number of chemical elements in the mixture
- n number of compounds in the mixture
- x_j number of moles for compound j , $j = 1, \dots, n$
- s total number of moles in the mixture ($s = \sum_{j=1}^n x_j$)
- a_{ij} number of atoms of element i in a molecule of compound j
- b_i atomic weight of element i in the mixture

Constraints for the Mixture:

- The number of moles must be positive:

$$x_j > 0, \quad j = 1, \dots, n$$

- There are m mass balance relationships,

$$\sum_{j=1}^n a_{ij} x_j = b_i, \quad i = 1, \dots, m$$

Objective Function: Total Free Energy of Mixture

$$f(x) = \sum_{j=1}^n x_j \left[c_j + \ln \left(\frac{x_j}{s} \right) \right]$$

with

$$c_j = \left(\frac{F^\circ}{RT} \right)_j + \ln P$$

where F°/RT is the model standard free energy function for the j th compound (found in tables) and P is the total pressure in atmospheres.

Minimization Problem:

Determine the parameters x_j that minimize the objective function $f(x)$ subject to the nonnegativity and linear balance constraints.

Numeric Example:

Determine the equilibrium composition of compound $\frac{1}{2}N_2H_4 + \frac{1}{2}O_2$ at temperature $T = 3500^\circ\text{K}$ and pressure $P = 750\text{psi}$.

j	Compound	$(F^\circ/RT)_j$	c_j	a_{ij}		
				$i = 1$	$i = 2$	$i = 3$
1	H	-10.021	-6.089	1		
2	H_2	-21.096	-17.164	2		
3	H_2O	-37.986	-34.054	2		1
4	N	-9.846	-5.914		1	
5	N_2	-28.653	-24.721		2	
6	NH	-18.918	-14.986	1	1	
7	NO	-28.032	-24.100		1	1
8	O	-14.640	-10.708			1
9	O_2	-30.594	-26.662			2
10	OH	-26.111	-22.179	1		1

Example Specification:

```

proc nlp tech=tr pall;
  array c[10] -6.089 -17.164 -34.054 -5.914 -24.721
           -14.986 -24.100 -10.708 -26.662 -22.179;
  array x[10] x1-x10;
  min y;
  parms x1-x10 = .1;
  bounds 1.e-6 <= x1-x10;
  lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
         1. = x4 + 2. * x5 + x6 + x7,
         1. = x3 + x7 + x8 + 2. * x9 + x10;
  s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
  y = 0.;
  do j = 1 to 10;
    y = y + x[j] * (c[j] + log(x[j] / s));
  end;
run;

```

Displayed Output:

The iteration history given in [Output 6.8.1](#) does not show any problems.

Output 6.8.1 Iteration History
PROC NLP: Nonlinear Minimization
Trust Region Optimization
Without Parameter Scaling

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Trust Region Radius
1	0	2	3	-47.33412	2.2790	6.0765	2.456	1.000
2	0	3	3	-47.70043	0.3663	8.5592	0.908	0.418
3	0	4	3	-47.73074	0.0303	6.4942	0	0.359
4	0	5	3	-47.73275	0.00201	4.7606	0	0.118
5	0	6	3	-47.73554	0.00279	3.2125	0	0.0168
6	0	7	3	-47.74223	0.00669	1.9552	110.6	0.00271
7	0	8	3	-47.75048	0.00825	1.1157	102.9	0.00563
8	0	9	3	-47.75876	0.00828	0.4165	3.787	0.0116
9	0	10	3	-47.76101	0.00224	0.0716	0	0.0121
10	0	11	3	-47.76109	0.000083	0.00238	0	0.0111
11	0	12	3	-47.76109	9.609E-8	2.733E-6	0	0.00248

Optimization Results			
Iterations	11	Function Calls	13
Hessian Calls	12	Active Constraints	3
Objective Function	-47.76109086	Max Abs Gradient Element	1.8637498E-6
Lambda	0	Actual Over Pred Change	0
Radius	0.0024776027		

GCONV convergence criterion satisfied.

Output 6.8.2 lists the optimal parameters with the gradient.

Output 6.8.2 Optimization Results
PROC NLP: Nonlinear Minimization

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function
1	x1	0.040668	-9.785055
2	x2	0.147730	-19.570110
3	x3	0.783153	-34.792170
4	x4	0.001414	-12.968921
5	x5	0.485247	-25.937841
6	x6	0.000693	-22.753976
7	x7	0.027399	-28.190984
8	x8	0.017947	-15.222060
9	x9	0.037314	-30.444120
10	x10	0.096871	-25.007115

Output 6.8.2 *continued*

Value of Objective Function = -47.76109086

The three equality constraints are satisfied at the solution, as shown in [Output 6.8.3](#).

Output 6.8.3 Linear Constraints at Solution

PROC NLP: Nonlinear Minimization

Linear Constraints Evaluated at Solution

1 ACT 4.8572E-16 = 2.0000 - 1.0000 * x1 - 2.0000 * x2 - 2.0000 * x3 - 1.0000 * x6 - 1.0000 * x10
 2 ACT 2.8796E-16 = 1.0000 - 1.0000 * x4 - 2.0000 * x5 - 1.0000 * x6 - 1.0000 * x7
 3 ACT 1.1102E-16 = 1.0000 - 1.0000 * x3 - 1.0000 * x7 - 1.0000 * x8 - 2.0000 * x9 - 1.0000 * x10

The Lagrange multipliers are given in [Output 6.8.4](#).

Output 6.8.4 Lagrange Multipliers

PROC NLP: Nonlinear Minimization

First Order Lagrange Multipliers	
Active Constraint	Lagrange Multiplier
Linear EC [1]	9.785055
Linear EC [2]	12.968921
Linear EC [3]	15.222060

The elements of the projected gradient must be small to satisfy a necessary first-order optimality condition. The projected gradient is given in [Output 6.8.5](#).

Output 6.8.5 Projected Gradient

PROC NLP: Nonlinear Minimization

Projected Gradient	
Free Dimension	Projected Gradient
1	4.5770097E-9
2	6.868334E-10
3	-7.283017E-9
4	-0.000001864
5	-0.000001434
6	-0.000001361
7	-0.000000294

The projected Hessian matrix shown in [Output 6.8.6](#) is positive definite, satisfying the second-order optimality condition.

Output 6.8.6 Projected Hessian Matrix**PROC NLP: Nonlinear Minimization**

Projected Hessian Matrix							
	X1	X2	X3	X4	X5	X6	X7
X1	20.903196985	-0.122067474	2.6480263467	3.3439156526	-1.373829641	-1.491808185	1.1462413516
X2	-0.122067474	565.97299938	106.54631863	-83.7084843	-37.43971036	-36.20703737	-16.635529
X3	2.6480263467	106.54631863	1052.3567179	-115.230587	182.89278895	175.97949593	-57.04158208
X4	3.3439156526	-83.7084843	-115.230587	37.529977667	-4.621642366	-4.574152161	10.306551561
X5	-1.373829641	-37.43971036	182.89278895	-4.621642366	79.326057844	22.960487404	-12.69831637
X6	-1.491808185	-36.20703737	175.97949593	-4.574152161	22.960487404	66.669897023	-8.121228758
X7	1.1462413516	-16.635529	-57.04158208	10.306551561	-12.69831637	-8.121228758	14.690478023

The following **PROC NLP** call uses a specified analytical gradient and the Hessian matrix is computed by finite-difference approximations based on the analytic gradient:

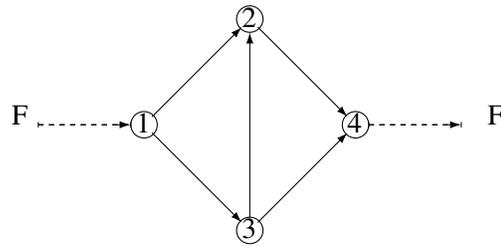
```
proc nlp tech=tr fdhessian all;
  array c[10] -6.089 -17.164 -34.054 -5.914 -24.721
           -14.986 -24.100 -10.708 -26.662 -22.179;
  array x[10] x1-x10;
  array g[10] g1-g10;
  min y;
  parms x1-x10 = .1;
  bounds 1.e-6 <= x1-x10;
  lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
         1. = x4 + 2. * x5 + x6 + x7,
         1. = x3 + x7 + x8 + 2. * x9 + x10;
  s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
  y = 0.;
  do j = 1 to 10;
    y = y + x[j] * (c[j] + log(x[j] / s));
    g[j] = c[j] + log(x[j] / s);
  end;
run;
```

The results are almost identical to those of the previous run.

Example 6.9: Minimize Total Delay in a Network

The following example is taken from the user's guide of GINO (Liebman et al. 1986). A simple network of five roads (arcs) can be illustrated by the path diagram:

Figure 6.13 Simple Road Network



The five roads connect four intersections illustrated by numbered nodes. Each minute F vehicles enter and leave the network. Arc (i, j) refers to the road from intersection i to intersection j , and the parameter x_{ij} refers to the flow from i to j . The law that traffic flowing into each intersection j must also flow out is described by the linear equality constraint

$$\sum_i x_{ij} = \sum_i x_{ji}, \quad j = 1, \dots, n$$

In general, roads also have an upper capacity, which is the number of vehicles which can be handled per minute. The upper limits c_{ij} can be enforced by boundary constraints

$$0 \leq x_{ij} \leq c_{ij}, \quad i, j = 1, \dots, n$$

Finding the maximum flow through a network is equivalent to solving a simple linear optimization problem, and for large problems, **PROC LP** or **PROC NETFLOW** can be used. The objective function is

$$\max \quad f = x_{24} + x_{34}$$

and the constraints are

$$\begin{aligned} x_{13} &= x_{32} + x_{34} \\ x_{12} + x_{32} &= x_{24} \\ x_{12} + x_{13} &= x_{24} + x_{34} \\ 0 \leq x_{12}, x_{32}, x_{34} &\leq 10 \\ 0 \leq x_{13}, x_{24} &\leq 30 \end{aligned}$$

The three linear equality constraints are linearly dependent. One of them is deleted automatically by the **PROC NLP** subroutines. Even though the default technique is used for this small example, any optimization subroutine can be used.

```

proc nlp all initial=.5;
max y;
parms x12 x13 x32 x24 x34;
bounds x12 <= 10,
       x13 <= 30,
       x32 <= 10,
       x24 <= 30,

```

```

x34 <= 10;
/* what flows into an intersection must flow out */
lincon x13 = x32 + x34,
      x12 + x32 = x24,
      x24 + x34 = x12 + x13;
y = x24 + x34 + 0*x12 + 0*x13 + 0*x32;
run;

```

The iteration history is given in Output 6.9.1, and the optimal solution is given in Output 6.9.2.

Output 6.9.1 Iteration History

PROC NLP: Nonlinear Maximization

Newton-Raphson Ridge Optimization

Without Parameter Scaling

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Between Actual and Predicted Change
1 *	0	2	4	20.25000	19.2500	0.5774	0.0313	0.860
2 *	0	3	5	30.00000	9.7500	0	0.0313	1.683

Optimization Results			
Iterations	2	Function Calls	4
Hessian Calls	3	Active Constraints	5
Objective Function	30	Max Abs Gradient Element	0
Ridge	0	Actual Over Pred Change	1.6834532374

All parameters are actively constrained. Optimization cannot proceed.

Output 6.9.2 Optimization Results

PROC NLP: Nonlinear Maximization

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function Active Bound Constraint
1	x12	10.000000	0 Upper BC
2	x13	20.000000	0
3	x32	10.000000	0 Upper BC
4	x24	20.000000	1.000000
5	x34	10.000000	1.000000 Upper BC

Value of Objective Function = 30

Finding a traffic pattern that minimizes the total delay to move F vehicles per minute from node 1 to node 4 introduces nonlinearities that, in turn, demand nonlinear optimization techniques. As traffic volume increases, speed decreases. Let t_{ij} be the travel time on arc (i, j) and assume that the following formulas describe the travel time as decreasing functions of the amount of traffic:

$$\begin{aligned}t_{12} &= 5 + 0.1x_{12}/(1 - x_{12}/10) \\t_{13} &= x_{13}/(1 - x_{13}/30) \\t_{32} &= 1 + x_{32}/(1 - x_{32}/10) \\t_{24} &= x_{24}/(1 - x_{24}/30) \\t_{34} &= 5 + 0.1x_{34}/(1 - x_{34}/10)\end{aligned}$$

These formulas use the road capacities (upper bounds), assuming $F = 5$ vehicles per minute have to be moved through the network. The objective function is now

$$\min f = t_{12}x_{12} + t_{13}x_{13} + t_{32}x_{32} + t_{24}x_{24} + t_{34}x_{34}$$

and the constraints are

$$\begin{aligned}x_{13} &= x_{32} + x_{34} \\x_{12} + x_{32} &= x_{24} \\x_{24} + x_{34} &= F = 5 \\0 \leq x_{12}, x_{32}, x_{34} &\leq 10 \\0 \leq x_{13}, x_{24} &\leq 30\end{aligned}$$

Again, the default algorithm is used:

```
proc nlp all initial=.5;
  min y;
  parms x12 x13 x32 x24 x34;
  bounds x12 x13 x32 x24 x34 >= 0;
  lincon x13 = x32 + x34, /* flow in = flow out */
         x12 + x32 = x24,
         x24 + x34 = 5; /* = f = desired flow */
  t12 = 5 + .1 * x12 / (1 - x12 / 10);
  t13 = x13 / (1 - x13 / 30);
  t32 = 1 + x32 / (1 - x32 / 10);
  t24 = x24 / (1 - x24 / 30);
  t34 = 5 + .1 * x34 / (1 - x34 / 10);
  y = t12*x12 + t13*x13 + t32*x32 + t24*x24 + t34*x34;
run;
```

The iteration history is given in [Output 6.9.3](#), and the optimal solution is given in [Output 6.9.4](#).

Output 6.9.3 Iteration History
PROC NLP: Nonlinear Minimization
Newton-Raphson Ridge Optimization
Without Parameter Scaling

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Between Actual and Predicted Change
1	0	2	4	40.30303	0.3433	4.44E-16	0	0.508

Optimization Results			
Iterations	1	Function Calls	3
Hessian Calls	2	Active Constraints	4
Objective Function	40.303030303	Max Abs Gradient Element	4.440892E-16
Ridge	0	Actual Over Pred Change	0.5083585587

ABSGCONV convergence criterion satisfied.

Output 6.9.4 Optimization Results
PROC NLP: Nonlinear Minimization

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function Active Bound Constraint
1	x12	2.500000	5.777778
2	x13	2.500000	5.702479
3	x32	1.114018E-17	1.000000 Lower BC
4	x24	2.500000	5.702479
5	x34	2.500000	5.777778

Value of Objective Function = 40.303030303

The active constraints and corresponding Lagrange multiplier estimates (costs) are given in [Output 6.9.5](#) and [Output 6.9.6](#), respectively.

Output 6.9.5 Linear Constraints at Solution
PROC NLP: Nonlinear Minimization

Linear Constraints Evaluated at Solution	
1 ACT 4.4409E-16 =	0 + 1.0000 * x13 - 1.0000 * x32 - 1.0000 * x34
2 ACT 4.4409E-16 =	0 + 1.0000 * x12 + 1.0000 * x32 - 1.0000 * x24
3 ACT	0 = -5.0000 + 1.0000 * x24 + 1.0000 * x34

Output 6.9.6 Lagrange Multipliers at Solution**PROC NLP: Nonlinear Minimization**

First Order Lagrange Multipliers		
	Active Constraint	Lagrange Multiplier
Lower BC	x32	0.924702
Linear EC	[1]	5.702479
Linear EC	[2]	5.777778
Linear EC	[3]	11.480257

Output 6.9.7 shows that the projected gradient is very small, satisfying the first-order optimality criterion.

Output 6.9.7 Projected Gradient at Solution**PROC NLP: Nonlinear Minimization**

Projected Gradient	
Free Dimension	Projected Gradient
1	4.440892E-16

The projected Hessian matrix (shown in Output 6.9.8) is positive definite, satisfying the second-order optimality criterion.

Output 6.9.8 Projected Hessian at Solution**PROC NLP: Nonlinear Minimization**

Projected Hessian Matrix	
	X1
X1	1.535309013

References

- Abramowitz, M., and Stegun, I. A., eds. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. 10th printing. New York: Dover.
- Al-Baali, M., and Fletcher, R. (1985). "Variational Methods for Nonlinear Least Squares." *Journal of the Operations Research Society* 36:405–421.
- Al-Baali, M., and Fletcher, R. (1986). "An Efficient Line Search for Nonlinear Least Squares." *Journal of Optimization Theory and Applications* 48:359–377.
- Bard, Y. (1974). *Nonlinear Parameter Estimation*. New York: Academic Press.
- Beale, E. M. L. (1972). "A Derivation of Conjugate Gradients." In *Numerical Methods for Nonlinear Optimization*, edited by F. A. Lootsma, 39–43. London: Academic Press.

- Betts, J. T. (1977). "An Accelerated Multiplier Method for Nonlinear Programming." *Journal of Optimization Theory and Applications* 21:137–174.
- Bracken, J., and McCormick, G. P. (1968). *Selected Applications of Nonlinear Programming*. New York: John Wiley & Sons.
- Chamberlain, R. M., Powell, M. J. D., Lemarechal, C., and Pedersen, H. C. (1982). "The Watchdog Technique for Forcing Convergence in Algorithms for Constrained Optimization." *Mathematical Programming* 16:1–17.
- Cramer, J. S. (1986). *Econometric Applications of Maximum Likelihood Methods*. Cambridge: Cambridge University Press.
- Dennis, J. E., Gay, D. M., and Welsch, R. E. (1981). "An Adaptive Nonlinear Least-Squares Algorithm." *ACM Transactions on Mathematical Software* 7:348–368.
- Dennis, J. E., and Mei, H. H. W. (1979). "Two New Unconstrained Optimization Algorithms Which Use Function and Gradient Values." *Journal of Optimization Theory and Applications* 28:453–482.
- Dennis, J. E., and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.
- Eskow, E., and Schnabel, R. B. (1991). "Algorithm 695: Software for a New Modified Cholesky Factorization." *ACM Transactions on Mathematical Software* 17:306–312.
- Fletcher, R. (1987). *Practical Methods of Optimization*. 2nd ed. Chichester, UK: John Wiley & Sons.
- Fletcher, R., and Powell, M. J. D. (1963). "A Rapidly Convergent Descent Method for Minimization." *Computer Journal* 6:163–168.
- Fletcher, R., and Xu, C. (1987). "Hybrid Methods for Nonlinear Least Squares." *Journal of Numerical Analysis* 7:371–389.
- Gallant, A. R. (1987). *Nonlinear Statistical Models*. New York: John Wiley & Sons.
- Gay, D. M. (1983). "Subroutines for Unconstrained Minimization." *ACM Transactions on Mathematical Software* 9:503–524.
- George, J. A., and Liu, J. W. (1981). *Computer Solutions of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall.
- Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H. (1983). "Computing Forward-Difference Intervals for Numerical Optimization." *SIAM Journal on Scientific and Statistical Computing* 4:310–321.
- Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H. (1984). "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints." *ACM Transactions on Mathematical Software* 10:282–298.
- Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical Optimization*. New York: Academic Press.
- Goldfeld, S. M., Quandt, R. E., and Trotter, H. F. (1966). "Maximisation by Quadratic Hill-Climbing." *Econometrica* 34:541–551.

- Hambleton, R. K., Swaminathan, H., and Rogers, H. J. (1991). *Fundamentals of Item Response Theory*. Newbury Park, CA: Sage Publications.
- Hartmann, W. M. (1992a). *Applications of Nonlinear Optimization with PROC NLP and SAS/IML Software*. Technical report, SAS Institute Inc., Cary, NC.
- Hartmann, W. M. (1992b). *Nonlinear Optimization in IML, Releases 6.08, 6.09, 6.10*. Technical report, SAS Institute Inc., Cary, NC.
- Haverly, C. A. (1978). “Studies of the Behavior of Recursion for the Pooling Problem.” *SIGMAP Bulletin, Association for Computing Machinery* 25:19–28.
- Hock, W., and Schittkowski, K. (1981). *Test Examples for Nonlinear Programming Codes*. Vol. 187 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer-Verlag.
- Jennrich, R. I., and Sampson, P. F. (1968). “Application of Stepwise Regression to Nonlinear Estimation.” *Technometrics* 10:63–72.
- Lawless, J. F. (1982). *Statistical Methods and Methods for Lifetime Data*. New York: John Wiley & Sons.
- Liebman, J., Lasdon, L., Schrage, L., and Waren, A. (1986). *Modeling and Optimization with GINO*. Redwood City, CA: Scientific Press.
- Lindström, P., and Wedin, P. A. (1984). “A New Line-Search Algorithm for Nonlinear Least-Squares Problems.” *Mathematical Programming* 29:268–296.
- Moré, J. J. (1978). “The Levenberg-Marquardt Algorithm: Implementation and Theory.” In *Lecture Notes in Mathematics*, vol. 30, edited by G. A. Watson, 105–116. Berlin: Springer-Verlag.
- Moré, J. J., Garbow, B. S., and Hillstom, K. E. (1981). “Testing Unconstrained Optimization Software.” *ACM Transactions on Mathematical Software* 7:17–41.
- Moré, J. J., and Sorensen, D. C. (1983). “Computing a Trust-Region Step.” *SIAM Journal on Scientific and Statistical Computing* 4:553–572.
- Moré, J. J., and Wright, S. J. (1993). *Optimization Software Guide*. Philadelphia: SIAM.
- Murtagh, B. A., and Saunders, M. A. (1983). *MINOS 5.0 User's Guide*. Technical Report SOL 83-20, Stanford University.
- Nelder, J. A., and Mead, R. (1965). “A Simplex Method for Function Minimization.” *Computer Journal* 7:308–313.
- Polak, E. (1971). *Computational Methods in Optimization*. New York: Academic Press.
- Powell, M. J. D. (1977). “Restart Procedures for the Conjugate Gradient Method.” *Mathematical Programming* 12:241–254.
- Powell, M. J. D. (1978a). “Algorithms for Nonlinear Constraints That Use Lagrangian Functions.” *Mathematical Programming* 14:224–248.
- Powell, M. J. D. (1978b). “A Fast Algorithm for Nonlinearly Constrained Optimization Calculations.” In *Lecture Notes in Mathematics*, vol. 630, edited by G. A. Watson, 144–175. Berlin: Springer-Verlag.

- Powell, M. J. D. (1982a). “Extensions to Subroutine VF02AD.” In *Systems Modeling and Optimization, Lecture Notes in Control and Information Sciences*, vol. 38, edited by R. F. Drenick, and F. Kozin, 529–538. Berlin: Springer-Verlag.
- Powell, M. J. D. (1982b). *VMCWD: A Fortran Subroutine for Constrained Optimization*. Technical Report DAMTP 1982/NA4, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.
- Powell, M. J. D. (1992). *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*. Technical Report DAMTP 1992/NA5, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.
- Rosenbrock, H. H. (1960). “An Automatic Method for Finding the Greatest or Least Value of a Function.” *Computer Journal* 3:175–184.
- Schittkowski, K. (1980). *Nonlinear Programming Codes—Information, Tests, Performance*. Vol. 183 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer-Verlag.
- Schittkowski, K. (1987). *More Test Examples for Nonlinear Programming Codes*. Vol. 282 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer-Verlag.
- Schittkowski, K., and Stoer, J. (1979). “A Factorization Method for the Solution of Constrained Linear Least Squares Problems Allowing Subsequent Data Changes.” *Numerische Mathematik* 31:431–463.
- Stewart, G. W. (1967). “A Modification of Davidon’s Minimization Method to Accept Difference Approximations of Derivatives.” *Journal of the Association for Computing Machinery* 14:72–83.
- Wedin, P. A., and Lindström, P. (1987). *Methods and Software for Nonlinear Least Squares Problems*. Technical Report No. UMINF 133.87, Umeå University, Sweden.
- Whitaker, D., Triggs, C. M., and John, J. A. (1990). “Construction of Block Designs Using Mathematical Programming.” *Journal of the Royal Statistical Society, Series B* 52:497–503.
- Wolfe, P. (1982). “Checking the Calculation of Gradients.” *ACM Transactions on Mathematical Software* 8:337–343.

Subject Index

- active set methods, 606
 - quadratic programming, 569, 595
- Bard function, 637
- BFGS update method, 570
- boundary constraints
 - NLP procedure, 572
- Cholesky factor, 598
- COBYLA algorithm, 600, 605
- computational problems
 - NLP procedure, 610–612
- computational resources
 - NLP procedure, 627, 628
- confidence intervals, 583
 - output options, 622
 - profile confidence limits, 555
- conjugate-descent update method, 570
- conjugate gradient methods, 569, 599
- covariance matrix, 555, 613, 621
 - displaying, 566
- crossproduct Jacobian matrix, 573, 621
 - definition, 590
 - displaying, 566
 - saving, 565
- derivatives, 590
 - computing, 539, 591
 - finite differences, 601
- DFP update method, 570
- displayed output
 - NLP procedure, 624, 625
- double dogleg method, 569, 599
- dual BFGS update method, 570
- dual DFP update method, 570
- examples, *see* NLP examples
- feasible region, 589
- feasible solution, 589
- finite-difference approximations
 - central differences, 602
 - computation of, 557
 - forward differences, 602
 - NLP procedure, 556, 557
 - second-order derivatives, 557
- first-order conditions
 - local minimum, 590
- Fletcher-Reeves update method, 570
- function convergence
 - NLP procedure, 553
- functional summary
 - NLP procedure, 550
- global solution, 589
- Goldstein conditions, 597, 600, 608
- gradient vector
 - checking correctness, 604
 - convergence, 554
 - definition, 590
 - local optimality conditions, 589
 - projected gradient, 607
 - specifying, 574
- grid points, 554, 564, 565
- Hessian matrix, 621
 - definition, 590
 - displaying, 567
 - finite-difference approximations, 557, 601
 - initial estimate, 560
 - local optimality conditions, 589
 - projected, 607
 - saving, 565
 - scaling, 559, 603
 - specifying, 574
 - update method, 570
- hybrid quasi-Newton methods, 569, 570, 601
- input data sets
 - NLP procedure, 539, 616, 617
- intermediate variable, 634
- Jacobian matrix, 621
 - constraint functions, 575
 - definition, 590
 - displaying, 567
 - objective functions, 576
 - saving, 565, 566
 - scaling, 603
- Karush-Kuhn-Tucker conditions, 589, 590, 606
- Kuhn-Tucker conditions, *see* Karush-Kuhn-Tucker conditions
- labels
 - assigning to decision variables, 577
- Lagrange multipliers, 589, 607, 621
- Lagrangian function, 589

- least squares problems
 - definition of, 538
 - optimization algorithms, 594
- Levenberg-Marquardt minimization, 569
 - least squares method, 601
- line-search methods, 561, 608
 - step length, 555, 563
- linear complementarity problem, 569
 - quadratic programming, 595
- linear constraints
 - NLP procedure, 577
- linearly constrained optimization, 597
- local minimum
 - first-order conditions, 590
 - second-order conditions, 590
- local solution, 589

- matrix
 - definition (NLP), 578, 579
- maximum likelihood Weibull estimation
 - using PROC NLP, 649
- merit function, 598
- migration to PROC OPTMODEL
 - from PROC NLP, 629
- missing values
 - NLP procedure, 564, 626
- Moore-Penrose conditions, 615

- Nelder-Mead simplex method, 569, 600
- Newton-Raphson method, 569
 - with line search, 596
 - with ridging, 596
- NLP examples, 637
 - approximate standard errors, 644
 - Bard function, 637
 - blending problem, 655
 - boundary constraints, 542
 - chemical equilibrium, 664
 - covariance matrix, 644
 - Hock and Schittkowski problem, 641
 - introductory examples, 540–543, 545, 547
 - least squares problem, 541, 637
 - linear constraints, 543, 641
 - maximum likelihood Weibull estimation, 649
 - maximum-likelihood estimates, 547, 645
 - migration to PROC OPTMODEL, 629
 - nonlinear constraints, 545
 - nonlinear network problem, 668
 - quadratic programming problem, 639
 - restarting an optimization, 643
 - Rosenbrock function, 643
 - starting point, 641
 - statistical analysis, 644
 - trust region method, 643
 - unconstrained optimization, 540
- NLP procedure
 - active set methods, 595, 606
 - boundary constraints, 572
 - choosing an optimization algorithm, 593
 - computational problems, 610–612
 - computational resources, 627, 628
 - conjugate gradient methods, 599
 - convergence difficulties, 611, 612
 - convergence status, 626
 - covariance matrix, 555, 566, 613, 615, 621
 - crossproduct Jacobian, 590, 621
 - debugging options, 562, 585
 - derivatives, 590
 - display function values, 566, 567
 - displayed output, 566, 624, 625
 - double dogleg method, 599
 - eigenvalue tolerance, 555
 - feasible region, 589
 - feasible solution, 589
 - feasible starting point, 608
 - finite-difference approximations, 556, 557, 601
 - first-order conditions, 590
 - function convergence, 553, 556
 - functional summary, 550
 - global solution, 589
 - Goldstein conditions, 597, 600, 608
 - gradient, 558, 590
 - gradient convergence, 554, 558
 - grid points, 554
 - Hessian, 560, 590, 601, 621
 - Hessian scaling, 559, 603
 - Hessian update method, 570
 - initial values, 560, 567, 616
 - input data sets, 539, 616, 617
 - iteration history, 624
 - Jacobian, 590, 621
 - Karush-Kuhn-Tucker conditions, 589, 590
 - Lagrange multipliers, 589, 621
 - Lagrangian function, 589
 - least squares problems, 594
 - Levenberg-Marquardt method, 601
 - limiting function calls, 562
 - limiting number of iterations, 563
 - line-search methods, 561, 608
 - linear complementarity, 595
 - linear constraints, 577, 597
 - local optimality conditions, 589
 - local solution, 589
 - memory limit, 629
 - memory requirements, 592
 - missing values, 564, 626
 - Nelder-Mead simplex method, 600
 - Newton-Raphson method, 596

- nonlinear constraints, 573, 598
- optimality criteria, 589
- optimization algorithms, 569, 591
- optimization history, 567
- options classified by function, 550
- output data sets, 540, 564, 565, 618, 619, 623
- overview, 538
- parameter convergence, 554, 571
- precision, 554, 557, 603, 613
- predicted reduction convergence, 556
- profile confidence limits, 555
- program statements, 584
- projected gradient, 621
- projected Hessian matrix, 621
- quadratic programming, 594, 595, 598
- quasi-Newton methods, 597, 601
- rank of covariance matrix, 621
- restricting output, 567, 568
- restricting step length, 609
- second-order conditions, 590
- singularity criterion, 554, 564, 571
- stationary point, 612
- step length, 555
- storing model files, 623
- suppress printing, 564
- table of syntax elements, 550
- termination criteria, 605
- time limit, 563
- trust region method, 596
- TYPE variable, 616–618, 620, 622, 623
- unconstrained optimization, 597
- variables, 616
- nonlinear optimization, 538, *see* NLP procedure
 - algorithms, 591
 - computational problems, 610
 - conjugate gradient methods, 599
 - feasible starting point, 608
 - hybrid quasi-Newton methods, 601
 - Levenberg-Marquardt method, 601
 - Nelder-Mead simplex method, 600
 - Newton-Raphson method with line search, 596
 - Newton-Raphson method with ridging, 596
 - nonlinear constraints, 573, 582, 598
 - optimization algorithms, 594
 - quasi-Newton method, 597
 - trust region method, 596
- objective function
 - NLP procedure, 538, 548, 580
- optimality criteria, 589
- optimization
 - double dogleg method, 599
 - linear constraints, 597
 - nonlinear constraints, 598
 - unconstrained, 597
- optimization algorithms
 - least squares problems, 594
 - NLP procedure, 569, 591
 - nonlinear optimization, 594
 - quadratic programming, 594, 595
- options classified by function, *see* functional summary
- output data sets
 - NLP procedure, 540, 564, 565, 618, 619, 623
- overview
 - NLP procedure, 538
- Polak-Ribiere update method, 570
- Powell-Beale update method, 570
- precision
 - nonlinear constraints, 554, 603
 - objective function, 557, 603
- profile confidence limits, 583
 - parameters for, 555
- program statements
 - NLP procedure, 584
- projected gradient, 607, 621
- projected Hessian matrix, 607, 621
- quadratic programming, 560, 598
 - active set methods, 595
 - definition, 538
 - linear complementarity problem, 595
 - optimization algorithms, 594, 595
 - specifying the objective function, 580
- quasi-Newton methods, 569, 570, 597
- random numbers
 - seed, 568
- Rosenbrock function, 540, 573–575, 577
- Rosen-Suzuki problem, 583
- second-order conditions
 - local minimum, 590
- second-order derivatives
 - finite-difference approximations, 557
- singularity, 568, 615
 - absolute singularity criterion, 554
 - relative singularity criterion, 564, 571
- standard errors
 - computing, 567
- step length, 609
- syntax skeleton
 - NLP procedure, 550
- table of syntax elements, *see* functional summary
- termination criteria, 605
 - absolute function convergence, 553
 - absolute gradient convergence, 554
 - absolute parameter convergence, 554

number of function calls, 562

number of iterations, 563

predicted reduction convergence, 556

relative function convergence, 556

relative gradient convergence, 558

relative parameter convergence, 571

time limit, 563

trust region methods, 569

TYPE variable

NLP procedure, 616–618, 620, 622, 623

unconstrained optimization, 597

VF02AD algorithm, 598

VMCWD algorithm, 598

Wald confidence limits, 583, 584

Syntax Index

- ABORT statement
 - NLP program statements, 585
- ABSCONV= option
 - PROC NLP statement, 553
- ABSFCONV= option
 - PROC NLP statement, 553
- ABSFTOL= option, *see* ABSFCONV= option
- ABSGCONV= option
 - PROC NLP statement, 554, 598, 605, 613
- ABSGTOL= option, *see* ABSGCONV= option
- ABSTOL= option, *see* ABSCONV= option
- ABSXCONV= option
 - PROC NLP statement, 554
- ABSXTOL= option, *see* ABSXCONV= option
- ACTBC keyword
 - TYPE variable (NLP), 620
- ALL keyword
 - FDINT= option (NLP), 557
- ALL option, *see* PALL option
- ARRAY statement
 - NLP procedure, 571
- ASING= option, *see* ASINGULAR= option
- ASINGULAR= option
 - PROC NLP statement, 554, 615
- BEST= option
 - PROC NLP statement, 554, 574
- BFGS keyword
 - UPDATE= option (NLP), 570, 597, 598
- BOTH keyword
 - CLPARM= option (NLP), 555
- BOUNDS statement
 - NLP procedure, 572, 582, 595
- BY statement
 - NLP procedure, 572
- CD keyword
 - UPDATE= option (NLP), 570, 599
- CDIGITS= option
 - PROC NLP statement, 554, 603
- CENTRAL keyword
 - FD= option (NLP), 556
 - FDHESSIAN= option (NLP), 557
- CHI keyword
 - FORCHI= option (NLP), 584
- CLPARM= option
 - PROC NLP statement, 555, 622
- CON keyword
 - FDINT= option (NLP), 557
- CONGRA keyword
 - TECH= option (NLP), 569, 594, 595, 599, 600, 605
- CONST keyword
 - TYPE variable (NLP), 617
- COV= option
 - PROC NLP statement, 555, 613
- COVARIANCE= option, *see* COV= option
- COVRANK keyword
 - TYPE variable (NLP), 621
- COVSING= option
 - PROC NLP statement, 555, 615
- COV_x keyword
 - TYPE variable (NLP), 621
- CRPJAC keyword
 - TYPE variable (NLP), 621
- CRPJAC statement
 - NLP procedure, 573, 593, 603
- DAMPSTEP= option
 - PROC NLP statement, 555, 609, 610
- DATA= option
 - PROC NLP statement, 555, 616
- DBFGS keyword
 - UPDATE= option (NLP), 570, 597–599, 601
- DBLDOG keyword
 - TECH= option (NLP), 569, 594, 599, 610
- DDFP keyword
 - UPDATE= option (NLP), 570, 597–599, 601
- DECVAR statement
 - NLP procedure, 573, 612
- DESCENDING option
 - BY statement (NLP), 572
- DETAIL keyword
 - GRADCHECK= option (NLP), 558, 604
- DETERMIN keyword
 - TYPE variable (NLP), 621
- DF keyword
 - VARDEF= option (NLP), 570
- DFP keyword
 - UPDATE= option (NLP), 570, 597, 598
- DIAHES option
 - PROC NLP statement, 555, 573, 575
- DO statement
 - NLP program statements, 585
- DS=option, *see* DAMPSTEP= option
- EQ keyword
 - TYPE variable (NLP), 617, 618, 620

ESTDATA= option, *see* INEST= option
 EVERYOBS option
 NLINCON statement (NLP), 582

 F keyword
 FORCHI= option (NLP), 584
 FAST keyword
 GRADCHECK= option (NLP), 558, 604
 FCONV2= option
 PROC NLP statement, 556
 FCONV= option
 PROC NLP statement, 556
 FCONV2= option
 PROC NLP statement, 598
 FD= option
 PROC NLP statement, 556, 601, 604
 FDH= option, *see* FDHESSIAN= option
 FDHES= option, *see* FDHESSIAN= option
 FDHESSIAN= option
 PROC NLP statement, 557, 601, 604
 FDIGITS= option
 PROC NLP statement, 557, 603
 FDINT= option
 PROC NLP statement, 557, 603, 611
 FEASRATIO= option
 PROFILE statement (NLP), 584
 FFACTOR= option
 PROFILE statement (NLP), 584
 FORCHI= option
 PROFILE statement (NLP), 584
 FORWARD keyword
 FD= option (NLP), 556
 FDHESSIAN= option (NLP), 557
 FR keyword
 UPDATE= option (NLP), 570, 599
 FSIZE= option
 PROC NLP statement, 558
 FTOL2= option, *see* FCONV2= option
 FTOL= option, *see* FCONV= option

 G4= option
 PROC NLP statement, 558, 615
 GC= option, *see* GRADCHECK= option
 GCONV2= option
 PROC NLP statement, 558
 GCONV= option
 PROC NLP statement, 558, 598, 605, 613
 GE keyword
 TYPE variable (NLP), 617, 620
 GRAD keyword
 TYPE variable (NLP), 620
 GRADCHECK= option
 PROC NLP statement, 558, 604
 GRADIENT statement
 NLP procedure, 574, 577, 593, 603, 604
 GRIDPNT keyword
 TYPE variable (NLP), 620
 GTOL2= option, *see* GCONV2= option
 GTOL= option, *see* GCONV= option

 HESCAL= option
 PROC NLP statement, 559, 603
 HESSIAN keyword
 TYPE variable (NLP), 621
 HESSIAN statement
 NLP procedure, 574, 593, 603
 HS= option, *see* HESCAL= option
 HYQUAN keyword
 TECH= option (NLP), 569, 594, 601, 609

 IFP option, *see* INFEASIBLE option
 INCLUDE statement
 NLP procedure, 564, 575, 623
 INEST= option
 PROC NLP statement, 559, 577, 595, 616, 617, 622
 INFEASIBLE option
 PROC NLP statement, 559, 574
 INHES= option, *see* INHESSIAN= option
 INHESSIAN= option
 PROC NLP statement, 560, 599, 611
 INITIAL keyword
 TYPE variable (NLP), 620
 INITIAL= option
 PROC NLP statement, 560
 INQUAD= option
 PROC NLP statement, 560, 595, 617, 618
 INSTEP= option
 PROC NLP statement, 560, 598, 610, 611
 INVAR= option, *see* INEST= option

 JACNLC statement
 NLP procedure, 575, 593
 JACOBIAN keyword
 TYPE variable (NLP), 621
 JACOBIAN statement
 NLP procedure, 576, 593, 603, 604

 LABEL statement
 NLP procedure, 577
 LAGRANGE keyword
 TYPE variable (NLP), 621
 LB keyword
 TYPE variable (NLP), 616, 617, 620
 LCD= option, *see* LCDEACT= option
 LCDEACT= option
 PROC NLP statement, 561, 599, 607
 LCE= option, *see* LCEPSILON= option
 LCEPS= option, *see* LCEPSILON= option

LCEPSILON= option
 PROC NLP statement, 561, 596, 599, 606
 LCS= option, *see* LCSINGULAR= option
 LCSING= option, *see* LCSINGULAR= option
 LCSINGULAR= option
 PROC NLP statement, 561, 599, 607
 LE keyword
 TYPE variable (NLP), 617, 620
 LEVMAR keyword
 TECH= option (NLP), 569, 594, 601, 610
 LICOMP keyword
 TECH= option (NLP), 569, 594, 595, 617
 LINCON statement
 NLP procedure, 595, 600
 LINEAR keyword
 TYPE variable (NLP), 617, 618
 LINESEARCH= option
 PROC NLP statement, 561, 600, 601, 608
 LIS= option, *see* LINESEARCH= option
 LIST option
 PROC NLP statement, 562, 625
 LISTCODE option
 PROC NLP statement, 562, 625
 LOWERBD keyword
 TYPE variable (NLP), 616, 617, 620
 LSP= option, *see* LSPRECISION= option
 LSPRECISION= option
 PROC NLP statement, 562, 609
 LSQ statement
 NLP procedure, 580

 MATRIX statement
 NLP procedure, 578
 MAX statement
 NLP procedure, 580
 MAXFU= option, *see* MAXFUNC= option
 MAXFUNC= option
 PROC NLP statement, 562, 627
 MAXIT= option, *see* MAXITER= option
 MAXITER= option
 PROC NLP statement, 563, 627
 MAXQUAD statement
 NLP procedure, 580, 595
 MAXSTEP= option
 PROC NLP statement, 563, 610
 MAXTIME= option
 PROC NLP statement, 563, 627
 MIN statement
 NLP procedure, 580
 MINIT= option, *see* MINITER= option
 MINITER= option
 PROC NLP statement, 563
 MINQUAD statement
 NLP procedure, 580, 595

 MOD= option, *see* MODEL= option
 MODEL= option
 PROC NLP statement, 564, 623
 MODFILE= option, *see* MODEL= option
 MSING= option, *see* MSINGULAR= option
 MSINGULAR= option
 PROC NLP statement, 564, 615

 N keyword
 VARDEF= option (NLP), 570
 NACTBC keyword
 TYPE variable (NLP), 620
 NACTLC keyword
 TYPE variable (NLP), 620
 NEIGNEG keyword
 TYPE variable (NLP), 621
 NEIGPOS keyword
 TYPE variable (NLP), 621
 NEIGZER keyword
 TYPE variable (NLP), 621
 NEWRAP keyword
 TECH= option (NLP), 569, 594, 596
 NLC statement, *see* NLINCON statement
 NLDACTLC keyword
 TYPE variable (NLP), 620
 NLINCON statement
 NLP procedure, 582, 600
 NLP procedure
 ARRAY statement, 571
 BOUNDS statement, 572, 582, 595
 BY statement, 572
 CRPJAC statement, 573, 593, 603
 DECVAR statement, 573
 GRADIENT statement, 574, 577, 593, 603, 604
 HESSIAN statement, 574, 593, 603
 INCLUDE statement, 564, 575, 623
 JACNLC statement, 575, 593
 JACOBIAN statement, 576, 593, 603, 604
 LABEL statement, 577
 LINCON statement, 577, 595, 600
 LSQ statement, 580
 MATRIX statement, 578
 MAX statement, 580
 MAXQUAD statement, 580, 595
 MIN statement, 580
 MINQUAD statement, 580, 595
 NLINCON statement, 582, 600
 PROC NLP statement, 553
 PROFILE statement, 583, 622
 NMSIMP keyword
 TECH= option, 600
 TECH= option (NLP), 569, 594, 595
 NOBS keyword
 TYPE variable (NLP), 620

NOEIGNUM option
 PROC NLP statement, 564
 NOMISS option
 PROC NLP statement, 564, 616, 626
 NONE keyword
 GRADCHECK= option (NLP), 558
 TECH= option (NLP), 569
 NOP option, *see* NOPRINT option
 NOPRINT option
 PROC NLP statement, 564, 625
 NOTSORTED option
 BY statement (NLP), 572
 NRRIDG keyword
 TECH= option (NLP), 569, 594, 596

 OBJ keyword
 FDINT= option (NLP), 557
 OPTCHECK= option
 PROC NLP statement, 564, 612
 OTHERWISE statement
 NLP program statements, 586
 OUT= option
 PROC NLP statement, 564, 618
 OUTALL option
 PROC NLP statement, 565
 OUTCRPJAC option
 PROC NLP statement, 565
 OUTDER= option
 PROC NLP statement, 565, 618
 OUTEST= option
 PROC NLP statement, 565, 577, 619, 622
 OUTGRID option
 PROC NLP statement, 565
 OUTHES option, *see* OUTHESIAN option
 OUTHESIAN option
 PROC NLP statement, 565
 OUTITER option
 PROC NLP statement, 565
 OUTJAC option
 PROC NLP statement, 565
 OUTM= option, *see* OUTMODEL= option
 OUTMOD= option, *see* OUTMODEL= option
 OUTMODEL= option
 PROC NLP statement, 565, 575, 623
 OUTNCJAC option
 PROC NLP statement, 566
 OUTTABLE option
 PROFILE statement (NLP), 584
 OUTTIME option
 PROC NLP statement, 566
 OUTVAR= option, *see* OUTEST= option

 PALL option
 PROC NLP statement, 566, 625

 PARAMETERS statement, *see* DECVAR statement
 PARS keyword
 TYPE variable (NLP), 616, 617, 620
 PARS statement, *see* DECVAR statement
 PB keyword
 UPDATE= option (NLP), 570, 599
 PCOV option
 PROC NLP statement, 566, 625
 PCRPJAC option
 PROC NLP statement, 566, 625
 PEIGVAL option
 PROC NLP statement, 566, 615, 625
 PERROR option
 PROC NLP statement, 566
 PFUNCTION option
 PROC NLP statement, 566, 625
 PGRID option
 PROC NLP statement, 567, 625
 PHES option, *see* PHESSIAN option
 PHESSIAN option
 PROC NLP statement, 567, 625
 PHIS option, *see* PHISTORY option
 PHISTORY option
 PROC NLP statement, 567, 625
 PIN option, *see* PINIT option
 PINIT option
 PROC NLP statement, 567, 625
 PJAC option, *see* PJACOBI option
 PJACOBI option
 PROC NLP statement, 567, 625
 PJTJ option, *see* PCRPJAC option
 PL keyword
 CLPARG= option (NLP), 555
 PL_CL keyword
 TYPE variable (NLP), 623
 PLC_LOW keyword
 TYPE variable (NLP), 622
 PLC_UPP keyword
 TYPE variable (NLP), 622
 PNLJAC option
 PROC NLP statement, 567, 625
 PR keyword
 UPDATE= option (NLP), 570, 599
 PROC NLP statement, *see* NLP procedure
 statement options, 553
 PROFILE keyword
 TYPE variable (NLP), 623
 PROFILE statement
 NLP procedure, 583, 622
 PROJCRPJ keyword
 TYPE variable (NLP), 621
 PROJGRAD keyword
 TYPE variable (NLP), 621
 PROJHES keyword

TYPE variable (NLP), 621
 PSH option, *see* PSHORT option
 PSHORT option
 PROC NLP statement, 567, 625
 PSTDERR option
 PROC NLP statement, 567, 625
 PSUMMARY option
 PROC NLP statement, 568, 625
 PTIME option
 PROC NLP statement, 568
 PUT statement
 NLP program statements, 585

 QUAD keyword
 TYPE variable (NLP), 617, 618
 QUADAS keyword
 TECH= option (NLP), 569, 594, 595, 617
 QUANEW keyword
 TECH= option (NLP), 569, 594, 597, 598, 605, 611

 RANDOM= option
 PROC NLP statement, 568
 REST= option, *see* RESTART= option
 RESTART= option
 PROC NLP statement, 568

 SE option, *see* PSTDERR option
 SELECT statement
 NLP program statements, 586
 SHORT option, *see* PSHORT option
 SIGSQ keyword
 TYPE variable (NLP), 621
 SIGSQ= option
 PROC NLP statement, 568, 614
 SING= option, *see* SINGULAR= option
 SINGULAR= option
 PROC NLP statement, 568
 STDERR keyword
 TYPE variable (NLP), 620
 STDERR option, *see* PSTDERR option
 SUM option
 PSUMMARY option, 568
 SUMMARY option, *see* PSUMMARY option
 SUMOBS option
 NLINCON statement (NLP), 582

 TECH= option
 PROC NLP statement, 569, 612
 TECHNIQUE= option, *see* TECH= option
 TERMINAT keyword
 TYPE variable (NLP), 621
 TIME keyword
 TYPE variable (NLP), 621
 TRUREG keyword

 TECH= option (NLP), 569, 594, 596, 610

 UB keyword
 TYPE variable (NLP), 616, 617, 620
 UPD= option, *see* UPDATE= option
 UPDATE= option
 PROC NLP statement, 570, 596, 597
 UPPERBD keyword
 TYPE variable (NLP), 616, 617, 620

 VAR statement, *see* DECVAR statement
 VARDEF= option
 PROC NLP statement, 570, 613
 VERSION= option
 PROC NLP statement, 570, 598, 601, 611
 VS= option, *see* VERSION= option
 VSING= option, *see* VSINGULAR= option
 VSINGULAR= option
 PROC NLP statement, 571, 615

 WALD keyword
 CLPARM= option (NLP), 555
 WALD_CL keyword
 TYPE variable (NLP), 622
 WHEN statement
 NLP program statements, 586

 XCONV= option
 PROC NLP statement, 571
 XSIZE= option
 PROC NLP statement, 571
 XTOL= option, *see* XCONV= option