USER'S GUIDE FOR SNOPT 5.3: A FORTRAN PACKAGE FOR LARGE-SCALE NONLINEAR PROGRAMMING

Philip E. GILL
Department of Mathematics
University of California, San Diego
La Jolla, California 92093-0112

Walter MURRAY and Michael A. SAUNDERS
Systems Optimization Laboratory
Department of EESOR
Stanford University
Stanford, California 94305-4023

DRAFT, October 1997

Abstract

SNOPT is a set of Fortran subroutines for minimizing a smooth function subject to constraints, which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints. SNOPT is a general-purpose optimizer, designed to find locally optimal solutions for models involving smooth nonlinear functions. They are often more widely useful. (For example, local optima are often global solutions, and discontinuities in the function gradients can often be tolerated if they are not too close to an optimum.) Ideally, users should provide gradients. Unknown components are estimated by finite differences.

SNOPT incorporates a sequential quadratic programming (SQP) method that obtains search directions from a sequence of quadratic programming subproblems. Each QP subproblem minimizes a quadratic model of a certain Lagrangian function subject to a linearization of the constraints. An augmented Lagrangian merit function is reduced along each search direction to ensure convergence from any starting point.

SNOPT is most efficient if only some of the variables enter nonlinearly, or if the number of active constraints (including simple bounds) is nearly as large as the number of variables. SNOPT requires relatively few evaluations of the problem functions. Hence it is especially effective if the objective or constraint functions are expensive to evaluate.

The source code for SNOPT is suitable for any machine with a reasonable amount of memory and a Fortran compiler. SNOPT may be called from a driver program (typically in Fortran, C or MATLAB).

Keywords: Nonlinear programming, constrained optimization, nonlinear constraints, SQP methods, limited-storage quasi-Newton updates, Fortran software.

Contents

1. Introduction

SNOPT is a collection of Fortran 77 subroutines for solving the *nonlinear programming problem*, which is assumed to be stated in following form:

$$\boxed{\begin{array}{ll} \text{NP} & \underset{x}{\text{minimize}} \quad f(x) \\ & \text{subject to} \quad l \leq \begin{pmatrix} x \\ F(x) \\ Gx \end{pmatrix} \leq u, \end{array}} \tag{1.1}$$

where l and u are constant lower and upper bounds, f is a smooth scalar objective function, G is a sparse matrix, and $F(x)$ is a vector of smooth nonlinear constraint functions $\{F_i(x)\}$. (The optional parameter `maximize` may be used to specify a problem in which f is maximized instead of minimized.) If all constraints are linear, F is absent and the problem is said to be *linearly constrained*. In general, the objective and constraint functions are *structured* in the sense that they are formed from linear combinations of linear and nonlinear functions. This structure can be exploited by SNOPT during the solution process (see §3 below).

Ideally, the first derivatives (gradients) of f and F_i should be known and coded by the user. If only some gradients are known, SNOPT will estimate the missing ones with finite differences.

Note that upper and lower bounds are specified for all variables and constraints. This form allows full generality in specifying various types of constraint. In particular, the jth constraint may be defined as an *equality* by setting $l_j = u_j$. If certain bounds are not present, the associated elements of l or u may be set to special values that are treated as $-\infty$ or $+\infty$.

1.1. Subroutines

SNOPT is accessed via the following routines:

`snopt` (§4) The top-level routine, called by the user.

`funcon, funobj` (§5) Called by SNOPT. Supplied by the user. Define the constraint functions $F(x)$ and objective $f(x)$ for given vectors x.

`snInit` (§7.1) *MUST be called before the first call to SNOPT*. Called by the user to initialize default parameters and read a SPECS file (if any).

`snprm, snprmi, snprmr` (§7.1) Called by the user to input a single optional parameter.

1.2. Files

SNOPT reads or creates certain external files. The most important of these are described in the following.

SPECS file. If present, this is input by calling `snInit`.

Summary file. Intended for output to the screen in an interactive environment. It contains error messages and a brief iteration log, or may be suppressed.

Print file. Intended for a permanent file. It contains error messages, a more detailed iteration log, and optionally the printed solution.

Unit numbers for for the `specs`, `print` and `summary` files are defined at compile time; typically they will be 5, 6 and 9, but they depend upon the installation. For a more detailed description of the files that can be created by SNOPT, see §9.

2. Description of the method

Here we briefly describe the main features of the SQP algorithm used in SNOPT and introduce some terminology used in the description of the subroutine and its arguments.

The upper and lower bounds on the m components of F and Gx are said to define the *general constraints* of the problem. SNOPT converts the general constraints to equalities by introducing a set of *slack variables* s, where $s = (s_1, s_2, \ldots, s_m)^T$. For example, the linear constraint $5 \leq 2x_1 + 3x_2 \leq +\infty$ is replaced by $2x_1 + 3x_2 - s_1 = 0$ together with the bounded slack $5 \leq s_1 \leq +\infty$. The problem defined by (1.1) can therefore be rewritten in the following equivalent form

$$\begin{array}{ll} \underset{x,s}{\text{minimize}} & f(x) \\ \text{subject to} & \begin{pmatrix} F(x) \\ Gx \end{pmatrix} - s = 0, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u. \end{array}$$

The linear and nonlinear general constraints become equalities of the form $F(x) - s_N = 0$ and $Gx - s_L = 0$, where s_L and s_N are known as the *linear* and *nonlinear* slacks.

The basic structure of SNOPT involves *major* and *minor* iterations. The major iterations generate a sequence of iterates (x_k) that satisfy the linear constraints and converge to a point that satisfies the first-order conditions for optimality. At each iterate a QP subproblem is used to generate a search direction towards the next iterate (x_{k+1}). The constraints of the subproblem are formed from the linear constraints $Gx - s_L = 0$ and the nonlinear constraint linearization:

$$F(x_k) + F'(x_k)(x - x_k) - s_N = 0,$$

where $F'(x_k)$ denotes the Jacobian matrix of first derivatives of $F(x_k)$ evaluated at x_k. The QP constraints therefore comprise the m linear constraints

$$\begin{array}{rrrl} F'(x_k)x & -s_N & = & -F(x_k) + F'(x_k)x_k, \\ Gx & -s_L & = & 0, \end{array}$$

where x and s are bounded above and below by u and l as before. If the $m \times n$ matrix A and m-vector b are defined as

$$A = \begin{pmatrix} F'(x_k) \\ G \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} -F(x_k) + F'(x_k)x_k \\ 0 \end{pmatrix},$$

then the QP subproblem can be written as

$$\underset{x,s}{\text{minimize}} \;\; q(x) \quad \text{subject to} \;\; Ax - s = b, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u,$$

where $q(x)$ is a quadratic approximation to the modified Lagrangian function (see later). The linear constraint matrix A above is input to SNOPT by means of the arguments `a(*)`, `ha(*)`, and `ka(*)` (see Section 4). Access to this matrix allows the user to specify the pattern of nonzero elements in $F'(x)$ and G, and identify any nonzero elements that remain constant throughout the minimization.

Solving the QP subproblem is itself an iterative procedure, with the *minor* iterations of an SQP method being the iterations of the QP method. At each minor iteration, the constraints $Ax - s = b$ are (conceptually) partitioned into the form

$$Bx_B + Sx_S + Nx_N = b,$$

where the *basis matrix* B is square and nonsingular. The elements of x_B x_S and x_N are called the *basic*, *superbasic* and *nonbasic* variables respectively; they are a permutation of the elements of x and s. At a QP solution, the basic and superbasic variables will lie somewhere between their bounds, while the nonbasic variables will be equal to one of their upepr or lower bounds. At each iteration, x_S is regarded as a set of independent variables that are free to move in any desired direction, namely one that will improve the value of the QP objective (or sum of infeasibilities). The basis variables are then adjusted in order to ensure that that (x, s) continues to satisfy $Ax - s = b$. The number of superbasic variables (n_S say) therefore indicates the number of degrees of freedom remaining after the constraints have been satisfied. In broad terms, n_S is a measure of *how nonlinear* the problem is. In particular, n_S will always be zero for LP problems.

If it appears that no improvement can be made with the current definition of B, S and N, a nonbasic variable is selected to be added to S, and the process is repeated with the value of n_S increased by one. At all stages, if a basic or superbasic variables encounters one of its bounds, the variables is made nonbasic and the value of n_S is decreased by one.

Associated with each of the m equality constraints $Ax - s = b$ are the *dual variables* π. Similarly, each variable in (x, s) has an associated *reduced gradient* d_j. The reduced gradients for the variables x are the quantities $g - A^T\pi$, where g is the gradient of the QP objective; and the reduced gradients for the slacks are the dual variables π. The QP subproblem is optimal if $d_j \geq 0$ for all nonbasic variables at their lower bounds, $d_j \leq 0$ for all nonbasic variables at their upper bounds, and $d_j = 0$ for superbasic variables. In practice, an *approximate* QP solution is found by relaxing these conditions on d_j (see the `Minor optimality tolerance` described in §7.3).

If $\widehat{x}_k$, $\widehat{s}_k$ and $\widehat{\pi}_k$ denote the optimal values of x, s and π from the QP subproblem, new estimates of the NP solution are computed using a linesearch on the augmented Lagrangian merit function

$$\mathcal{M}(x, \pi, s) = f(x) - \pi^T\big(F(x) - s_N\big) + \tfrac{1}{2}\big(F(x) - s_N\big)^T D\big(F(x) - s_N\big), \tag{2.1}$$

where D is a diagonal matrix of penalty parameters. If $\widehat{x}_k$, $\widehat{s}_k$ and $\widehat{\pi}_k$ denote the optimal values of x, s and π from the QP subproblem, and (x_k, s_k, π_k) are the current estimates of the optimal values (x^*, s^*, π^*), then the linesearch determines a step α_k $(0 < \alpha_k \leq 1)$ such that the new point

$$\begin{pmatrix} x_{k+1} \\ \pi_{k+1} \\ s_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ \pi_k \\ s_k \end{pmatrix} + \alpha_k \begin{pmatrix} \widehat{x}_k - x_k \\ \widehat{\pi}_k - \pi_k \\ \widehat{s}_k - s_k \end{pmatrix} \tag{2.2}$$

gives a *sufficient decrease* in the merit function (2.1). When necessary, the penalties in D are increased by the minimum-norm perturbation that ensures descent for $\mathcal{M}$ [6]. As in NPSOL, s_N is adjusted to minimize the merit function as a function of s prior to the solution of the QP subproblem. For more details, see [5, 2].

2.1. Treatment of constraint infeasibilities

SNOPT makes explicit allowance for infeasible constraints. Infeasible linear constraints are detected first by solving a problem of the form

FLP	$\underset{x,v,w}{\text{minimize}}$	$e^T(v + w)$
	subject to	$l \leq \begin{pmatrix} x \\ Gx - v + w \end{pmatrix} \leq u, \quad v \geq 0, \quad w \geq 0,$

where e is a vector of ones. This is equivalent to minimizing the sum of the general linear constraint violations subject to the simple bounds. (In the linear programming literature, the approach is often called elastic programming.)

If the linear constraints are infeasible ($v \neq 0$ or $w \neq 0$), SNOPT terminates without computing the nonlinear functions.

If the linear constraints are feasible, all subsequent iterates satisfy the linear constraints. (Such a strategy allows linear constraints to be used to define a region in which f and c can be safely evaluated.) SNOPT proceeds to solve NP as given, using search directions obtained from a sequence of quadratic programming subproblems. Each QP subproblem minimizes a quadratic model of a certain Lagrangian function subject to linearized constraints. An augmented Lagrangian merit function is reduced along each search direction to ensure convergence from any starting point.

If a QP subproblem proves to be infeasible or unbounded (or if the dual variables π for the nonlinear constraints become large), SNOPT enters "elastic" mode and solves the problem

$$\text{NP}(\rho) \qquad \underset{x,v,w}{\text{minimize}} \quad f(x) + \rho e^T(v+w)$$
$$\text{subject to} \quad l \leq \begin{pmatrix} x \\ F(x) - v + w \\ Gx \end{pmatrix} \leq u, \quad v \geq 0, \quad w \geq 0,$$

where ρ is a nonnegative parameter (the *elastic weight*), and $f(x) + \rho e^T(v+w)$ is called a *composite objective*. If ρ is sufficiently large, this is equivalent to minimizing the sum of the nonlinear constraint violations subject to the linear constraints and bounds. A similar ℓ_1 formulation of NP is fundamental to the Sℓ_1QP algorithm of Fletcher [3]. See also Conn [1].

References

[1] A. R. Conn, *Constrained optimization using a nondifferentiable penalty function*, SIAM J. Numer. Anal., 10 (1973), pp. 760–779.

[2] S. K. Eldersveld, *Large-scale sequential quadratic programming algorithms*, PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1991.

[3] R. Fletcher, *An ℓ_1 penalty method for nonlinear constraints*, in Numerical Optimization 1984, P. T. Boggs, R. H. Byrd, and R. B. Schnabel, eds., Philadelphia, 1985, SIAM, pp. 26–40.

[4] R. Fourer, *Solving staircase linear programs by the simplex method. 1: Inversion*, Math. Prog., 23 (1982), pp. 274–313.

[5] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *User's guide for NPSOL (Version 4.0): a Fortran package for nonlinear programming*, Report SOL 86-2, Department of Operations Research, Stanford University, Stanford, CA, 1986.

[6] ———, *Some theoretical properties of an augmented Lagrangian merit function*, in Advances in Optimization and Parallel Computing, P. M. Pardalos, ed., North Holland, North Holland, 1992, pp. 101–128.

3. Identifying structure in the objective and constraints

Consider the following nonlinear optimization problem with four variables (u, v, z, w):

$$\begin{array}{lll} \text{minimize} & (u+v+z)^2 + 3z + 5w & \\ \text{subject to} & u^2 + v^2 + z & = 2 \\ & u^4 + v^4 + w & = 4 \\ & 2u + 4v & \geq 0 \\ & z \geq 0 \quad w & \geq 0. \end{array}$$

This problem has several characteristics that can be exploited by SNOPT:

- The objective function is nonlinear, and it is the sum of a *nonlinear* function of the three variables (u, v, z), and a *linear* function of the variables (z, w).
- The first two constraints are nonlinear constraints. The third constraint is linear.
- Each nonlinear constraint function is the sum of a *nonlinear* function of the two variables (u, v), and a *linear* function of (z, w).

SNOPT allows `funobj` and `funcon` to be defined so that only the nonlinear variables are involved. This simplifies the coding of the routines and allows a smaller Jacobian to be evaluated at each step.

In the following, we will define this nonlinear structure for the simple problem given above.

For the objective function, the idea is to define the function $f = (u+v+z)^2$ that includes only the nonlinear part of the objective. The variables u, v, z associated with this function are known as *nonlinear objective variables*. The number of nonlinear objective variables is given by the `snopt` input argument `nnObj`. They are the only variables needed in the subroutine `funobj`, the linear part $3z + 5w$ of the objective is input to SNOPT as part of the linear constraint matrix A (see the parameter `iObj` below.)

If (x', y') denote the nonlinear and linear objective variables, then the objective may be written in the form

$$f(x') + c^T x' + d^T y',$$

where f is the nonlinear part of the objective and c, d are constant vectors.

The constraint functions are separated in a similar way by considering a vector function F that contains only the nonlinear terms of the nonlinear constraints. In the example above, F is a function of the two variables u and v, with $F_1 = u^2 + v^2$, and $F_2 = u^4 + v^4$. The variables (u, v) are known as the *nonlinear Jacobian variables*. The number of nonlinear constraint variables is specified by means of the `snopt` input argument `nnJac` below. They are the only variables needed in the user-supplied subroutine `funcon`.

As in the case of the objective function, the variables are partitioned as (x, y), where x defines the nonlinear constraint variables. In general x and x' have differing dimensions, as in the example above. The nonlinear objective variables and nonlinear Jacobian variables together comprise the *nonlinear variables*, so that the number of nonlinear variables will be the *larger* of the nonlinear objective variables and nonlinear Jacobian variables. When calling SNOPT the user implicitly orders the variables of the problem. This ordering determines other quantities such as the order of the columns of the constraint Jacobian. When `funcon` and `funobj` involve only nonlinear variables, it is important that the nonlinear variables occur first in `x(*)`. In particular, the constraint functions must be in the form

$$\begin{pmatrix} F(x) + A_2 y \\ A_3 x + A_4 y \end{pmatrix}, \tag{3.1}$$

which ensures that the Jacobian is of the form

$$A = \begin{pmatrix} J & A_2 \\ A_3 & A_4 \end{pmatrix},$$

with the Jacobian of F always appearing in the *top left-hand corner* of the full Jacobian. The (constant) linear terms A_2, A_3 and A_4 are input to SNOPT via the arguments `a(*)`, `ha(*)`, and `ka(*)` (see Section 4).

If there are more nonlinear objective variables than nonlinear constraint variables, the position of J in A implies that any nonlinear objective variables that are not nonlinear constraint variables should appear last. In the example above, this would imply the ordering $x_1 \equiv u$, $x_2 \equiv v$, $x_3 \equiv z$, and $x_4 \equiv w$.

The inequalities $l_1 \leq F(x)+A_2y \leq u_1$ and $l_2 \leq A_3x+A_4y \leq u_2$ implied by the constraint functions (3.1) are known as the *nonlinear* and *linear* constraints respectively. Together, these two sets of inequalities constitute the *general constraints*.

The constraint vector $F(x)$ (3.1) and (optionally) its partial derivative matrix $J(x)$ are set in subroutine `funcon` (see below). The matrices A_2, A_3, A_4 and the sparsity pattern of $J(x)$ are entered column-wise in the arrays `a`, `ha`, `ka` (below).

If all constraint gradients are known (`Derivative Level = 2` or `3`), any Jacobian elements that are constant may be given their correct values in the in the arrays `a`, `ha`, `ka`, and then they need not be reset by subroutine `funcon`. This includes values that are identically zero—such elements do not have to be specified in `funcon`. In other words, Jacobian elements are assumed to be zero unless specified otherwise.

4. Specification of subroutine snopt

Problem NP is solved by a call to subroutine `snopt`, whose parameters are defined here. Note that most machines use `double precision` declarations as shown, but some machines use `real`. The same applies to the user routines `funobj` and `funcon`.

```
      subroutine snopt ( start, m, n, ne, nName,
     $                   nnCon, nnObj, nnJac,
     $                   iObj, ObjAdd, Prob,
     $                   funcon, funobj,
     $                   a, ha, ka, bl, bu, Names,
     $                   hs, xs, pi, rc,
     $                   inform, mincw, miniw, minrw,
     $                   nS, nInf, sInf, Obj,
     $                   cu, lencu, iu, leniu, ru, lenru,
     $                   cw, lencw, iw, leniw, rw, lenrw )

      external            funcon, funobj
      character*(*)       start
      character*8         Prob
      character*8         Names(nName)
      integer             m, n, ne, nName, nnCon, nnObj, nnJac
      integer             iObj, nS, nInf
      integer             inform, mincw, miniw, minrw
      integer             ha(ne), hs(n+m)
      integer             ka(n+1)
      double precision    ObjAdd, sInf, Obj
      double precision    a(ne), bl(n+m), bu(n+m)
      double precision    xs(n+m), pi(m), rc(n+m)

      integer             lencu, lencw, leniu, lenru, leniw, lenrw
      character*8         cu(lencu), cw(lencw)
      integer             iu(leniu), iw(leniw)
      double precision    ru(lenru), rw(lenrw)
```

On entry:

`start` is a character string that specifies how a starting basis (and certain other items) are to be obtained.

- `'Cold'` requests that the Crash procedure be used to choose an initial basis, unless a basis file is provided via `OLD BASIS`, `INSERT` or `LOAD` in the Specs file.
- `'Basis file'` is the same as `start = 'Cold'` but is more meaningful when a basis file is given.
- `'Warm'` means that a basis is already defined in `hs` (probably from an earlier call).

`m` is m, the number of general constraints (`m` > 0). This is the number of rows in the full Jacobian matrix A.

Note that A must have at least one row. If your problem has no constraints, or only upper and lower bounds on the variables, then you must include a dummy row

with sufficiently wide upper and lower bounds. See the discussion of the parameters `a(ne)`, `ha(ne)`, and `ka(n+1)` below.

`n` is the number of variables, excluding slacks (`n` > 0). This is the number of columns in A.

`ne` is the number of nonzero entries in A (including the Jacobian for any nonlinear constraints) (`ne` > 0).

`nName` is the number of column and row names provided in the character array `Names`. If `nName` $= 1$, there are *no* names. Generic names will be used in the printed solution. Otherwise, `nName` $=$ `n` $+$ `m` and all names must be provided.

`nnCon` is m_1, the number of nonlinear constraints (`nnCon` ≥ 0).

`nnObj` is n_1', the number of nonlinear objective variables (`nnObj` ≥ 0).

`nnJac` is n_1'', the number of nonlinear Jacobian variables. If `nnCon` $= 0$, `nnJac` $= 0$. If `nnCon` > 0, `nnJac` > 0.

`iObj` says which row of A is a free row containing a linear objective vector c. If there is no such vector, `iobj` $= 0$. Otherwise, this row must come after any nonlinear rows, so that `nnCon` $<$ `iObj` $\leq m$.

`ObjAdd` is a constant that will be added to the objective for printing purposes. Typically `ObjAdd = 0.0d+0`.

`Prob` is an 8-character name for the problem. `Prob` is used in the printed solution and in some routines that print basis files.)

A blank name can be assigned if required.

`funcon` is the name of a subroutine that calculates the vector $F(x)$ of nonlinear constraint functions and (optionally) its Jacobian for a specified vector x. `funcon` must be declared as `external` in the routine that calls `snopt`. For a detailed description of `funcon`, see §5.2.

`funobj` is the name of a subroutine that calculates the objective function $f(x)$ and (optionally) its gradient for a specified vector x. `funobj` must be declared as `external` in the routine that calls `snopt`. For a detailed description of `funobj`, see §5.1.

`a(ne), ha(ne), ka(n+1)` define the nonzero elements of the constraint matrix A (Jacobian), stored column-wise. A pair of values (`a(i),ha(i)`) contains a Jacobian element and its corresponding row index. The array `ka(n+1)` defines a set of pointers to the beginning of each column of A within `a(*)` and `ha(*)`; i.e., `a(ka(j))` is the position in `a(*)` of the start of the elements of column `j`. It follows that the entries of column j are held in `a(ka(j):ka(j+1)-1)` and their corresponding row indices are held in `ha(ka(j):ka(j+1)-1)`.

Note: every element of `a(*)` must be assigned a value in the calling program. Elements in the nonlinear part of `a(*)` (see the Notes below) can be any dummy value (e.g., zero) since they are initialized by SNOPT at the first point that is feasible with respect to the linear constraints. The linear part of `a(*)` must contain the constant Jacobian elements.

If `Derivative level` $= 2$ or 3, the nonlinear part of `a(*)` can be used to define any constant Jacobian elements. These elements will remain unaltered during the solution process.

1. It is *essential* that `ka`$(1) = 1$ and `ka`$(n + 1) =$ `ne` $+ 1$.

2. To allocate storage, SNOPT needs to know if the Jacobian is dense or sparse. The default is dense. If this is not appropriate, define the optional parameter "`Jacobian Sparse`" in the `specs` file, or

```
call snprm ( 'Jacobian  Sparse', 0, 0, inform, ...
```

   before calling `snopt`.

3. If the problem has a nonlinear objective, the first `nnObj` columns of `a` and `ha` belong to the nonlinear objective variables. Subroutine `funobj` deals with these variables.

4. If the problem has nonlinear constraints, the first `nnJac` columns of `a` and `ha` belong to the nonlinear Jacobian variables, and the first `nnCon` rows of `a` and `ha` belong to the nonlinear constraints. Subroutine `funcon` deals with these variables and constraints.

5. If `nnObj` > 0 and `nnJac` > 0, the two sets of nonlinear variables overlap. The total number of nonlinear variables is `nnL` $= \max($`nnObj`$,$`nnJac`$)$.

6. The Jacobian forms the top left corner of `a` and `ha`. If a Jacobian column j ($1 \leq j \leq$ `nnJac`) contains any entries `a`(k), `ha`(k) associated with nonlinear constraints ($1 \leq$ `ha`$(k) \leq$ `nnCon`), those entries must come before any other (linear) entries.

7. The row indices `ha(k)` for a column may be in any order (subject to Jacobian entries appearing first). Subroutine `funcon` must define the Jacobian entries in the same order.

8. If your problem has no constraints, or only upper and lower bounds on the variables, then you can include a dummy "free" row with a single (zero) element. In this case, `ha`$(1) = 1$, `a`$(1) = 0.0$ and `ka` is defined such that `ka`$(1) = 1$, `ka`$(j) = 2$ for $j = 2$,3, ..., n.

`bl` contains the lower bounds on the variables and slacks (x, s).

The first n entries of `bl`, `bu`, `hs` and `xs` refer to the variables x. The last m entries refer to the slacks s.

To specify a non-existent lower bound ($l_j = -\infty$), set `bl`$(j) \leq -$`bigbnd`, where `bigbnd` is the `Infinite Bound`, whose default value is 10^{20}.

To fix the jth variable (say $x_j = \beta$, where $|\beta| <$ `bigbnd`), set `bl`$(j) =$ `bu`$(j) = \beta$.

To make the jth constraint an *equality* constraint (say $s_j = \beta$, where $|\beta| <$ `bigbnd`), set `bl`$(n + j) =$ `bu`$(n + j) = \beta$.

`bu` contains the upper bounds on (x, s). To specify a non-existent upper bound ($u_j = \infty$), set `bu`$(j) \geq$ `bigbnd`. For the data to be meaningful, it is required that `bl`$(j) \leq$ `bu`(j) for all j.

`Names(nName)` sometimes contains names for the variables and constraints. If `nName = 1`, `Names` are not used. The printed solution will use generic names for the columns and row. If `nName` $=$ `n` $+$ `m`, `Names`(j) should contain the 8 character name of the jth variable ($j = 1$, 2,..., `n` $+$ `m`). If $j = n + i$, the jth variable is the ith row.

`hs(n+m)` sometimes contains a set of initial states for each variable xs, or for each variable and slack (x, s). See the discussion of the parameter `xs`.

`xs(n+m)` sometimes contains a set of initial values for the variables and slacks (x, s).

1. If `start` = `'Cold'` or `'Basis file'`, and a BASIS file of some sort is to be input (an OLD BASIS file, INSERT file or LOAD file), then `hs` and `xs` need not be set at all.

2. Otherwise, `hs(1:n)` must be defined for a Cold start. If nothing special is known about the problem, or if there is no wish to provide special information, you may set $\texttt{hs}(j) = 0$, $\texttt{xs}(j) = 0$ for all $j = 1, 2, \ldots, n$. All variables will be eligible for the initial basis.

   Less trivially, to say that variable j will probably be equal to one of its bounds, set $\texttt{hs}(j) = 4$ and $\texttt{xs}(j) = \texttt{bl}(j)$ or $\texttt{hs}(j) = 5$ and $\texttt{xs}(j) = \texttt{bu}(j)$ as appropriate.

3. For Cold starts with no basis file, a CRASH procedure is used to select an initial basis. The initial basis matrix will be triangular (ignoring certain small entries in each column). The values $\texttt{hs}(j) = 0, 1, 2, 3, 4, 5$ have the following meaning:

$\texttt{hs}(j)$	State of variable j during crash
0, 1, 3	Eligible for the basis. 3 is given preference
2, 4, 5	Ignored

   After CRASH, columns for which $\texttt{hs}(j) = 2$ entries are made superbasic. Other entries not selected for the basis are made nonbasic at the value $\texttt{x}(j)$ if $\texttt{bl}(j) \leq \texttt{x}(j) \leq \texttt{bu}(j)$, or at the value $\texttt{bl}(j)$ or $\texttt{bu}(j)$ closest to $\texttt{x}(j)$.

4. For Warm starts, all of `hs(1:n+m)` is assumed to be set to the values 0, 1, 2 or 3 (probably from some previous call) and all of `xs(1:n+m)` must have values.

`pi` contains an estimate of the vector of Lagrange multipliers (shadow prices) for the nonlinear constraints. The first `nnCon` components must be defined. They will be used as λ in the Lagrangian function for the first major iteration. If nothing is known about λ, set $\texttt{pi}(i) =$ `0.0d+0`, $i = 1, 2, \ldots,$ `nnCon`.

`nS` need not be specified for Cold starts, but should retain its value from a previous call when a Warm start is used.

`cu(lencu),iu(leniu),ru(lenru)` are 8-character, integer and real arrays of user workspace. These arrays are available to pass 8-character, integer and real data to your versions of the routines `funcon` and `funobj`. These arrays are not touched by `snopt`.

If no workspace is required, you can use dummy arrays for `cu`, `iu` and `ru`, or use `cw`, `iw` and `rw` in their place (see below).

`cw(lencw),iw(leniw),rw(lenrw)` are 8-character, integer and real arrays of workspace used by `snopt`.

`lencw` must be at least 500. The bigger the values of `leniw` and `lenrw`, the better the performance since it is not certain how much storage the basis factors need. `leniw` should be about $10(m + n)$ or larger. `lenrw` should be about $20(m + n)$ or larger.

If `Print Level` is positive, the required amounts of workspace are printed. Thus, appropriate values may be obtained from a preliminary run with `lencw` = `leniw` = `lenrw` = 500 and `Print Level` > 0. (The values will be printed before `snopt` terminates with `inform` = 42, 43 or 44.)

On exit:

`hs` is the final state vector. The elements of `hs` have the following meaning:

`hs`(j)	State of variable j	Usual value of `x`(j)
0	nonbasic	`bl`(j)
1	nonbasic	`bu`(j)
2	superbasic	Between `bl`(j) and `bu`(j)
3	basic	ditto

Basic and superbasic variables may be outside their bounds by as much as the value of the optional parameter `Feasibility Tolerance`. Note that if scaling is specified, the `Feasibility Tolerance` applies to the variables of the *scaled* problem. In this case, the variables of the original problem may be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled. Check the "Primal infeasibility" printed after the `EXIT` message.

Very occasionally some nonbasic variables may be outside their bounds by as much as the `Feasibility Tolerance`, and there may be some nonbasics for which `xs`(j) lies strictly between its bounds.

If `nInf` > 0, some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by `sInf` if scaling was not used).

`xs(n+m)` is the final variables and slacks (x, s).

`pi(m)` is the vector of dual variables π (a set of Lagrange multipliers for the general constraints).

`rc(n+m)` is a vector of reduced costs, $g-(\,A\;\;-I\,)^T\pi$, where g is the gradient of the objective if `xs` is feasible (or the gradient of the phase-1 objective otherwise). If `nInf` $= 0$, the last m entries are π.

`inform` reports the result of the call to `snopt`. Here is a summary of possible values:

0 Optimal solution found. (The reduced gradient is negligible, the Lagrange multipliers are optimal, and `xs` satisfies the constraints to the accuracy requested.

1 The problem is infeasible.

2 The problem is unbounded (or badly scaled).

3 Too many iterations.

4 Feasible solution, but requested accuracy could not be achieved.

5 The `Superbasics limit` is too small.

6 User requested termination by returning `mode` < 0 from `funobj` or `funcon`.

7 Subroutine `funobj` seems to be giving incorrect gradients.

8 Subroutine `funcon` seems to be giving incorrect gradients.

9 The current point cannot be improved.

10 Numerical error in trying to satisfy the linear constraints (or the linearized nonlinear constraints). The basis is very ill-conditioned.

12 Basis factorization requested twice in a row. Should probably be treated as `inform` $=$ `9`.

20 Not enough storage for the basis factorization.

21 Error in basis package.

22 The basis is singular after several attempts to factorize it (and add slacks where necessary).

30 An OLD BASIS file had dimensions that did not match the current problem.

32 System error. Wrong number of basic variables.

42 Not enough 8-character workspace to solve the problem.

43 Not enough integer workspace to solve the problem.

44 Not enough real workspace to solve the problem.

`mincw,miniw,minrw` say how much character, integer and real storage is needed to solve the problem. If SNOPT terminates because of insufficent storage (`inform` = `42`, `43` or `44`), these values may be used to define better values of `lencw`, `leniw` or `lenrw`.

If `inform` = `42`, the work array `cw`(`lencw`) was too small. `snopt` may be called again with `lencw` suitably larger than `mincw`.

If `inform` = `43` or 44, the work arrays `iw`(`leniw`) or `rw`(`lenrw`) are too small. `snopt` may be called again with `leniw` or `lenrw` suitably larger than `miniw` or `minrw`. (The bigger the better, since it is not certain how much storage the basis factors need.)

`nS` is the final number of superbasics.

`nInf` is the number of infeasibilities.

`sInf` is the sum of infeasibilities.

`Obj` is the value of the objective function. If `nInf` = 0, `Obj` includes the quadratic objective if any. If `nInf` > 0, `Obj` is just the linear objective if any.

5. User-Supplied Subroutines

The user must provide subroutines that define the nonlinear parts of the objective function and nonlinear constraints. (A dummy subroutine must be provided if the objective or constraints have no nonlinear part.) Nonlinearities in the objective function are defined by subroutine `funobj`. Those in the constraints are defined separately by subroutine `funcon`. *On every entry* except perhaps the last, these subroutines must return appropriate function values `f`. Wherever possible, they should also return *all gradient components* in the array `g`. This provides maximum reliability and corresponds to the default setting, `Derivative Level = 3`.

In practice it is often convenient *not* to code gradients. SNOPT is able to estimate gradients by finite differences, by making a call to `funobj` or `funcon` for each variable x_j whose partial derivatives need to be estimated. *However*, this reduces the reliability of the optimization algorithms, and it can be very expensive if there are many such variables x_j.

As a compromise, SNOPT allows you to code *as many gradients as you like*. This option is implemented as follows: just before a function routine is called, each element of the gradient array `g` is initialized to a specific value. On exit, any element retaining that value must be estimated by finite differences.

Some rules of thumb follow:

1. For maximum simplicity and reliability, compute `f` and all components of `g`.
2. If not all gradients are known, compute as many of them as you can. (It often happens that some of them are constant or even zero.)
3. If *some* gradients are known (but not all), it may be convenient to compute them each time the function routines are called, even though they will be ignored if `mode` $= 2$.
4. If the known gradients are *expensive* to compute, use the parameter `mode` to avoid computing them on certain entries.
5. While the function routines are being developed, use the `verify` parameter to check the computation of any gradient elements that are supposedly known.

5.1. Subroutine funobj

This subroutine must calculate the objective function $f(x)$ and (optionally) the gradient $g(x)$. The specification of `funobj` is

```
 subroutine funobj( mode, nnObj,
$                   x, fObj, gObj, nState,
$                   cu, lencu, iu, leniu, ru, lenru )
 integer            mode, nnObj, nState
 real               fObj
 real               x(nnObj), gObj(nnObj)

 integer            lencu, leniu, lenru
 character*8        cu(lencu)
 integer            iu(leniu)
 real               ru(lenru)
```

On entry:

`mode` is set by `snopt` to indicate which values are to be assigned during the call of `funobj` ($0 \leq$ `mode` ≤ 2).

This parameter can be ignored if the (default) derivative linesearch is selected, and if `Derivative level` is 1 or 3 (i.e., if all elements of `gObj` are are specified by the user). In this case, `mode` will always have the value 2.

If some derivatives are unspecified, `snopt` will call `funobj` with `mode` = 0, 1 or 2. You may test `mode` to decide what to do:

- If `mode` = 2, assign `fObj` and the known components of `gObj`.
- If `mode` = 1, assign all available components of `gObj`; `fObj` is not required and is ignored.
- If `mode` = 0, only `fObj` needs to be assigned; `gObj` is ignored.

`nnObj` is the number of variables involved in f, i.e., the number of nonlinear objective variables (> 0). These must be the first `nnObj` variables in the problem.

`x` is an array of dimension at least `nnObj` containing the values of the variables x for which f must be evaluated.

`nState` allows you to save computation time if certain data must be read or calculated only once.

If `nState` = 0, there is nothing special about the current call to `funobj`.

If `nState` = 1, SNOPT is calling your subroutine for the first time. Some data may need to be input or computed and saved in local or `common` storage. Note that if there are nonlinear constraints, the first call to `funcon` will occur *before* the first call to `funobj`.

If `nState` ≥ 2, SNOPT is calling your subroutine for the *last* time. You may wish to perform some additional computation on the final solution. Note again that if there are nonlinear constraints, the last call to `funcon` will occur *before* the last call to `funobj`.

In general, the last call is made with `nState` = 2 + `ierr`, where `ierr` indicates the status of the final solution. In particular, if `nState` = 2, the current `x` is *optimal*;

if `nState` = 3, the problem appears to be infeasible; if `nState` = 4, the problem appears to be unbounded; and if `nState` = 5, the iterations limit was reached. In some cases, the solution may be *nearly* optimal if `nState` = 11; this value occurs if the line search procedure was unable to find an improved point.

If the nonlinear functions are expensive to evaluate, it may be desirable to do nothing on the last call, by including a statement of the form

```
if (nState .ge. 2) return
```

at the start of the subroutine.

`cu(lencu), iu(leniu), ru(lenru)` are character, integer and real arrays that can be used to pass user-defined auxiliary information into `funobj`. The arrays `cu`,`iu` and `ru` are not touched by `snopt` and can be used to retain information between calls of `funobj`.

In certain applications, the objective may depend on the values of certain internal SNOPT variables stored in the arrays `cw`, `iw` and `rw`. In this case, `cw`, `iw` and `rw` can be made accessible to `funobj` by calling `sqopt` with `cw`, `iw` and `rw` used for the arguments `cu`, `iu` and `ru`. Once this is done, various values of `iw` can be used to pinpoint the required locations of `snopt` arrays. For example, the dual variables are stored in `rw(lpi)` onward, where the address `lpi` is the value of `iw(257)`. If you still require user work space, elements `501:maxcu`, `501:maxru` and `501:maxiu` of `cw`, `rw` and `iw` are set aside for this purpose. (See the definition of the optional parameters `User real workspace` and `User integer workspace` in §7.3.

If you do not require workspace to be passed into `funobj`, the `snopt` work arrays `cw`, `iw` and `rw` can be used for `cu`, `iu` and `ru`.

On exit:

`mode` can be used to indicate that you are unable or unwilling to evaluate the objective function at the current x.

During the line search, f is evaluated at points x of the form $x = x_k + \alpha p_k$, where f already has been evaluated satisfactorily at x_k. At any such α, it you set `mode` to -1 SNOPT will evaluate f at some point closer to x_k (a point at which f is more likely to be defined.

If for some reason you wish to terminate solution of the current problem, set `mode` to a negative value (other than -1).

`fObj` must contain the computed value of $f(x)$.

`gObj(nnObj)` must contain the assigned components of the gradient vector $g(x)$, i.e., `gObj`(j) contains the partial derivative $\partial f/\partial x_j$ (except perhaps if `mode` = 0—see above).

5.2. Subroutine funcon

This subroutine must compute the nonlinear constraint functions $F(x)$ and (optionally) their gradients. (A dummy subroutine `funcon` must be provided if all constraints are linear.) The gradients of the vector $F(x)$ define the Jacobian matrix $A_1(x)$. The jth column of $A_1(x)$ is the vector $\partial F/\partial x_j$.

Subroutine `funcon` may be coded in two different ways, depending on the method used for storing the Jacobian, as specified in the SPECS file. For the option

```
Jacobian = dense
```

the specification of `funcon` is

```
 subroutine funcon( mode, nnCon, nnJac, neJac,
$                   x, fCon, gCon, nState,
$                   cu, lencu, iu, leniu, ru, lenru )
 integer            mode, nnCon, nnJac, neJac, nState
 real               x(nnJac), fCon(nnCon), gCon(nnCon,nnJac)

 integer            lencu, leniu, lenru
 character*8        cu(lencu)
 integer            iu(leniu)
 real               ru(lenru)
```

On entry:

`mode` This parameter can be ignored if `Derivative linesearch` is selected (the default), and if `Derivative level = 2` or 3 (i.e., if all elements of `gCon` are computed). In this case, `mode` will always have the value 2.

Otherwise, you must specify `Derivative level = 0` or `1` in the SPECS file to indicate that `funcon` will not compute all of `gCon`. You may then test `mode` in `funcon` to decide what to do:

- If `mode` = 2, compute `fCon` and as many components of `gCon` as possible.
- If `mode` = 0, compute `fCon` but set `gCon` only if you wish. (On return, the contents of `gCon` will be ignored.)

`nnCon` is the number of nonlinear constraints (`nnCon` > 0). These must be the first `nnCon` constraints in the problem.

`nnJac` is the number of variables involved in $F(x)$ ($0 <$ `nnJac` $\leq$ `n`). These must be the first `nnJac` variables in the problem.

`neJac` is the value `nnCon*nnJac`.

`x(nnJac)` contains the current values of the nonlinear variables x.

On exit:

`fCon(nnCon)` contains the computed values of the functions in the constraint vector $F(x)$.

`gCon(nnCon,nnJac)` contains the computed Jacobian $J(x)$. The jth column of $J(x)$ should be stored in the jth column of the 2-dimensional array `gCon` (except perhaps if `mode` = 0—see above). Equivalently, the gradient of the ith constraint should be stored in the ith row of `gCon`.

The other parameters are the same as for subroutine `funobj`.

If the option

Jacobian = Sparse

is selected, the specification of `funcon` is as follows.

```
 subroutine funcon( mode, nnCon, nnJac, neJac,
$                   x, fCon, gCon, nState,
$                   cu, lencu, iu, leniu, ru, lenru )
 integer            mode, nnCon, nnJac, neJac, nState
 real               x(nnJac), fCon(nnCon), gCon(neJac)

 integer            lencu, leniu, lenru
 character*8        cu(lencu)
 integer            iu(leniu)
 real               ru(lenru)
```

This is the same as for `Jacobian = Dense`, except for the declaration of `gCon(neJac)`.

On entry:

`neJac` is the number of nonzero elements in the constraint Jacobian $J(x)$ (> 0).

Usually `neJac` will be less than `nnCon*nnJac`. The actual value of `neJac` may not be of any use when coding `funcon`, but in all cases, any expression involving `gCon(`l`)` should have the subscript l between `1` and `neJac`.

On exit:

`gCon(neJac)` contains the computed elements of the Jacobian (except perhaps if `mode` $= 0$—see previous page). These elements must be stored into `gCon` in exactly the same positions as implied by the definitions of the arrays `a`, `ha` and `ka`. There is no internal check for consistency (except indirectly via the `Verify` parameter), so great care is essential.

The other parameters are the same as for `Jacobian = dense`.

5.3. Constant Jacobian elements

If all constraint gradients (Jacobian elements) are known (`Derivative level = 2` or `3`), any *constant* elements may be assigned to the array `a` if desired. An element of `gCon` that is not computed in `funcon` will retain the value implied by its value in `a`. (The value is taken to be *zero* if not given explicitly in `a`.)

This feature is useful when `Jacobian = Dense` and many Jacobian elements are identically zero. Such elements need not be set in `funcon`.

Note that constant *nonzero* elements do affect `fCon`. Thus, if J_{ij} is assigned in `a` and is constant, the array element `gCon(`i`,`j`)` need not be set in `funcon`, but the value `gCon(`i`,`j`)*x(`j`)` must be added to `fCon(`i`)`.

When `Jacobian = Sparse`, constant Jacobian elements will normally not be listed in `a` unless they are nonzero. If the correct value *is* entered in `a`, the corresponding element `gCon(`l`)` need not be reassigned, but an appropriate linear term of the form `gCon(`l`)*x(`j`)` must be added to one of the elements of `fCon`.

Remember, if `Derivative level < 2`, unassigned elements of `gCon` are *not* treated as constant; they are estimated by finite differences, at significant expense.

5.4. Example

Here we give the subroutines `funobj` and `funcon` for the four dimensional example of Section 3, repeated here for convenience.

$$\begin{array}{lrcl} \text{minimize} & (x_1+x_2+x_3)^2+3x_3+5x_4 & & \\ \text{subject to} & x_1^2+x_2^2+\ x_3 & = & 2 \\ & x_1^4+x_2^4 \quad +\ x_4 & = & 4 \\ & 2x_1+4x_2 & \geq & 0 \\ & x_3\ \geq 0 \quad x_4 & \geq & 0. \end{array}$$

This problem has 4 variables, 3 nonlinear objective variables, 2 nonlinear Jacobian variables, 2 nonlinear constraints and 1 linear constraint. This implies that the calling program must include the statements

```
n      =  4
m      =  4
nnCon  =  2
nnJac  =  2
nnObj  =  3
```

The subroutine `funobj` involves only the nonlinear objective variables, which in this case are the variables x_1, x_2, and x_3.

```
      subroutine funobj( mode, nnObj,
     $                   x, fObj, gObj, nState,
     $                   cu, lencu, iu, leniu, ru, lenru )

      real               x(nnObj), gObj(nnObj)

      integer            lencu, leniu, lenru
      character*8        cu(lencu)
      integer            iu(leniu)
      real               ru(lenru)

*     ================================================================
*     Toy NLP problem from the SNOPT User's Guide
*     ================================================================
      sum     = x(1) + x(2) + x(3)
      fObj    = sum*sum

      sum     = 2.0d+0*sum
      gObj(1) = sum
      gObj(2) = sum
      gObj(3) = sum

*     end of funobj for toy NLP.
      end
```

The subroutine `funobj` involves the variable x_3 that occurs only linearly in the constraints. In such cases we recommend that the objective variables be placed *after* the Jacobian variables in the in the arrays `a`, `ha`, `ka`.

```
      subroutine funcon( mode, nnCon, nnJac, neJac,
     $                   x, fCon, gCon, nState,
     $                   cu, lencu, iu, leniu, ru, lenru )
```

```
      real               x(nnJac), fCon(nnCon), gCon(nnCon,nnJac)

      integer            lencu, leniu, lenru
      character*8        cu(lencu)
      integer            iu(leniu)
      real               ru(lenru)

*     ==================================================================
*     Toy NLP problem from the SNOPT User's Guide
*     Jacobian = DENSE
*     ==================================================================
      fCon( 1) = x(1)**2 +  x(2)**2
      fCon( 2) = x(1)**4 +  x(2)**4

*     Nonlinear Jacobian elements for column 1.
*     rows = (1,2).

      gCon(1,1) = 2.0d+0*x(1)
      gCon(2,1) = 4.0d+0*x(1)**3

*     Nonlinear Jacobian elements for column 2.
*     Rows = (1,2).

      gCon(1,2) = 2.0d+0*x(2)
      gCon(2,2) = 4.0d+0*x(2)**3

*     end of funcon for toy NLP.
      end
```

6. The SPECS File

The performance of SNOPT is controlled by a number of parameters or "options". Each option has a default value that should be appropriate for most problems. (The defaults are given in the next section.) For special situations it is possible to specify non-standard values for some or all of the options, using data in the following general form:

```
Begin SNOPT options
   Iterations limit             500
   Minor feasibility tolerance  1.0e-7
   Scale all variables
End SNOPT options
```

We shall call such data a SPECS file, since it specifies various options. The file starts with the keyword `Begin` and ends with `End`. The file is in free format. Each line specifies a single option, using one or more items as follows:

1. A *keyword* (required for all options).
2. A *phrase* (one or more words) that qualifies the keyword (only for some options).
3. A *number* that specifies an integer or real value (only for some options). Such numbers may be up to 16 contiguous characters in Fortran 77's `I`, `F`, `E` or `D` formats, terminated by a space.

The items may be entered in upper or lower case or a mixture of both. Some of the keywords have synonyms, and certain abbreviations are allowed, as long as there is no ambiguity. Blank lines and comments may be used to improve readability. A comment begins with an asterisk (`*`), which may appear anywhere on a line. All subsequent characters on the line are ignored.

Although most options take default values, some of them must be specified; for example, the number of nonlinear variables if there are any.

It may be useful to include a comment on the first (`Begin`) line of the file. This line is echoed to the SUMMARY file, and appears on the screen in an interactive environment.

Most of the options described in the next section should be left at their default values for any given model. If experimentation is necessary, we recommend changing just one option at a time.

7. SPECS File Checklist and Defaults

The following example SPECS file shows all valid *keywords* and their *default values*. The keywords are grouped according to the function they perform.

Some of the default values depend on ϵ, the relative precision of the machine being used. The values given here correspond to double-precision arithmetic on most current machines ($\epsilon \approx 2.22 \times 10^{-16}$). Similar values would apply to any machine having about 15 decimal digits of precision.

```
BEGIN checklist of SPECS file parameters and their default values
* Printing
  Major print level                1        * summary line each major iteration
  Minor print level                0        * summary line each minor iteration
  Summary frequency                1        * iteration log on SUMMARY file
  Print  frequency                 1        * iteration log on PRINT file
  Solution                         Yes      * on the PRINT file
 *Suppress options listing                  * Options listed by default
```

* Problem dimensions		
Nonlinear objective variables	0	* use if different from Jacobian variables
Nonlinear constraints	0	* must be the exact number, m_1
Nonlinear variables	0	* must be the exact number, n_1
Nonlinear Jacobian variables	0	* use if different from objective variables
* Convergence Tolerances		
Minor feasibility tolerance	1.0e-6	* for satisfying the QP bounds
Minor optimality tolerance	1.0e-6	* target value for reduced gradients
Major feasibility tolerance	1.0e-6	* target nonlinear constraint violation
Major optimality tolerance	1.0e-6	* target complementarity gap
* Derivative checking		
Verify level	0	* gives cheap check on gradients
Start objective check at col	1	*
Stop objective check at col	n_1	*
Start constraint check at col	1	*
Stop constraint check at col	n_1	*
* Scaling		
Scale option	1	* linear constraints and variables
Scale tolerance	0.9	*
* Other Tolerances		
Crash tolerance	0.1	*
Linesearch tolerance	0.9	* smaller for more accurate search
LU factor tolerance	10.0	* limits size of multipliers in L
LU update tolerance	10.0	* the same during updates
LU singularity tolerance	2.0e-6	*
Pivot tolerance	3.7e-11	* $\epsilon^{\frac{2}{3}}$
* QP subproblems		
Crash option	0	* All slack initial basis
Elastic weight	100	* used only during elastic mode
Iterations limit	300	* or 3*Rows + 10*Nonlinear variables
Partial price	1	* or Cols/(2*Rows) if Cols is large
* SQP method		
Minimize		* (opposite of Maximize)
Feasible point	not set	* (alternative to Max or min)
Feasible Exit	not set	* (get feasible before exiting)
Jacobian	Dense	*
Major iterations limit	40	*
Minor iterations limit	40	*
Major step limit	2.0	*
Superbasics limit	50	*
Reduced Hessian dimension	50	* or Superbasics limit
Derivative linesearch	Yes	*
Function precision	3.0e-13	* $\epsilon^{0.8}$ (almost full accuracy)
Difference interval	5.5e-7	* (Function precision)$^{\frac{1}{2}}$
Central difference interval	6.7e-5	* (Function precision)$^{\frac{1}{3}}$
Derivative level	3	* assumes all gradients are known
Violation limit	10.0	* unscaled constraint violation limit
Unbounded step size	1.0e+18	*
Unbounded objective	1.0e+15	*
* Hessian approximation		

```
  Hessian limited memory                    * default if n > 75
  Hessian full    memory                    * default if n ≤ 75
  Hessian updates                  20       * ignored if Hessian is full
  Hessian flush                    999999   * no flushing

* Frequencies
  Factorization frequency          50       *
  Expand frequency                 10000    *
  Hessian frequency                999999   * never reset the Hessian
  Check frequency                  60       * numerical test on row residuals
  Factorization frequency          100      * refactorize the basis matrix
  Save frequency                   100      * basis map
  Save new basis map               100      *

* BASIS files
  OLD BASIS file                   0        * input basis map
  NEW BASIS file                   0        * output basis map
  BACKUP BASIS file                0        * output basis map
  INSERT file                      0        * input in industry format
  PUNCH file                       0        * output INSERT data
  LOAD file                        0        * input names and values
  DUMP file                        0        * output LOAD data
  SOLUTION file                    0        * separate from printed solution

* Miscellaneous
  Total character workspace        ?        * depends on the installation
  Total integer   workspace        ?        * depends on the installation
  Total real      workspace        ?        * depends on the installation
End of SPECS file checklist
```

7.1. Subroutine snInit (to read a Specs file)

Subroutine `snInit` is provided with SNOPT to initialize the default parameters and read options from an external Specs file (if any). `snInit` *MUST be called before the first call of SNOPT*.

Specification:

```
 subroutine snInit( iSpecs, iPrint, iSumm, inform,
$                   cw, lencw, iw, leniw, rw, lenrw )
 integer            iSpecs, iPrint, iSumm, inform
 integer            lencw, leniw, lenrw
 character*8        cw(lencw)
 integer            iw(leniw)
 real               rw(lenrw)
```

On entry:

`iSpecs` says whether or not a SPECS file exists. If `iSpecs` > 0, a file is read from the specified Fortran file number. Typically `iSpecs = 4`.

`iPrint` says if a PRINT file is to be created. Typically `iPrint = 9`.

`iSumm` says if a SUMMARY file is to be created. Typically `iSumm` $= 6$. In an interactive environment, this value usually denotes the screen.

The remaining parameters have the same function as those of SNOPT.

On exit:

All input parameters are unchanged except `inform`, which is defined as follows.

`inform` is 0 if there was no SPECS file, or if the SPECS file was successfully read. Otherwise, it returns the number of errors encountered.

7.2. Subroutines snprm, snprmi, snprmr (to define a single option)

These subroutines may be called from the program that calls `snopt`. They specify a single option that might otherwise be defined in one line of a SPECS file.

Specification:

```
 subroutine snprm ( buffer,         iPrint, iSumm, inform,
$                   cw, lencw, iw, leniw, rw, lenrw )
 subroutine snprmi( buffer, ivalue, iPrint, iSumm, inform,
$                   cw, lencw, iw, leniw, rw, lenrw )
 subroutine snprmr( buffer, rvalue, iPrint, iSumm, inform,
$                   cw, lencw, iw, leniw, rw, lenrw )
 character*(*)      buffer
 integer            ivalue
 real               rvalue
 integer            iPrint, iSumm, inform
 integer            lencw, leniw, lenrw
 character*8        cw(lencw)
 integer            iw(leniw)
 real               rw(lenrw)
```

On entry:

`buffer` is a string to be decoded as if it were a line of a SPECS file. For `snprm`, the maximum length of `buffer` is 72 characters. Use `snprm` if the string contains all of the data associated with a particular keyword. For example,

```
call snprm ( 'Iterations 1000',    iPrint, iSumm, inform, ...
```

is suitable if the value 1000 is known at compile time.

For `snprmi` and `snprmr` the maximum length of `buffer` is 55 characters.

`ivalue` is an integer value associated with the keyword in `buffer`. Use `snprmi` if it is convenient to define the value at run time. For example,

```
itnlim = 1000
if (m .gt. 500) itnlim = 8000
call snprmi( 'Iterations', itnlim, iPrint, iSumm, inform, ...
```

allows the iteration limit to be computed.

`rvalue` is a floating-point value associated with the keyword in `buffer`. Use `snprmr` if it is convenient to define the value at run time. For example,

```
factol = 100.0d+0
if ( illcon ) factol = 5.0d+0
call snprmr( 'LU factor tol', factol, iPrint, iSumm, inform, ...
```

allows the *LU* stability tolerance to be computed.

`iPrint` is a file number for printing each line of data, along with any error messages. `iPrint` = 0 suppresses this output.

`iSumm` is a file number for printing any error messages. `iSumm` = 0 suppresses this output.

`inform` should be 0.

On exit:

`inform` is the number of errors encountered so far.

7.3. Description of the optional parameters

The following is an alphabetical list of the options that may appear in the SPECS file, and a description of their effect.

`Backup Basis file` i Default = 0

This is intended as a safeguard against losing the results of a long run. Suppose that a NEW BASIS file is being saved every 100 iterations, and that SNOPT is about to save such a basis at iteration 2000. It is conceivable that the run may time-out during the next few milliseconds (i.e., in the middle of the save), or the host computer could unexpectedly crash. In this case the basis file will be corrupted and the run will have been essentially wasted.

To eliminate this risk, both a NEW BASIS file and a BACKUP BASIS file may be specified. The following would be suitable for the above example:

```
OLD BASIS file      11     (or 0)
BACKUP BASIS file   11
NEW BASIS file      12
Save frequency     100
```

The current basis will then be saved every 100 iterations, first on file 12 and then immediately on file 11. If the run is interrupted at iteration 2000 during the save on file 12, there will still be a useable basis on file 11 (corresponding to iteration 1900).

Note that a NEW BASIS will be saved at the end of a run if it terminates normally, but there is no need for a further BACKUP BASIS. In the above example, if an optimum solution is found at iteration 2050 (or if the iteration limit is 2050), the final basis on file 12 will correspond to iteration 2050, but the last basis saved on file 11 will be the one for iteration 2000.

`Central difference interval` r Default = $\epsilon^{\frac{1}{3}}$

When `Derivative level` < 3, the central-difference interval r is used near an optimal solution to obtain more accurate (but more expensive) estimates of gradients. Twice as many function evaluations are required compared to forward differencing. The interval used for the jth variable is $h_j = r(1 + |x_j|)$. The resulting gradient estimates should be accurate to $O(r^2)$, unless the functions are badly scaled.

`Check frequency` i Default = 60

Every ith minor iteration after the most recent basis factorization, a numerical test is made to see if the current solution x satisfies the general linear constraints (including linearized nonlinear constraints, if any). The constraints are of the form $Ax - s = b$, where s is the set of slack variables. To perform the numerical test, the residual vector $r = b - Ax + s$ is computed. If the largest component of r is judged to be too large, the current basis is refactorized and the basic variables are recomputed to satisfy the general constraints more accurately.

`Check frequency 1` is useful for debugging purposes, but otherwise this option should not be needed.

`Crash option` i Default = 0
`Crash tolerance` r Default = 0.1

Except on restarts, a CRASH procedure is used to select an initial basis from certain rows and columns of the constraint matrix $(A \quad -I)$. The `Crash option` i determines which

rows and columns of A are eligible initially, and how many times CRASH is called. Columns of $-I$ are used to pad the basis where necessary.

i	*Meaning*
0	The initial basis contains only slack variables: $B = I$.
1	CRASH is called once, looking for a triangular basis in all rows and columns of the matrix A.
2	CRASH is called twice (if there are nonlinear constraints). The first call looks for a triangular basis in linear rows, and the iteration proceeds with simplex iterations until the linear constraints are satisfied. The Jacobian is then evaluated for the first major iteration and CRASH is called again to find a triangular basis in the nonlinear rows (retaining the current basis for linear rows).
3	CRASH is called up to three times (if there are nonlinear constraints). The first two calls treat *linear equalities* and *linear inequalities* separately. As before, the last call treats nonlinear rows before the first major iteration.

If $i \geq 1$, certain slacks on inequality rows are selected for the basis first. (If $i \geq 2$, numerical values are used to exclude slacks that are close to a bound.) CRASH then makes several passes through the columns of A, searching for a basis matrix that is essentially triangular. A column is assigned to "pivot" on a particular row if the column contains a suitably large element in a row that has not yet been assigned. (The pivot elements ultimately form the diagonals of the triangular basis.) For remaining unassigned rows, slack variables are inserted to complete the basis.

The `Crash tolerance` r allows the starting procedure CRASH to ignore certain "small" nonzeros in each column of A. If $a_{\max}$ is the largest element in column j, other nonzeros a_{ij} in the column are ignored if $|a_{ij}| \leq a_{\max} \times r$. (To be meaningful, r should be in the range $0 \leq r < 1$.)

When $r > 0.0$, the basis obtained by CRASH may not be strictly triangular, but it is likely to be nonsingular and almost triangular. The intention is to obtain a starting basis containing more columns of A and fewer (arbitrary) slacks. A feasible solution may be reached sooner on some problems.

For example, suppose the first m columns of A are the matrix shown under `LU factor tolerance`; i.e., a tridiagonal matrix with entries -1, 4, -1. To help CRASH choose all m columns for the initial basis, we would specify `Crash tolerance` r for some value of $r > 1/4$.

`Derivative Level` i Default = 3

This specifies which nonlinear function gradients are known analytically and will be supplied to SNOPT by the user subroutines `funobj` and `funcon`.

i	*Meaning*
3	All objective and constraint gradients are known.
2	All constraint gradients are known, but some or all components of the objective gradient are unknown.
1	The objective gradient is known, but some or all of the constraint gradients are unknown.

0 Some components of the objective gradient are unknown and some of the constraint gradients are unknown.

The value $i =$ `3` should be used whenever possible. It is the most reliable and will usually be the most efficient.

If $i =$ `0` or `2`, SNOPT will *estimate* the missing components of the objective gradient, using finite differences. This may simplify the coding of subroutine `funobj`. However, it could increase the total run-time substantially (since a special call to `funobj` is required for each missing element), and there is less assurance that an acceptable solution will be located. If the nonlinear variables are not well scaled, it may be necessary to specify a nonstandard `Difference interval` (see below).

If $i =$ `0` or `1`, SNOPT will estimate missing elements of the Jacobian. For each column of the Jacobian, one call to `funcon` is needed to estimate all missing elements in that column, if any. If `Jacobian = sparse` and the sparsity pattern of the Jacobian happens to be

$$\begin{pmatrix} * & * & & * \\ & ? & ? & \\ * & & ? & \\ & * & & * \end{pmatrix}$$

where `*` indicates known gradients and `?` indicates unknown elements, SNOPT will use one call to `funcon` to estimate the missing element in column 2, and another call to estimate both missing elements in column 3. No calls are needed for columns 1 and 4.

At times, central differences are used rather than forward differences. Twice as many calls to `funobj` and `funcon` are then needed. (This is not under the user's control.)

Remember: when analytic derivatives are not provided, the attainable accuracy in locating an optimal solution is usually less than when all gradients are available. `Derivative level` = 3 is strongly recommended.

`Derivative linesearch` Default
`Nonderivative linesearch`
`No derivative linesearch`

At each major iteration a linesearch is used to improve the merit function. A `Derivative linesearch` uses safeguarded cubic interpolation and requires both function and gradient to compute estimates of the step α_k. If some analytic derivatives are not provided, or a `Nonderivative linesearch` is specified, SNOPT employs a linesearch based upon a safeguarded quadratic interpolation, which does not require the evaluation of the gradients or their approximations.

A nonderivative linesearch can be slightly less robust on difficult problems, and it is recommended that the default be used if the functions and derivatives can be computed at approximately the same cost. If the gradients are very expensive relative to the functions, a nonderivative linesearch may give a significant decrease in computation time.

If the `Nonderivative linesearch` is selected, SNOPT signals the evaluation of the linesearch by calling the user-defined routines `funobj` and `funcon` with `mode` = 0. Once the linesearch is completed, the problem functions are called again with `mode` = 2. If the potential savings provided by a nonderivative linesearch are to be realized, it is essential that `funobj` and `funcon` be coded so that the derivatives are not computed when `mode` = 0.

`Difference Interval` h_1 Default = $\epsilon^{1/2}$

This alters the interval h_1 that is used to estimate gradients by forward differences in the following circumstances:

- In the initial ("cheap") phase of verifying the objective gradients.
- For verifying the constraint gradients.
- For estimating missing objective gradients.
- For estimating missing Jacobian elements.

In the last three cases, a derivative with respect to x_j is estimated by perturbing that component of x to the value $x_j + h_1(1 + |x_j|)$, and then evaluating $F(x)$ or $f(x)$ at the perturbed point. The resulting gradient estimates should be accurate to $O(h_1)$ unless the functions are badly scaled. Judicious alteration of h_1 may sometimes lead to greater accuracy.

`Dump File` i Default = 0

If $i > 0$, the last solution obtained will be output to the file with unit number i in the format described in Section 9.3. The file will usually have been output previously as a LOAD file.

`Elastic weight` r Default = 0.0

This keyword invokes the so-called *composite objective* technique, if the first solution obtained is infeasible, and if linear terms for the objective function are specified in the MPS file. While trying to reduce the sum of infeasibilities, the method also attempts to optimize the linear objective.

- At each infeasible iteration, the objective function is defined to be

$$\text{minimize} \quad \sigma c^T x + r(\text{sum of infeasibilities}),$$

  where $\sigma = 1$ for `Minimize`, $\sigma = -1$ for `Maximize`, and c is the linear objective row.

- If an "optimal" solution is reached while still infeasible, r is increased by a factor of 10. This helps to allow for the possibility that the initial r is too large. It also provides dynamic allowance for the fact the sum of infeasibilities is tending towards zero.
- The effect of r is disabled if a feasible solution is obtained.

`Expand frequency` i Default = 10000

This option is part of an anti-cycling procedure designed to make progress even on highly degenerate problems. *

For linear models, the strategy is to force a positive step at every iteration, at the expense of violating the bounds on the variables by a small amount. Suppose that the `Minor feasibility tolerance` is δ. Over a period of i iterations, the tolerance actually used by SNOPT increases from 0.5δ to δ (in steps of $0.5\delta/i$).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can occur only when the current solution is at a vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

*The EXPAND procedure is described in "A practical anti-cycling procedure for linearly constrained optimization", P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright, *Mathematical Programming* 45 (1989), pp. 437–474.

Increasing i helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see `Pivot tolerance`).

`Factorization Frequency` k Default = 50

At most k basis changes will occur between factorizations of the basis matrix.

- With linear programs, the basis factors are usually updated every iteration. The default k is reasonable for typical problems. Higher values up to $k =$ `100` (say) may be more efficient on problems that are extremely sparse and well scaled.
- When the objective function is nonlinear, fewer basis updates will occur as an optimum is approached. The number of iterations between basis factorizations will therefore increase. During these iterations a test is made regularly (according to the `Check frequency`) to ensure that the general constraints are satisfied. If necessary the basis will be refactorized before the limit of k updates is reached.
- When the constraints are nonlinear, the `Minor iterations` limit will probably preempt k.

`Feasibility tolerance` t Default = `1.0E-6`
see `Minor feasibility tolerance`

`Feasible point`

Find a feasible point for the nonlinear constraints (the objective is ignored). This option can be used to check that the nonlinear constraints are feasible.

`Feasible exit`
`Infeasible exit ok` Default

This option requests that SNOPT terminates with a point that is feasible for the nonlinear constraints. If SNOPT terminates abnormally, additional iterations are made to get feasible with respect to the nonlinear constraints. This option is ignored if the maximum number of iterations is exceeded, or the linear constraints are infeasible.

`Function Precision` ϵ_R Default = $\epsilon^{0.8}$

The *relative function precision* ϵ_R is intended to be a measure of the relative accuracy with which the nonlinear functions can be computed. For example, if $f(x)$ is computed as `1000.56789` for some relevant x and if the first 6 significant digits are known to be correct, the appropriate value for ϵ_R would be `1.0E-6`.

(Ideally the functions $f(x)$ or $F^i(x)$ should have magnitude of order 1. If all functions are substantially *less* than 1 in magnitude, ϵ_R should be the *absolute* precision. For example, if $f(x) =$ `1.23456789E-4` at some point and if the first 6 significant digits are known to be correct, the appropriate value for ϵ_R would be `1.0E-10`.)

- The default value of ϵ_R is appropriate for simple analytic functions.

- In some cases the function values will be the result of extensive computation, possibly involving an iterative procedure that can provide rather few digits of precision at reasonable cost. Specifying an appropriate `Function precision` may lead to savings, by allowing the linesearch procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

`Hessian limited memory`
`Hessian full memory` Default if $\max\{n_1', n_1''\} < 75$

These options specify the form of the quasi-Newton approximation to the Hessian of the Lagrangian.

If `Hessian Full memory` is specified, the approximate Hessian is treated as a dense matrix and apply the BFGS quasi-Newton updates are applied explicitly. This option is most efficient when the number of nonlinear variables $\bar{n}$ is not too large (say, less than 75). In this case, the storage requirement is fixed, and the user can expect 1-step Q-superlinear convergence to the solution.

`Hessian limited memory` should be used on problems where the number of nonlinear variables $\bar{n}$ is very large. In this case a limited-memory procedure is used to update an initial Hessian approximation H_r a limited number of times.

`Hessian updates` i Default = 20

If `Hessian limited memory` is selected, this option defines the maximum number of pairs of Hessian update vectors that are used to define the quasi-Newton approximate Hessian. Once the `Hessian update` limit is attained, the accumulated updates are discarded and the process starts again.

Broadly speaking, the more updates stored, the better the quality of the approximate Hessian. However, the more vectors stored, the greater the cost of each QP iteration. The default value is likely to give a robust algorithm without significant expense, but faster converged can be obtained with far fewer updates (e.g., $i = 5$).

`Hessian frequency` i Default = 20

This option forces the approximate Hessian to be reset to a diagonal every i successful updates. if `Hessian frequency = 20` is used with `Hessian full memory`, the have a similar effect to `Hessian limited memory` with `Hessian updates = 20`.

`Insert File` f Default = 0

If $f > 0$, this references a file containing basis information in the format of Section 9.2.

- The file will usually have been output previously as a PUNCH file.
- The file will not be accessed if an OLD BASIS file is specified.

`Invert Frequency` k Default = 50
see `Factorization Frequency`

`Iterations Limit` k Default = $3 * $`rows` $+ 10 * $`Nonlinear Vars`

This is the maximum number of minor iterations allowed (i.e., iterations of the simplex method or the QP algorithm).

- `Itns` is an alternative keyword.
- $k = 0$ is valid. Both feasibility and optimality are checked.

`Infinite Bound Size` r Default = `1.0E+20`

If $r > 0$, r defines the "infinite" bound `BigBnd` in the definition of the problem constraints. Any upper bound greater than or equal to `BigBnd` will be regarded as plus infinity (and similarly for a lower bound less than or equal to $-$`BigBnd`). If $r \leq 0$, the default value is used.

`Jacobian`	`Dense`	Default
`Jacobian`	`Sparse`	

This determines the manner in which the constraint gradients are evaluated and stored. It affects the `snopt` array arguments `a`, `ha` and `ka`, and the subroutine `funcon`.

- The `Dense` option is convenient if there are not many nonlinear constraints or variables. It requires storage for three dense matrices of order $m_1 \times n_1$.
- For efficiency, the `Sparse` option is preferable in all nontrivial cases. (*Beware – it must be specifically requested.*) The `snopt` arrays `a`, `ha` and `ka` must then include all Jacobian elements (that are not identically zero), and subroutine `funcon` must store the elements of the Jacobian array `gCon` in exactly the same order.
- In both cases, if `Derivative Level = 2` or `3` the arrays `a`, `ha` and `ka` may specify Jacobian elements that are constant for all values of the nonlinear variables. The corresponding elements of `gCon` need not be reset in `funcon`.

`Linesearch tolerance` t Default = 0.9

This controls the accuracy with which a steplength will be located along the direction of search each iteration. At the start of each linesearch a target directional derivative for the merit function is identified. This parameter determines the accuracy to which this target value is approximated.

- t must be a real value in the range `0.0` $\leq t \leq$ `1.0`.
- The default value $t =$ `0.1` requests a moderately accurate search. It should be satisfactory for many problems.
- If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try $t =$ `0.01` or $t =$ `0.001`. The number of iterations should decrease, and this will reduce total run time if there are many linear or nonlinear constraints.
- If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. *If all gradients are known*, try $t =$ `0.5` or perhaps $t =$ `0.9`. (The number of iterations will probably increase, but the total number of function evaluations may decrease enough to compensate.)
- If not all gradients are known, a reasonably accurate search remains appropriate. Each search will require only 2–5 function values (typically), but many function calls will then be needed to estimate missing gradients for the next iteration.

`Load File` f Default = 0

If $f > 0$, this references a file containing basis information in the format of Section 9.3.

- The file will usually have been output previously as a DUMP file.
- The file will not be accessed if an OLD BASIS file or an INSERT file is specified.

`Log Frequency` k Default = 1
see `Print Frequency`

`LU factor tolerance` r_1 Default = 100.0
`LU update tolerance` r_2 Default = 10.0

These tolerances affect the stability and sparsity of the basis factorization $B = LU$ during refactorization and updating, respectively. They must satisfy r_1, $r_2 \geq 1.0$. The matrix L is a product of matrices of the form

$$\begin{pmatrix} 1 & \\ \mu & 1 \end{pmatrix},$$

where the multipliers μ satisfy $|\mu| \leq r_i$. Smaller values of r_i favor stability, while larger values favor sparsity. The default values usually strike a good compromise.

- For large and relatively dense problems, $r_1 = 10.0$ or 5.0 (say) may give a useful improvement in stability without impairing sparsity to a serious degree.
- For certain very regular structures (e.g., band matrices) it may be necessary to reduce r_1 and/or r_2 in order to achieve stability. For example, if the columns of A include a submatrix of the form

$$\begin{pmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & 4 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 4 & -1 \\ & & & & -1 & 4 \end{pmatrix},$$

  one should set both r_1 and r_2 to values in the range $1.0 \leq r_i < 4.0$.

`LU singularity tolerance` r_3 Default = $\epsilon^{2/3}$

This tolerance should satisfy $r_3 \leq \epsilon^{1/4} \approx 10^{-4}$. It helps guard against ill-conditioned basis matrices. When the LU factors of B are computed directly (not updated), the diagonal elements of U are tested as follows: if $|U_{jj}| \leq r_3$ or $|U_{jj}| < r_3 \max_i |U_{ij}|$, the j-th column of the basis is replaced by the corresponding slack variable. (Replacements are rare because the LU updating method is stable. They are most likely to occur during the first factorization.)

`LU density tolerance` r_1 Default = 0.6
`LU singularity tolerance` r_2 Default = $\epsilon^{2/3} \approx 10^{-11}$

The density tolerance r_1 is used during LU factorization of the basis matrix. Columns of L and rows of U are formed one at a time, and the remaining rows and columns of the basis

are altered appropriately. At any stage, if the density of the remaining matrix exceeds r_1, the Markowitz strategy for choosing pivots is altered to reduce the time spent searching for each remaining pivot. Raising the density tolerance towards 1.0 may give slightly sparser LU factors, with a slight increase in factorization time.

The singularity tolerance r_2 helps guard against ill-conditioned basis matrices. When the basis is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \leq r_2$ or $|U_{jj}| < r_2 \max_i |U_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular. (For example, a whole row of the Jacobian could be zero at an optimal solution.) Before exact singularity occurs, the basis could become very ill-conditioned and the optimization could progress very slowly (if at all). Setting $r_2 =$ `1.0e-5`, say, may help cause a judicious change of basis.

`Major feasibility tolerance` ϵ_r Default = `1.0E-6`

This specifies how accurately the nonlinear constraints should be satisfied. The default value of `1.0E-6` is appropriate when the linear and nonlinear constraints contain data to about that accuracy.

Let `rowerr` be defined as the maximum component of the constraint residual vector $F(x) + A_1 y - b_1$, normalized by the size of the solution. Thus,

$$\texttt{rowerr} = \|F(x) + A_1 y - b_1\|_\infty / \texttt{xnorm}, \tag{7.1}$$

where `xnorm` is a measure of the size of the basic and superbasic variables. The solution (x, y) is regarded as acceptably feasible if $\texttt{rowerr} \leq \epsilon_r$.

At each major iteration, `rowerr` appears in the output to the PRINT and SUMMARY files as the quantity labeled "`Feasbl`".

If some of the problem functions are known to be of low accuracy, a larger `Major feasibility tolerance` may be appropriate.

`Major optimality tolerance` ϵ Default = `1.0E-6`

This specifies the final accuracy of the dual variables. On successful termination, SNOPT will have computed a primal and dual solution such that $\texttt{maxgap} \leq \epsilon$, where `maxgap` is the scaled maximum complementarity gap:

$$\texttt{maxgap} = \max_{1 \leq i \leq n+m} \texttt{gap}_j / \|\pi\|, \tag{7.2}$$

and $\texttt{gap}_j$ is an estimate of the complementarity gap for variable j. $\texttt{gap}_j$ is computed using the reduced gradients $d_j = g_j - \pi^T a_j$, where g_j is the jth component of the objective gradient, a_j is the associated column of the constraint matrix $(A \quad -I)$, and π is the set of QP dual variables. If d_j is zero, $\texttt{gap}_j$ is undefined; otherwise

$$\texttt{gap}_j = \begin{cases} d_j \min\{x_j - l_j, 1\} & \text{if } d_j > 0; \\ -d_j \min\{u_j - x_j, 1\} & \text{if } d_j < 0. \end{cases}$$

- At each major iteration, `maxgap` appears in the output to the PRINT and SUMMARY files as the quantity labeled "`Optimal`".
- For linear programming (i.e., `nnObj` $= 0$ and `nnCon` $= 0$), $\texttt{gap}_j$ is the optimal LP complementarity gap at the end of the first major iteration.

- In the case of unconstrained or linearly constrained optimization, the major iterations are terminated when `maxgap` $\leq \epsilon$. This is equivalent to requiring that the infinity norm of the reduced gradient for the superbasic variables be less than ϵ.
- When there are nonlinear constraints, $\texttt{gap}_j$ is an estimate of how much the jth dual variable has the "wrong sign". It is therefore an estimate of how much the merit function can be improved in subsequent iterations.
- In the above test, $\|\pi\|$ is a measure of the size of the dual variables. It is included to make the tests independent of a scale factor on the objective function. (see the definition of $\|\pi\|$ for the `Minor optimality tolerance`.

`Major Iteration limit` k Default = 20

This is the maximum number of major iterations allowed. It is intended to guard against an excessive number of linearizations of the constraints.

For preliminary runs on a new problem, a fairly low `Major iterations` limit should be specified (e.g., `10` or `20`).

`Major print level` p Default = 00001

This varies the amount of information that will be output to the printer file. It is independent of the `Print frequency`. Typical values are

```
Major print level          1
```

which gives normal output for linear and nonlinear problems, and

```
Major print level         11
```

which in addition gives details of the Jacobian factorization that commences each major iteration.

In general, the value being specified is best thought of as a binary number of the form

```
Major print level    JFDXbs
```

where each letter stands for a digit that is either `0` or `1`. The quantities referred to are:

- `s` a single line that gives a summary of each major iteration. (This entry in `JFMXbs` is not strictly binary since the summary line is printed whenever `JFMXbs` ≥ 1).
- `b` BASIS statistics, i.e., information relating to the basis matrix whenever it is refactorized. (This output is always provided if `JFMXbs` ≥ 10).
- `X` x_k, the nonlinear variables involved in the objective function or the constraints.
- `D` π_k, the dual variables for the nonlinear constraints.
- `F` $F(x_k)$, the values of the nonlinear constraint functions.
- `J` $J(x_k)$, the Jacobian matrix.

To obtain output of any of the items `JFMX`, set the corresponding digit to `1`, otherwise to `0`.

If `J=1`, the Jacobian matrix will be output column-wise at the start of each major iteration. Column j will be preceded by the value of the corresponding variable x_j and a key to indicate whether the variable is basic, superbasic or nonbasic. (Hence if `J = 1`, there

is no reason to specify `X=1` unless the objective contains more nonlinear variables than the Jacobian.) A typical line of output is

```
3  1.250000D+01BS      1  1.00000E+00      4  2.00000E+00
```

which would mean that x_3 is basic at value 12.5, and the third column of the Jacobian has elements of 1.0 and 2.0 in rows 1 and 4.

`Major print level 0` may be used to suppress most output, including page ejects between major iterations. (Error messages will not be suppressed.) This print level should be used only for production runs on well understood models. A high `Print frequency` may also be appropriate for such cases, e.g. `100` or `1000`. (For convenience, `Print frequency 0` may be used as shorthand for `Print frequency 99999`.)

`Minor print level` k Default = 1

This controls the amount of printing done during the solution of the QP subproblem. Printing is controlled by choosing k as follows.

0	No minor iteration output except error messages.
≥ 1	A single line of output each minor iteration (controlled by `Print frequency`).
≥ 10	Basis factorization statistics generated during the periodic refactorization of the basis (see `Factorization frequency`). Statistics associated with the first factor is controlled by the `Major print level`.

`Major step limit` r Default = 2.0

This parameter limits the change in x during a linesearch. It applies to all nonlinear problems, once a "feasible solution" or "feasible subproblem" has been found.

1. A linesearch determines a step α over the range $0 < \alpha \leq \beta$, where β is 1 if there are nonlinear constraints, or the step to the nearest upper or lower bound on x if all the constraints are linear. Normally, the first steplength tried is $\alpha_1 = \min(1, \beta)$.

2. In some cases, such as $f(x) = ae^{bx}$ or $f(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow. The parameter r is therefore used to define a limit $\bar{\beta} = r(1 + \|x\|)/\|p\|$, and the first evaluation of $f(x)$ is at the potentially smaller steplength $\alpha_1 = \min(1, \bar{\beta}, \beta)$.

3. Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at meaningless points. The `Major step limit` provides an additional safeguard. The default value $r = 2.0$ should not affect progress on well behaved problems, but setting $r = 0.1$ or 0.01 may be helpful when rapidly varying functions are present. A "good" starting point may be required. An important application is to the class of nonlinear least-squares problems.

4. In cases where several local optima exist, specifying a small value for r may help locate an optimum near the starting point.

`Minimize` Default
`Maximize`

This specifies the required direction of optimization. It applies to both linear and nonlinear terms in the objective.

`Minor Iterations limit` k Default = 40

This is the maximum number of iterations allowed between successive linearizations of the nonlinear constraints. A moderate value (e.g., $\texttt{10} \leq k \leq \texttt{50}$) prevents excessive effort being expended on early major iterations, but allows later subproblems to be solved to completion.

In general it is unsafe to specify a value as small as $k = \texttt{1}$ or 2. (Even when an optimal solution has been reached, a few minor iterations may be needed for the corresponding subproblem to be recognized as optimal.)

Note that an independent limit on total iterations should be specified by the `Iterations` keyword as usual. If the problem is linearly constrained, this is the *only* limit (i.e., the `Minor Iterations` keyword is ignored).

`Minor feasibility tolerance` t Default = `1.0E-6`

A *feasible QP* is one in which all variables satisfy their upper and lower bounds to within the absolute tolerance t. (This includes slack variables. Hence, the general linear constraints are also satisfied to within t.)

- SNOPT attempts to find a feasible point for the QP subproblem before optimizing the objective. If the sum of infeasibilities cannot be reduced to zero, the problem is declared INFEASIBLE. SNOPT is then in elastic mode thereafter (with only the linearized nonlinear constraints defined to be elastic). Let `sInf` be the corresponding sum of infeasibilities. If `sInf` is quite small, it may be appropriate to raise t by a factor of 10 or 100. Otherwise, some error in the data should be suspected.
- *Note*: if `sInf` is not small, there may be other points that have a *significantly smaller sum of infeasibilities*. SNOPT does not attempt to find the solution that minimizes the sum.
- If `scale` is used, feasibility is defined in terms of the scaled problem (since it is then more likely to be meaningful).
- A nonlinear objective function $f(x)$ will be evaluated only at feasible points. If there are regions where $f(x)$ is undefined, every attempt should be made to eliminate these regions from the problem. For example, if $f(x) = \sqrt{x_1} + \log x_2$, it is essential to place lower bounds on both variables. If `Feasibility tolerance` = `1.0E-6`, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious; in general, keep x as far away from singularities as possible.)
- Bounds should also be used to keep x more than t away from singularities in $f(x)$.
- If there are nonlinear constraints, each major iteration attempts to satisfy their linearization to within the tolerance t. If this is not possible, the bounds on the nonlinear constraints are relaxed temporarily (in several stages).
- Feasibility with respect to the nonlinear constraints themselves is measured against the `Major feasibility tolerance` (not against t). The relevant test is made at the *start* of a major iteration.

`Minor optimality tolerance` t Default = `1.0E-6`

This is used to judge the size of the QP reduced gradients $d_j = g_j - \pi^T a_j$, where g_j is the jth component of the QP gradient, a_j is the associated column of the QP constraint matrix, and π is the set of QP dual variables.

- By construction, the reduced gradients for basic variables are always zero. The QP subproblem will be declared optimal if the reduced gradients for nonbasic variables at their lower or upper bounds satisfy

$$d_j/\|\pi\| \geq -r \quad \text{or} \quad d_j/\|\pi\| \leq r$$

  respectively, and if $|d_j|/\|\pi\| \leq r$ for superbasic variables.

- In the above tests, $\|\pi\|$ is a measure of the size of the dual variables. It is included to make the tests independent of a scale factor on the objective function.

- The quantity $\|\pi\|$ actually used is defined by

$$\|\pi\| = \max\{\sigma/\sqrt{m}, 1\}, \quad \text{where} \quad \sigma = \sum_{i=1}^{m} |\pi_i|,$$

  so that only *large* scale factors are allowed for.

- If the objective is scaled down to be very *small*, the optimality test reduces to comparing d_j against $0.01r$.

`Minor print level` k Default = 0

This controls the amount of printing produced during the minor iterations. The levels of printing available are as follows.

0	No minor iteration output except error messages.
≥ 1	Brief problem statistics, a single line of output each iteration (controlled by `Print frequency`), and the exit condition for the QP subproblem.
≥ 10	Factorization statistics associated with basis factorizations controlled by `Factorize frequency`..

`New basis file` f Default = 0

If $f > 0$, a basis map will be saved on file f every kth iteration, where k is the `Save frequency`.

- The first card of the file will contain the word `PROCEEDING` if the run is still in progress.

- If $f > 0$, a basis map will also be saved at the end of a run, with some other word indicating the final solution status.

`Old basis file` f Default = 0

If $f > 0$, the starting point will be obtained from this file in the format of Section 9.1.

- The file will usually have been output previously as a `NEW BASIS FILE`.

- The file will not be acceptable if the number of rows or columns in the problem has been altered.

`Optimality tolerance` t Default = `1.0e-6`
see `Major optimality tolerance`

`Partial price` i Default = 10 (LP) or 1 (NLP)

This parameter is recommended for large problems that have significantly more variables than constraints. It reduces the work required for each "pricing" operation (when a nonbasic variable is selected to become superbasic).

- When $i = 1$, all columns of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$ are searched.
- Otherwise, A and I are partitioned to give i roughly equal segments A_j, I_j ($j = 1$ to i). If the previous pricing search was successful on A_j, I_j, the next search begins on the segments A_{j+1}, I_{j+1}. (All subscripts here are modulo i.)
- If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to become superbasic. If nothing is found, the search continues on the next segments A_{j+2}, I_{j+2}, and so on.
- `Partial price` t (or $t/2$ or $t/3$) may be appropriate for time-stage models having t time periods.

`Pivot tolerance` r Default = $\epsilon^{2/3}$

Broadly speaking, the pivot tolerance is used to prevent columns entering the basis if they would cause the basis to become almost singular.

- When x changes to $x+\alpha p$ for some search direction p, a "ratio test" is used to determine which component of x reaches an upper or lower bound first. The corresponding element of p is called the pivot element.
- For linear problems, elements of p are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance r.
- For nonlinear problems, elements smaller than $r\|p\|$ are ignored.
- It is common for two or more variables to reach a bound at essentially the same time. In such cases, the `Minor Feasibility tolerance` (say t) provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively small values of t should therefore not be specified.
- To a lesser extent, the `Expand frequency` (say f) also provides some freedom to maximize the pivot element. Excessively *large* values of f should therefore not be specified.

`Print frequency` k Default = 1

One line of the iteration log will be printed every kth minor iteration. A value such as $k =$ `10` is suggested for those interested only in the final solution.

`Punch file` f Default = 0

If $f > 0$, the final solution obtained will be output to file f in the format described in Section 9.2. For linear programs, this format is compatible with various commercial systems.

`Save frequency` k Default = 100

If a NEW BASIS file has been specified, a basis map describing the current solution will be saved on the appropriate file every kth iteration. A BACKUP BASIS file will also be saved if specified.

`Scale option`	i	Default = 2 (LP) or 1 (NLP)
`Scale`	`Yes`	
`Scale`	`No`	
`Scale linear variables`	`* Same as Scale option 1`	
`Scale nonlinear variables`	`* Same as Scale option 2`	
`Scale all variables`	`* Same as Scale option 2`	
`Scale tolerance`	r	Default = 0.9
`Scale, Print`		
`Scale, Print, Tolerance =`	r	Default = 0.9

Three scale options are available as follows:

i	*Meaning*
0	No scaling. If storage is at a premium, this option will save $m+n$ words of workspace.
1	Linear constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0 (see Fourer [4]). This will sometimes improve the performance of the solution procedures.
2	All constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $\begin{pmatrix} A & -I \end{pmatrix}$ that are fixed or have positive lower bounds or negative upper bounds.

`Scale option 2` is the default for linear programs.

`Scale option 1` is the default for nonlinear problems. (Only linear variables are scaled.)

`Scale Yes` sets the default. (*Caution*: If all variables are nonlinear, `Scale Yes` unexpectedly does *nothing*, because there are no linear variables to scale.)

`Scale No` suppresses scaling (equivalent to `Scale option 0`).

If nonlinear constraints are present, `Scale option 1` or `0` should generally be tried at first. `Scale option 2` gives scales that depend on the initial Jacobian, and should therefore be used only if (a) a good starting point is provided, and (b) the problem is not highly nonlinear.

`Scale, Print` causes the row-scales $r(i)$ and column-scales $c(j)$ to be printed. The scaled matrix coefficients are $\bar{a}_{ij} = a_{ij}c(j)/r(i)$, and the scaled bounds on the variables and slacks are $\bar{l}_j = l_j/c(j)$, $\bar{u}_j = u_j/c(j)$, where $c(j) \equiv r(j-n)$ if $j > n$.

All forms except `Scale option` may specify a tolerance r, where $0 < r < 1$ (for example: `Scale, Print, Tolerance = 0.99`). This affects how many passes might be needed through the constraint matrix. On each pass, the scaling procedure computes the ratio of the largest and smallest nonzero coefficients in each column:

$$\rho_j = \max_i |a_{ij}| / \min_i |a_{ij}| \qquad (a_{ij} \neq 0).$$

If $\max_j \rho_j$ is less than r times its previous value, another scaling pass is performed to adjust the row and column scales. Raising r from 0.9 to 0.99 (say) usually increases the number of scaling passes through A. At most 10 passes are made.

If a `Scale option` has not already been specified, `Scale, Print` or `Scale tolerance` both set the default scaling.

`Solution`	`Yes`	
`Solution`	`No`	
`Solution If Optimal, Infeasible, or Unbounded`		
`Solution File`	f	Default = 0

The first four options determine whether the final solution obtained is to be output to the PRINT file. The `FILE` option operates independently; if $f > 0$, the final solution will be output to file f (whether optimal or not).

- For the `Yes`, `If optimal`, and `If error` options, floating-point numbers are printed in `f16.5` format, and "infinite" bounds are denoted by the word `NONE`.
- For the `File` option, all numbers are printed in `1pe16.6` format, including "infinite" bounds which will have magnitude `1.000000E+20`.
- To see more significant digits in the printed solution, it will sometimes be useful to make f refer to the system PRINT file.

`Start Objective Check at Column`	k	Default = 1
`Start Constraint Check at Column`	k	Default = 1
`Stop Objective Check at Column`	l	Default = n_1'
`Stop Constraint Check at Column`	l	Default = n_1''

These keywords may be used to abbreviate the verification of individual gradient elements computed by subroutines `funobj` and `funcon`. For example:

- If the first 100 objective gradients appeared to be correct in an earlier run, and if you have just found a bug in `funobj` that ought to fix up the 101-th component, then you might as well specify `Start Objective Check at Column 101`. Similarly for columns of the Jacobian matrix.
- If the first 100 variables occur nonlinearly in the constraints, and the remaining variables are nonlinear only in the objective, then `funobj` must set the first 100 components of `g(*)` to zero, but these hardly need to be verified. The above data card would again be appropriate.

These keywords are effective if `Verify Level` > 0.

`Summary File`	f	Default = 0
`Summary Frequency`	k	Default = 100

If $f > 0$, a brief log will be output to file f, including one line of information every kth iteration. In an interactive environment, it is useful to direct this output to the terminal, to allow a run to be monitored on-line. (If something looks wrong, the run can be manually terminated.) Further details are given in Section 6.6.

`Superbasics Limit` i Default = 50

This places a limit on the storage allocated for superbasic variables. Ideally, i should be set slightly larger than the "number of degrees of freedom" expected at an optimal solution.

For linear programs, an optimum is normally a basic solution with no degrees of freedom. (The number of variables lying strictly between their bounds is no more than m, the number of general constraints.) The default value of i is therefore 1.

For nonlinear problems, the number of degrees of freedom is often called the "number of independent variables".

- Normally, i need not be greater than $n_1 + 1$, where n_1 is the number of nonlinear variables.
- For many problems, i may be considerably smaller than n_1. This will save storage if n_1 is very large.
- This parameter also sets the `Reduced Hessian dimension`, unless the latter is specified explicitly (and conversely).

`Suppress Parameters`

Normally SNOPT prints the SPECS file as it is being read, and then prints a complete list of the available keywords and their final values. The `Suppress Parameters` option tells SNOPT not to print the full list.

`Unbounded Objective Value` $f_{\max}$ Default = `1.0E+15`
`Unbounded Step Size` $\alpha_{\max}$ Default = `1.0E+18`

These parameters are intended to detect unboundedness in nonlinear problems. (They may or may not achieve that purpose!) During a line search, f is evaluated at points of the form $x+\alpha p$, where x and p are fixed and α varies. if $|f|$ exceeds $f_{\max}$ or α exceeds $\alpha_{\max}$, iterations are terminated with the exit message `problem is unbounded (or badly scaled)`.

- If singularities are present, unboundedness in $f(x)$ may be manifested by a floating-point overflow (during the evaluation of $f(x + \alpha p)$), before the test against $f_{\max}$ can be made.
- Unboundedness in x is best avoided by placing finite upper and lower bounds on the variables.

`Verify Level` l_1 Default = 0
`Verify` `No`
`Verify Level` `0`
`Verify Objective Gradients`
`Verify Level` `1`
`Verify Constraint Gradients`
`Verify Level` `2`
`Verify`
`Verify` `Yes`
`Verify Gradients`
`Verify Level` `3`

These keywords refer to finite-difference checks on the gradient elements computed by the user subroutines `funobj` and `funcon`. It is possible to specify `Verify Levels 0`–`3` in several

ways, as indicated above. For example, the nonlinear objective gradients (if any) will be verified if either `Verify Objective` or `Verify Level 1` is specified. Similarly, both the objective and the constraint gradients will be verified if `Verify Yes` or `Verify Level 3` or just `Verify` is specified.

If $0 \leq l_1 \leq 3$, gradients will be verified at the first point reached that satisfies the linear constraints and the upper and lower bounds. The current linearization of the nonlinear constraints must also be satisfied. If $l_1 = 0$, only a "cheap" test will be performed, requiring 3 calls to `funobj` or 2 calls to `funcon`. If $1 \leq l_1 \leq 3$, a more reliable check will be made on individual gradient components, within the ranges specified by the `start` and `stop` keywords. A key of the form "`OK`" or "`BAD?`" indicates whether or not each component appears to be correct.

- `Verify level 3` should be specified whenever a new function routine is being developed.
- Missing gradients are not checked; i.e., they result in no overhead.
- The default action is to perform a cheap check on the gradients at the first feasible point. Even on debugged function routines, the message "`Gradients seem to be OK`" will provide certain comfort at nominal expense.
- If necessary, checking can be suppressed by specifying `Verify Level -1` .

`Total real workspace`	`maxrw`	Default = `lenrw`
`Total integer workspace`	`maxiw`	Default = `leniw`
`Total character workspace`	`maxcw`	Default = `lencw`
`User real workspace`	`maxru`	Default = 0
`User integer workspace`	`maxiu`	Default = 0
`User character workspace`	`maxcu`	Default = 0

These keywords define the limits of the region of storage that SNOPT may use in solving the current problem. The main work arrays are declared in the calling program, along with their lengths, by statements of the form

```
data                lencw/500/
character*8      cw(lencw)
data                leniw/25000/
integer          iw(leniw)
data                lenrw/25000/
real             rw(lenrw)
```

where the actual lengths of `iw` and `rw` must be specified at compile time.

The parameters `maxcu`, `maxiu` and `maxru` are relevant if you want to call `snopt` while using `cw`, `iw` and `iw` to hold user information. This would be necessary, for example, if you wish to access `snopt` internal variables in `funobj` or `funcon`. The values specified by the `workspace` keywords are stored in

```
maxru = iw(  2)
maxrw = iw(  3)
maxiu = iw(  4)
maxiw = iw(  5)
maxcu = iw(  6)
maxcw = iw(  7)
```

and workspace may be shared according to the following rules:

- `cw(1:500)`, viw(1:500) and `rw(1:500)` are used by SNOPT, and must not be altered.
- `cw(501:maxcu)`, `iw(501:maxiu)` and `rw(501:maxru)` are available to the user.
- `cw(maxcu+1:maxcw)`, `iw(maxiu+1:maxiw)` and `rw(maxru+1:maxrw)` are used by SNOPT, and must not be altered.
- `cw(maxcw+1:lencw)`, `iw(maxiw+1:leniw)` and `rw(maxrw+1:lenrw)` are unused (or available to the user).

The `workspace` parameters are most useful on machines with a virtual (paged) store. Some systems will allow `leniw` and `lenrw` to be set to very large numbers (say `500000`) with no overhead in saving the resulting object code. At run time, when various problems of different size are to be solved, it may be sensible to confine SNOPT to a portion of `iw` and `rw` to reduce paging activity slightly. (However, SNOPT accesses storage contiguously wherever possible, so the benefit may be slight. In general it is far better to have too much storage than not enough.)

8. Output

The following information is output to the PRINT file during the solution of each problem referred to in the SPECS file.

- A listing of the relevant part of the SPECS file.
- A listing of the parameters that were or could have been set in the SPECS file.
- An estimate of the amount of working storage needed, compared to how much is available.
- Some statistics about the problem being solved.
- The amount of storage available for the LU factorization of the basis matrix.
- A summary of the scaling procedure, if `Scale` was specified.
- Notes about the initial basis resulting from a CRASH procedure or a BASIS file.
- The major iteration log.
- The minor iteration log.
- Basis factorization statistics.
- The EXIT condition and some statistics about the solution obtained.
- The printed solution, if requested.

The last five items are described in the following sections. Further brief output may be directed to the SUMMARY file, as discussed in §8.8.

8.1. The major iteration Log

If `Major print level` > 0, one line of information is output to the PRINT file every kth minor iteration, where k is the specified `Print frequency` (default $k =$ `1`).

Label	*Description*
`Maj`	The current major iteration number.
`Mnr`	is the number of iterations required by both the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see §2).
`Step`	The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.
`nObj`	The number of times subroutine `funobj` has been called to evaluate the nonlinear objective function. Evaluations needed for the estimation of the gradients by finite differences are not included. `nObj` is printed as a guide to the amount of work required for the linesearch.
`nCon`	The number of times subroutine `funcon` has been called to evaluate the nonlinear constraint functions.

`Merit` is the value of the augmented Lagrangian merit function (see (2.1)). This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see §2). As the solution is approached, `Merit` will converge to the value of the objective at the solution.

In elastic mode, the merit function is a composite function involving the constraint violations weighted by the elastic weight.

If there are no nonlinear constraints (i.e., `nnCon` = 0), this entry contains `Objective`, the value of the objective function. In this case, the objective will decrease monotonically to its optimal value.

`Feasbl` is the value of `rowerr`, the maximum component of the scaled nonlinear constraint residual (7.1). The solution is regarded as acceptably feasible if `Feasbl` is less than or equal to the `Major feasibility tolerance`. `Feasbl` will be approximately zero in the neighborhood of a solution.

If there are no nonlinear constraints (i.e., `nnCon` = 0), all iterates are feasible and this entry is not printed.

`Optimal` is the value of `maxgap`, the maximum complementarity gap (7.2). `Optimal` is an estimate of the degree of nonoptimality of the reduced costs. It is approximately zero in the neighborhood of a solution.

`nS` The current number of superbasic variables.

`Penalty` is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if `nnCon` is zero).

`LU` The number of nonzeros representing the basis factors L and U on completion of the QP subproblem.

If nonlinear constraints are present, the basis factorization $B = LU$ is computed at the start of the first minor iteration. At this stage, `LU` = `lenL` + `lenU`, where `lenL`, the number of subdiagonal elements in the columns of a lower triangular matrix and `lenU` is the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix.

As columns of B are replaced during the minor iterations, `LU` may fluctuate up or down; but in general it will tend to increase. As the solution is approached and the minor iterations settle down `LU` will reflect the number of nonzeros after the first LU.

If there are only linear constrants, refactorization will be subject only to the optional parameter `Factorize frequency` and `LU` will tend to increase between refactorizations.

`Swp`

`Cond Hz` An estimate of the condition number of the reduced Hessian of the Lagrangian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R. This constitutes a lower bound on the condition number of the matrix $R^T R$ that approximates the reduced Hessian. `Cond Hz` gives a rough indication of whether or not the optimization procedure is having difficulty. If ϵ is the relative precision of the machine being used, the SQP algorithm will make slow progress if `Cond Hz` becomes as large as $\epsilon^{-1/2}$, and will probably fail to find a better solution if `Cond Hz` reaches $\epsilon^{-3/4}$ or larger. (On most machines, these values are about 10^8 and 10^{12}.)

To guard against high values of `Cond Hz`, attention should be given to the scaling of the variables and the constraints. In some cases it may be necessary to add upper or lower bounds to certain variables to keep them a reasonable distance from singularities in the nonlinear functions or their derivatives.

`Opt` is a two-letter indication of the status of the two convergence tests involving the feasibility and optimality of the iterates (the tests indicated by (7.2) and 7.1) defined in the description of `Major feasibility tolerance` and `Major optimality tolerance`. Each letter is "`T`" if the test is satisfied, and "`F`" otherwise.

If either of the indicators is "`F`" when `snopt` terminates with `inform` $= 0$, the user should check the solution carefully.

8.2. The minor iteration Log

If `Minor print level` > 0, one line of information is output to the PRINT file every kth minor iteration, where k is the specified `Minor print frequency` (default $k =$ `1`). A heading is printed before the first such line following a basis factorization. The heading contains the items described below. In this description, a PRICE operation is defined to be the process by which one or more nonbasic variables are selected to become superbasic (in addition to those already in the superbasic set). The variable selected will be denoted by `jq`. If the problem is purely linear, variable `jq` will usually become basic immediately (unless it should happen to reach its opposite bound and return to the nonbasic set).

If `Partial price` is in effect, variable `jq` is selected from $A_{\texttt{pp}}$ or $I_{\texttt{pp}}$, the `pp`th segments of the constraint matrix $(A \quad -I)$.

Label — *Description*

`Itn` The current iteration number.

`pp` The Partial Price indicator. The variable selected by the last PRICE operation came from the `pp`th partition of A and $-I$. `pp` is set to zero when the basis is refactored.

`dj` This is `dj`, the reduced cost (or reduced gradient) of the variable `jq` selected by PRICE at the start of the present iteration. Algebraically, `dj` is $d_j = g_j - \pi^T a_j$ for $j =$ `jq`, where g_j is the gradient of the current objective function, π is the vector of dual variables, and a_j is the jth column of the constraint matrix $(A \quad -I)$.

Note that `dj` is the norm of the reduced-gradient vector at the start of the iteration, just after the PRICE operation.

`+SBS` The variable `jq` selected by PRICE to be added to the superbasic set.

`-SBS` The variable chosen to leave the set of superbasics. It has become basic if the entry under `-B` is nonzero; otherwise it has become nonbasic.

`-BS` The variable removed from the basis (if any) to become nonbasic.

`-B` The variable removed from the basis (if any) to swap with a slack variable made superbasic by the latest PRICE. The swap is done to ensure that there are no superbasic slacks.

`Step` The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. If a variable is made superbasic during the current iteration (i.e., `+SBS` is positive), `Step` will be the step to the nearest bound. During phase 2, the step can be greater than one only if the reduced Hessian is not positive definite.

`Pivot` If column a_q replaces the rth column of the basis B, `Pivot` is the rth element of a vector y satisfying $By = a_q$. Wherever possible, `Step` is chosen to avoid extremely small values of `Pivot` (since they cause the basis to be nearly singular). In rare cases, it may be necessary to increase the `Pivot tolerance` to exclude very small elements of y from consideration during the computation of `Step`.

`L` The number of nonzeros representing the basis factor L. Immediately after a basis factorization $B = LU$, this is `lenL`, the number of subdiagonal elements in the columns of a lower triangular matrix. Further nonzeros are added to `L` when various columns of B are later replaced. (Thus, `L` increases monotonically.)

`U` The number of nonzeros in the basis factor U. Immediately after a basis factorization, this is `lenU`, the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `U` may fluctuate up or down; in general it will tend to increase.

`ncp` The number of compressions required to recover storage in the data structure for U. This includes the number of compressions needed during the previous basis factorization. Normally `ncp` should increase very slowly. If not, the amount of integer and real workspace available to SNOPT should be increased by a significant amount. As a suggestion, the work arrays `iw(*)` and `rw(*)` should be extended by `L + U` elements.

`nInf` The number of infeasibilities *before* the present iteration. This number will not increase unless the iterations are in elastic mode.

`Sinf,Objective` If `nInf` > 0, this is `sInf`, the sum of infeasibilities before the present iteration. (It will usually decrease at each nonzero `Step`, but if `nInf` decreases by 2 or more, `sInf` may occasionally increase. However, in elastic mode, it will decrease monotonically.)

Otherwise, it is the value of the current objective function *after* the present iteration.

Note: in elastic mode, the heading is `Composite Obj`.

The following items are printed if the problem is nonlinear or if the superbasic set is nonempty (i.e., if the current solution is nonbasic).

Label *Description*

`Norm rg` This quantity is `rg`, the norm of the reduced-gradient vector at the start of the iteration. (It is the Euclidean norm of the vector with elements d_j for variables j in the superbasic set. During phase 2 this norm will be approximately zero after a unit step.

`nS` The current number of superbasic variables.

`cond Hz` An estimate of the condition number of the reduced Hessian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R. This constitutes a lower bound on the condition number of the reduced Hessian $R^T R$.

To guard against high values of `cond Hz`, attention should be given to the scaling of the variables and the constraints.

8.3. Basis Factorization Statistics

When `Print Level` ≥ 20 and `Print file` > 0, the following lines of intermediate printout (< 120 characters) are produced on the unit number specified by `Print file` whenever the matrix B or $B_S = (B \;\; S)^T$ is factorized. Gaussian elimination is used to compute an LU factorization of B or B_S, where PLP^T is a lower triangular matrix and PUQ is an upper triangular matrix for some permutation matrices P and Q. This factorization is stabilized in the manner described under `LU factor tolerance` in §7.3.

Label — *Description*

`Factorize` The number of factorizations since the start of the run.

`Demand` A code giving the reason for the present factorization.

Code	*Meaning*
0	First LU factorization.
1	Number of updates reached the value of the optional parameter `Factorization Frequency`.
2	Excessive non-zeros in the updated factors.
7	Not enough storage to update factors.
10	Row residuals too large (see the description for `Check Frequency`).
11	Ill-conditioning has caused inconsistent results.

`Iteration` The current iteration number.

`Infeas` The number of infeasibilities at the *start* of the previous iteration.

`Objective` If `Infeas` > 0, this is the sum of infeasibilities at the start of the previous iteration.

If `Infeas` $= 0$, this is the value of the objective function *after* the previous iteration.

`Nonlinear` The number of nonlinear variables in the current basis B. (not printed if B_S is factorized).

`Linear` The number of linear variables in B. (not printed if B_S is factorized).

`Slacks` The number of slack variables in B. (not printed if B_S is factorized).

`Elems` The number of nonzero matrix elements in B. (not printed if B_S is factorized).

`Density` The percentage nonzero density of B, $100 \times$ `Elems`$/($`m`$\times$`m`$)$, where `m` is the number of rows in the problem (`m` $=$ `NONLINEAR` $+$ `Linear` $+$ `Slacks`).

`Comprssns` The number of times the data structure holding the partially factored matrix needed to be compressed, to recover unused storage. Ideally this number should be zero. If it is more than 3 or 4, the amount of workspace available to SNOPT should be increased for efficiency.

`Merit` The average Markowitz merit count for the elements chosen to be the diagonals of PUQ. Each merit count is defined to be $(c-1)(r-1)$ where c and r are the number of nonzeros in the column and row containing the element at the time it is selected to be the next diagonal. `Merit` is the average of `m` such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization.

`lenL` The number of nonzeros in L. On most machines, each nonzero is represented by one eight-byte `REAL` and two two-byte `integer` data types.

`lenU` The number of nonzeros in U. The storage required for each nonzero is the same as for the nonzeros of L.

`Increase` The percentage increase in the number of nonzeros in L and U relative to the number of nonzeros in B; i.e., $100 \times$ (`lenL` + `lenU` − `Elems`)/`Elems`.

`m` is the number of rows in the problem. Note that `m` = `Ut` + `Lt` + `bp`.

`Ut` is the number of triangular rows of B at the top of U.

`d1` is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.

`Lmax` The maximum subdiagonal element in the columns of L. This will be no larger than the `LU factor tolerance`.

`Bmax` The maximum nonzero element in B.

`Umax` The maximum nonzero element in U, excluding elements of B that remain in U unaltered. (For example, if a slack variable is in the basis, the corresponding row of B will become a row of U without alteration. Elements in such rows will not contribute to `Umax`. If the basis is strictly triangular, *none* of the elements of B will contribute, and `Umax` will be zero.)

Ideally, `Umax` should not be substantially larger than `Bmax`. If it is several orders of magnitude larger, it may be advisable to reduce the `LU factor tolerance` to some value nearer 1.0. (The default value is 10.0.)

`Umax` is not printed if B_S is factorized.

`Umin` The smallest *diagonal* element of PUQ in absolute magnitude.

`Growth` The ratio `Umax`/`Bmax`, which should not be too large (see above).

As long as `Lmax` is not large (say 10.0 or less), the ratio
max{`Bmax`, `Umax`} / `Umin` gives an estimate of the condition number of B. If this number is extremely large, the basis is nearly singular and some numerical difficulties could conceivably occur. (However, an effort is made to avoid near-singularity by using slacks to replace columns of B that would have made `Umin` extremely small. Messages are issued to this effect, and the modified basis is refactored.)

`Lt` is the number of triangular columns of B at the beginning of L.

`bp` is the size of the "bump" or block to be factorized nontrivially after the triangular rows and columns have been removed.

`d2` is the number of columns remaining when the density of the basis matrix being factorized reached 0.6.

8.4. Crash statistics

When `Print Level` $\geq$ 20 and `Print file` $>$ 0, the following CRASH statistics ($<$ 120 characters) are produced on the unit number specified by `Print file` whenever `Start` = 'Cold' (see §7.3). They refer to the number of columns selected by the CRASH procedure during each of several passes through A, whilst searching for a triangular basis matrix.

Label *Description*

`Slacks` is the number of slacks selected initially.

`Free cols` is the number of free columns in the basis.

`Preferred` is the number of "preferred" columns in the basis (i.e., $\texttt{hs}(j) = 3$ for some $j \leq n$).

`Unit` is the number of unit columns in the basis.

`Double` is the number of double columns in the basis.

`Triangle` is the number of triangular columns in the basis.

`Pad` is the number of slacks used to pad the basis.

8.5. EXIT Conditions

For each problem in the SPECS file, a message of the form `EXIT -- message` is printed to summarize the final result. Here we describe each message and suggest possible courses of action.

A number is associated with each message below. It is the final value assigned to the integer variable `inform`.

> The following messages arise when the SPECS file is found to contain no further problems.

`-2. EXIT -- input error. SNOPT encountered end-of-file or an endrun card before finding a SPECS file on unit nn`

The SPECS file may not be properly assigned. Its unit number `nn` is defined at compile time in subroutine `snInit`, and normally it is the system card input stream.

Otherwise, the SPECS file may be empty, or cards containing the keywords `Skip` or `Endrun` may imply that all problems should be ignored (see §6).

`-1. ENDRUN`

This message is printed at the end of a run if SNOPT terminates of its own accord. Otherwise, the operating system will have intervened for one of many possible reasons (excess time, missing file, arithmetic error in user routines, etc.).

> The following messages arise when optimization terminates gracefully. A solution exists, any of the BASIS files may be saved, and the solution will be printed and/or saved on the SOLUTION file if requested.

`0. EXIT -- optimal solution found`

This is the message we all hope to see! It is certainly preferable to every other message, and we naturally want to believe what it says, because this is surely one situation where *the computer knows best*. There may be cause for celebration if the objective function has reached an astonishingly new high (or low). Or perhaps it will signal the end of a strenuous series of runs that have iterated far into the night, depleting one's patience and computing funds to an equally alarming degree. (We hope not!)

In all cases, a distinct level of caution is in order, even if it can wait until next morning. For example, if the objective value *is* much better than expected, we may have obtained an optimal solution to the wrong problem! Almost any item of data could have that effect, if it has the wrong value or is entered in the wrong columns of an input record. There may be thousands of items of data in A, and the nonlinear functions (if any) could depend on

input files and other data in innumerable ways. Verifying that the problem has been defined correctly is one of the more difficult tasks for a model builder. It is good practice in the function subroutines to print any data that is read in on the first entry.

If nonlinearities exist, one must always ask the question: could there be more than one local optimum? When the constraints are linear and the objective is known to be convex (e.g., a sum of squares) then all will be well if we are *minimizing* the objective: a local minimum is a global minimum in the sense that no other point has a lower function value. (However, many points could have the *same* objective value, particularly if the objective is largely linear.) Conversely, if we are *maximizing* a convex function, a local maximum cannot be expected to be global, unless there are sufficient constraints to confine the feasible region.

Similar statements could be made about nonlinear constraints defining convex or concave regions. However, the functions of a problem are more likely to be neither convex nor concave. Our advice is always to specify a starting point that is as good an estimate as possible, and to include reasonable upper and lower bounds on all variables, in order to confine the solution to the specific region of interest. We expect modellers to *know something about their problem*, and to make use of that knowledge as they themselves know best.

One other caution about "`Optimal solution`"s. When nonlinearities are present, the final size of the reduced-gradient norm (`norm rg`) should be examined to see if it is reasonably small compared to the norm of the dual variables (`norm pi`). These quantities are printed following the `EXIT` message. SNOPT attempts to ensure that

$$\texttt{norm rg}/\texttt{norm pi} \leq \texttt{Major optimality tolerance}.$$

However, if messages of the form `XXX search terminated` occur at the end of the run, this condition will probably not have been satisfied. The final solution may or may not be acceptably close to optimal. Broadly speaking, if

$$\texttt{norm rg}\ /\texttt{norm pi} = 10^{-d},$$

then the objective function would probably change in the dth significant digit if optimization could be continued. One must judge whether or not d is sufficiently large.

`1. EXIT -- the problem is infeasible`

When the constraints are linear, this message can probably be trusted. Feasibility is measured with respect to the upper and lower bounds on the variables. The message tells us that among all the points satisfying the general constraints $Ax - s = b$, there is apparently no point that satisfies the bounds on x and s. Violations as small as the `Feasibility tolerance` are ignored, but at least one component of x or s violates a bound by more than the tolerance.

When nonlinear constraints are present, infeasibility is *much* harder to recognize correctly. Even if a feasible solution exists, the current linearization of the constraints may not contain a feasible point. In an attempt to deal with this situation, when solving the QP subproblem, SNOPT is prepared to relax the bounds on the slacks associated with nonlinear rows. If a QP subproblem proves to be infeasible or unbounded (or if the Lagrange multiplier estimates for the nonlinear constraints become large), SNOPT enters so-called "nonlinear elastic" mode. In elastic mode, the subproblem includes the original QP objective and the sum of the infeasibilities—suitably weighted using the "elastic weight". In elastic mode, some of the bounds on the nonlinear rows become "elastic"—i.e., they are allowed to violate their specified bounds. Variables subject to elastic bounds are known as *elastic variables*. An elastic variable is free to violate one or both of its original upper or lower bounds. If the original problem has a feasible solution and the elastic weight is sufficiently large, a feasible point eventually will be obtained for the perturbed constraints, and optimization can continue on the subproblem. If the nonlinear problem has no feasible solution, SNOPT

will tend to determine a "good" infeasible point if the elastic weight is sufficiently large. (If the elastic weight were infinite, SNOPT would locally minimize the nonlinear constraint violations subject to the linear constraints and bounds.)

Unfortunately, even though SNOPT locally minimizes the nonlinear constraint violations, there may still exist other regions in which the nonlinear constraints are satisfied. Wherever possible, nonlinear constraints should be defined in such a way that feasible points are known to exist when the constraints are linearized.

`2. EXIT -- the problem is unbounded (or badly scaled)`
For linear problems, unboundedness is detected by the simplex method when a nonbasic variable can apparently be increased or decreased by an arbitrary amount without causing a basic variable to violate a bound. A message prior to the EXIT message will give the index of the nonbasic variable. Consider adding an upper or lower bound to the variable. Also, examine the constraints that have nonzeros in the associated column, to see if they have been formulated as intended.

Very rarely, the scaling of the problem could be so poor that numerical error will give an erroneous indication of unboundedness. Consider using the `Scale` option.

For nonlinear problems, SNOPT monitors both the size of the current objective function and the size of the change in the variables at each step. If either of these is very large (as judged by the `Unbounded` parameters – see §7.3), the problem is terminated and declared UNBOUNDED. To avoid large function values, it may be necessary to impose bounds on some of the variables in order to keep them away from singularities in the nonlinear functions.

`3. EXIT -- too many iterations`
Either the `Iterations limit` or the `Major iterations limit` was exceeded before the required solution could be found. Check the iteration log to be sure that progress was being made. If so, restart the run using a basis file that was saved (or *should* have been saved!) at the end of the run.

`4. EXIT -- requested by user in subroutine funobj` (or `funcon`)
`after nnn calls`
This exit occurs if the subroutine parameter `Mode` is set to a negative number during some call to `funobj` or `funcon`. SNOPT assumes that you want the problem to be abandoned forthwith.

In some environments, this exit means that your subroutines were not successfully linked to SNOPTİf the default versions of `funobj` and `funcon` are ever called, they issue a warning message and then set `mode` to terminate the run. For example, you may have asked the operating system to

```
link  snopt, funobj, funcon
```

when in fact you should have said

```
link  funobj, funcon, snopt
```

(or something similar) to give your own subroutines priority. Most linkers or loaders retain the *first* version of any subprogram that they see.

`5. EXIT -- the superbasics limit is too small: nnn`
The problem appears to be more nonlinear than anticipated. The current set of basic and superbasic variables have been optimized as much as possible and a PRICE operation is necessary to continue, but there are already `nnn` superbasics (and no room for any more).

In general, raise the `Superbasics limit` s by a reasonable amount, bearing in mind the storage needed for the reduced Hessian. (The `Hessian dimension` h will also increase to s

unless specified otherwise, and the associated storage will be about $\frac{1}{2}s^2$ words.) In extreme cases you may have to set $h < s$ to conserve storage, but beware that the rate of convergence will probably fall off severely.

`7. EXIT -- subroutine funobj seems to be giving incorrect gradients`
A check has been made on some individual elements of the gradient array, and at least one component `gObj(`j`)` is being set to a value that disagrees markedly with a forward-difference estimate of $\partial f/\partial x_j$. (The relative difference between the computed and estimated values is 1.0 or more.) This exit is a safeguard, since SNOPT will usually fail to make progress when the computed gradients are seriously inaccurate. In the process it may expend considerable effort before terminating with Exit 9 below.

Check the function and gradient computation *very carefully*. A simple omission (such as forgetting to divide `f` by 2) could explain everything. If `fObj` or `gObj(`j`)` is very large, then give serious thought to scaling the function or the nonlinear variables.

If you feel *certain* that the computed `g(`j`)` is correct (and that the forward-difference estimate is therefore wrong), you can specify `Verify level 0` to prevent individual elements from being checked. However, the optimization procedure is likely to terminate unsuccessfully.

`8. EXIT -- subroutine funcon seems to be giving incorrect gradients`
This is analogous to the preceding exit. At least one of the computed Jacobian elements is significantly different from an estimate obtained by forward-differencing the constraint vector $F(x)$. Follow the advice given above, trying to ensure that the arrays `fcon` and `gCon` are being set correctly in subroutine `funcon`.

`9. EXIT -- the current point cannot be improved upon`
Several circumstances could lead to this exit.

1. Subroutine `funobj` and/or subroutine `funcon` could be returning accurate function values but inaccurate gradients (or *vice versa*). This is the most likely cause. Study the comments given for exits 7 and 8, and do your utmost to ensure that the subroutines are coded correctly.

2. The function and gradient values could be consistent, but their precision could be too low. For example, accidental use of a single-precision data type when double precision was intended throughout, would lead to a relative function precision of about 10^{-6} instead of something like 10^{-15}. The default `Optimality tolerance` of 10^{-6} would need to be raised to about 10^{-3} for optimality to be declared (at a rather suboptimal point). Of course, it is better to revise the function coding to obtain as much precision as economically possible.

3. If function values are obtained from an expensive iterative process, they may be accurate to rather few significant figures, and gradients will probably not be available. One should specify

   `Function precision` t

   `Major optimality tolerance` $\sqrt{t}$

   but even then, if t is as large as 10^{-5} or 10^{-6} (only 5 or 6 significant figures), the same exit condition may occur. At present the only remedy is to increase the accuracy of the function calculation.

`10. EXIT -- cannot satisfy the general constraints`
An LU factorization of the basis has just been obtained and used to recompute the basic variables x_B, given the present values of the superbasic and nonbasic variables. A single step

of "iterative refinement" has also been applied to increase the accuracy of x_B. However, a row check has revealed that the resulting solution does not satisfy the current constraints $Ax - s = b$ sufficiently well.

This probably means that the current basis is very ill-conditioned. Request the `Scale` option if there are any linear constraints and variables.

For certain highly structured basis matrices (notably those with band structure), a systematic growth may occur in the factor U. Consult the description of `Umax`, `Umin` and `Growth` in §8.3, and set the `LU factor tolerance` to 2.0 (or possibly even smaller, but not less than 1.0).

> If the following exits occur during the *first* basis factorization, the basic variables x_B will have certain default values that may not be particularly meaningful, and the dual vector π will be zero. BASIS files will be saved if requested, but certain values in the printed solution will not be meaningful. The problem will be terminated.

`20. EXIT -- not enough integer/real storage for the basis factors`
The main integer or real storage array `iw(*)` or `rw(*)` is apparently not large enough for this problem. The routine declaring `rw` should be recompiled with a larger dimension for `rw` or `iw`. The new value should also be assigned to `leniw` or `lenrw`.

In some cases it may be sufficient to increase the specified `Workspace (user)`, if it is currently less than `Workspace (total)`.

An estimate of the additional storage required is given in messages preceding the exit.

`21. EXIT -- error in basis package`
A preceding message will describe the error in more detail. One such message says that the current basis has more than one element in row i and column j. This could be caused by a corresponding error in the input parameters `a(*)`, `ha(*)`, and `ka(*)`.

`22. EXIT -- singular basis after nnn factorization attempts`
This exit is highly unlikely to occur. The first factorization attempt will have found the basis to be structurally or numerically singular. (Some diagonals of the triangular matrix PUQ were respectively zero or smaller than a certain tolerance.) The associated variables are replaced by slacks and the modified basis is refactorized. The ensuing singularity must mean that the problem is badly scaled, or the `LU factor tolerance` is too high.

> If the following messages arise, either an OLD BASIS file could not be loaded properly, or some fatal system error has occurred. New BASIS files cannot be saved, and there is no solution to print. The problem is abandoned.

`30. EXIT -- the basis file dimensions do not match this problem`
On the first line of the OLD BASIS file, the dimensions labelled `m` and `n` are different from those associated with the problem that has just been defined. You have probably loaded a file that belongs to some other problem.

Remember, if you have added rows or columns to `a(*)`, `ha(*)` and `ka(*)`, you will have to alter `m` and `n` *and* the map beginning on the third line (a hazardous operation). It may be easier to restart with a PUNCH or DUMP file from the earlier version of the problem.

`31. EXIT -- the basis file state vector does not match this problem`
For some reason, the OLD BASIS file is incompatible with the present problem, or is not consistent within itself. The number of basic entries in the state vector (i.e., the number of

3's in the map) is not the same as m on the first card, or else some of the 2's in the map did not have a corresponding "j x_j" entry following the map.

32. EXIT -- system error. Wrong no. of basic variables: nnn
This exit should never happen. If it does, something is seriously awry in the SNOPT source code. Perhaps the single- and double-precision files have been mixed up.

The following messages arise if additional storage is needed to allow optimization to begin. The problem is abandoned.

42. EXIT -- not enough 8-character storage to start solving the problem
The main character storage array cw(*) is not large enough.

43. EXIT -- not enough integer storage to start solving the problem
The main integer storage array iw(*) is not large enough to provide workspace for the optimization procedure. See the advice given for Exit 20.

44. EXIT -- not enough real storage to start solving the problem
The main real storage array rw(*) is not large enough to provide workspace for the optimization procedure. Be sure that the Superbasics limit and Hessian dimension are not unreasonably large. Otherwise, see the advice given for Exit 43.

8.6. Solution Output

At the end of a run, the final solution will be output to the PRINT file in accordance with the `Solution` keyword. Some header information appears first to identify the problem and the final state of the optimization procedure. A ROWS section and a COLUMNS section then follow, giving one line of information for each row and column. The format used is similar to that seen in commercial systems, though there is no rigid industry standard.

The ROWS section

The general constraints take the form $l \leq Ax \leq u$. The ith constraint is therefore of the form

$$\alpha \leq a^T x \leq \beta.$$

Internally, the constraints take the form $Ax - s = 0$, where s is the set of slack variables (which happen to satisfy the bounds $l \leq s \leq u$). For the ith constraint it is the slack variable s_i that is directly available, and it is sometimes convenient to refer to its state. To reduce clutter, a dot "." is printed for any numerical value that is exactly zero.

`Label` *Description*

`Number` The value $n + i$. This is the internal number used to refer to the ith slack in the iteration log.

`Row` The name of the ith row.

`State` The state of the ith row relative to the bounds α and β. The various states possible are as follows.

- `LL` The row is at its lower limit, α.
- `UL` The row is at its upper limit, β.
- `EQ` The lower and upper limit are the same, $\alpha = \beta$.
- `BS` The constraint is not binding. s_i is basic.
- `SBS` The constraint is not binding. s_i is superbasic.

A key is sometimes printed before the `State` to give some additional information about the state of the slack variable.

- `A` *Alternative optimum possible.* The slack is nonbasic, but its reduced gradient is essentially zero. This means that if the slack were allowed to start moving away from its bound, there would be no change in the value of the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case, the values of dual variables *might* also change.
- `D` *Degenerate.* The slack is basic or superbasic, but it is equal to (or very close to) one of its bounds.
- `I` *Infeasible.* The slack is basic or superbasic and it is currently violating one of its bounds by more than the `Feasibility tolerance`.
- `N` *Not precisely optimal.* The slack is nonbasic or superbasic. If the `Optimality tolerance` were tightened by a factor of 10 (e.g., if it were reduced from 10^{-5} to 10^{-6}), the solution would not be declared optimal because the reduced gradient for the slack would not be considered negligible. (If a loose tolerance

has been used, or if the run was terminated before optimality, this key might be helpful in deciding whether or not to restart the run.)

Note: If `Scale` is specified, the tests for assigning the `A`, `D`, `I`, `N` keys are made on the scaled problem, since the keys are then more likely to be correct.

`Activity` The row value; i.e., the value of $a^T x$.

`Slack activity` The amount by which the row differs from its nearest bound. (For free rows, it is taken to be minus the `Activity`.)

`Lower limit` α, the lower bound on the row.

`Upper limit` β, the upper bound on the row.

`Dual activity` The value of the dual variable π_i, often called the shadow price (or simplex multiplier) for the ith constraint. The full vector π always satisfies $B^T\pi = g_B$, where B is the current basis matrix and g_B contains the associated gradients for the current objective function.

`I` The constraint number, i.

The COLUMNS section

Here we talk about the "column variables" x. For convenience we let the jth component of x be the variable x_j and assume that it satisfies the bounds $\alpha \le x_j \le \beta$. A "." is printed for any numerical value that is exactly zero.

`Label` *Description*

`Number` The column number, j. This is the internal number used to refer to x_j in the iteration log.

`Column` The name of x_j.

`State` The state of x_j relative to the bounds α and β. The various states possible are as follows.

- `LL` x_j is nonbasic at its lower limit, α.
- `UL` x_j is nonbasic at its upper limit, β.
- `EQ` x_j is nonbasic and fixed at the value $\alpha = \beta$.
- `FR` x_j is nonbasic and currently zero, even though it is free to take any value. (Its bounds are $\alpha = -\infty$, $\beta = +\infty$. Such variables are normally basic.)
- `BS` x_j is basic.
- `SBS` x_j is superbasic.

A key is sometimes printed before the `State` to give some additional information about the state of x_j. The possible keys are `A`, `D`, `I` and `N`. They have the same meaning as described above (for the ROWS section of the solution), but the words "the slack" should be replaced by "x_j".

`Activity` The value of the variable x_j.

`Obj Gradient` g_j, the jth component of the linear and quadratic objective function $q(x) + c^T x$. (We define $g_j = 0$ if the current solution is infeasible.)

`Lower limit` α, the lower bound on x_j.

`Upper limit` β, the upper bound on x_j.

`Reduced gradnt` The reduced gradient $d_j = g_j - \pi^T a_j$, where a_j is the jth column of the constraint matrix (or the jth column of the Jacobian at the start of the final major iteration).

`M+J` The value $m + j$.

An example of the printed solution is given in §8. Infinite `Upper` and `Lower limits` are output as the word `None`. Other real values are output with format `f16.5`. The maximum record length is 111 characters, including the first (carriage-control) character.

Note: If two problems are the same except that one minimizes $q(x)$ and the other maximizes $-q(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j will be reversed.

8.7. The SOLUTION file

If a positive SOLUTION file is specified, the information contained in a printed solution may also be output to the relevant file (which may be the PRINT file if so desired). Infinite `Upper` and `Lower limits` appear as $\pm 10^{20}$ rather than `None`. Other real values are output with format `1pe16.6`. Again, the maximum record length is 111 characters, including what would be the carriage-control character if the file were printed.

A SOLUTION file is intended to be read from disk by a self-contained program that extracts and saves certain values as required for possible further computation. Typically the first 14 records would be ignored. Each subsequent record may be read using

```
format(i8, 2x, 2a4, 1x, a1, 1x, a3, 5e16.6, i7)
```

adapted to suit the occasion. The end of the ROWS section is marked by a record that starts with a `1` and is otherwise blank. If this and the next 4 records are skipped, the COLUMNS section can then be read under the same format. (There should be no need to use any `BACKSPACE` statements.)

8.8. The SUMMARY file

If `Summary file` f is specified with $f > 0$, certain brief information will be output to file f. When SNOPT is run interactively, file f will usually be the terminal. For batch jobs a disk file should be used, to retain a concise log of each run if desired. (A SUMMARY file is more easily perused than the associated PRINT file).

A SUMMARY file (like the PRINT file) is not rewound after a problem has been processed. It can therefore accumulate a log for every problem in the SPECS file, if each specifies the same file. The maximum record length is 72 characters, including a carriage-control character in column 1.

The following information is included:

1. The `Begin` card from the SPECS file.
2. The basis file loaded, if any.
3. The status of the solution after each basis factorization (whether feasible; the objective value; the number of function calls so far).
4. The same information every kth iteration, where k is the specified `Summary frequency` (default $k =$ `100`).
5. Warnings and error messages.
6. The exit condition and a summary of the final solution.

Item 4 is preceded by a blank line, but item 5 is not.

All items are illustrated below, where we give the SUMMARY file for the four-variable problem of §3, with the option `Summary frequency = 1`.

```
==============================
S N O P T  5.3       (Oct  97)
==============================

Begin  Toy NLP problem
     Jacobian Dense


Scale option  1,    Partial price   1


Linear constraints satisfied after      0  minor itns.      0  superbasics.

funcon  sets       4   out of       4   constraint gradients.
funobj  sets       3   out of       3   objective  gradients.



Major Minor   Step   nCon    Merit      Feasibl Optimal   nS Penalty PD
    0     3 0.0E+00     1  1.700000E+01 1.7E-01 1.7E+00    0 0.0E+00 FF  R
    1     0 1.0E+00     2  1.400000E+01 0.0E+00 0.0E+00    0 6.0E+00 TT s

EXIT -- optimal solution found

Problem name                 Toy NLP
No. of iterations                  3   Objective value      1.4000000000E+01
No. of major iterations            1   Linear objective     1.0000000000E+01
Penalty parameter           6.010E+00   Nonlinear objective  4.0000000000E+00
No. of calls to funobj             4   No. of calls to funcon             3
No. of degenerate steps            0   Percentage                      0.00
Norm of x   (scaled)          5.0E+00   Norm of pi  (scaled)          9.0E+00
Norm of x                     9.2E+00   Norm of pi                    7.0E+00
Max Prim inf(scaled)        0 0.0E+00   Max Dual inf(scaled)        0 0.0E+00
Max Primal infeas           0 0.0E+00   Max Dual infeas             0 0.0E+00
Nonlinear constraint violn    0.0E+00

 Solution printed on file   9

Time for MPS input                           0.00 seconds
Time for solving problem                     0.00 seconds
Time for solution output                     0.00 seconds
Time for constraint functions                0.00 seconds
Time for objective function                  0.00 seconds

snopt finished.
inform =           0
nInf   =           0
sInf   =  0.0000000000000000E+00
obj    =   4.000000000000000
```

9. BASIS Files

For non-trivial problems, it is advisable to save a BASIS file at the end of a run, in order to restart the run if necessary, or to provide a good starting point for some closely related problem.

Three formats are available for saving basis descriptions. They are invoked by SPECS lines of the following form:

```
NEW BASIS FILE   10
BACKUP   FILE    11      (same as NEW BASIS but on a different file)
PUNCH    FILE    20
DUMP     FILE    30
```

The file numbers may be whatever is convenient, or zero for files that are not wanted.

NEW BASIS and BACKUP files are saved every kth iteration, in that order, where k is the `Save frequency`.

NEW, PUNCH and DUMP files are saved at the end of a run, in that order. They may be re-loaded at the start of a subsequent run by specifying SPECS lines of the following form respectively:

```
OLD BASIS FILE      10
INSERT    FILE      20
LOAD      FILE      30
```

Only one such file will actually be loaded. If more than one positive file number is specified, the order of precedence is as shown. If no BASIS files are specified, one of the `CRASH OPTION`s takes effect.

Figures 1–3 illustrate the data formats used for BASIS files. 80-character fixed-length records are suitable in all cases. (36-character records would be adequate for PUNCH and DUMP files.) The files shown correspond to the optimal solution for the economic-growth model MANNE, described in Section 8.4. Selected column numbers are included to define significant data fields. The problem has 10 nonlinear constraints, 10 linear constraints, and 30 variables.

9.1. NEW and OLD BASIS Files

We sometimes call these files *basis maps*. They contain the most compact representation of the state of each variable. They are intended for restarting the solution of a problem at a point that was reached by an earlier run on the *same problem* or a related problem with the *same dimensions*. (Perhaps the `Iterations limit` was previously too small, or some other objective row is to be used.)

As illustrated in Figure 1, the following information is recorded in a NEW BASIS file.

1. A line containing the problem name, the iteration number when the file was created, the status of the solution (`OPTIMAL SOLN`, `INFEASIBLE`, `UNBOUNDED`, `EXCESS ITNS`, `ERROR CONDN`, or `PROCEEDING`), the number of infeasibilities, and the current objective value (or the sum of infeasibilities).

2. A line containing the `OBJECTIVE`, `RHS`, `RANGES` and `BOUNDS` names, `M` = the number of rows in the constraint matrix, `N` = the number of columns in the constraint matrix, and `SB` = the number of superbasic variables.

3. A set of $(\texttt{N}+\texttt{M}-1)/80+1$ lines indicating the state of the `N` column variables and the `M` slack variables in that order. One character `HS(`j`)` is recorded for each $j = 1, 2, \ldots, \texttt{N}+\texttt{M}$ as follows, written with `FORMAT(80I1)`.

HS(*j*)	*State of the j-th variable*
0	Nonbasic at lower bound
1	Nonbasic at upper bound
2	Superbasic
3	Basic

If variable j is *fixed* (lower bound = upper bound), then HS(j) may be 0 or 1. The same is true if variable j is *free* (infinite bounds) and still nonbasic, although free variables will almost always be basic.

4. A set of lines of the form

$$j \qquad\qquad x_j$$

written with `format(i8, 1pe24.14)` and terminated by an entry with $j = 0$, where j denotes the jth variable and x_j is a real value. The jth variable is either the jth column or the $(j - \mathtt{N})$th slack, if $j > \mathtt{N}$. Typically, HS(j) = 2 (superbasic). When nonlinear constraints are present, this list of superbasic variables is extended to include all basic nonlinear variables. The Jacobian matrix can then be reconstructed exactly for a restart.

Loading a NEW BASIS *file*

A file that has been saved as an OLD BASIS file may be input at the beginning of a later run as a NEW BASIS file. The following notes are relevant:

1. The first line is input and printed but otherwise not used.
2. The values labelled M and N on the second line must agree with those for the problem that has just been defined. The value labelled sb is input and printed but is not used.
3. The next set of lines must contain exactly m values hs(j) = 3, denoting the basic variables.
4. The list of j and x_j values must include an entry for every variable whose state is hs(j) = 2 (the superbasic variables).
5. Further j and x_j values may be included, in any order.
6. For any j in this list, if hs(j) = 3 (basic), the value x_j will be recorded for nonlinear variables, but the variable will remain basic.
7. If hs(j) $\neq$ 3, variable j will be initialized at the value x_j and its state will be reset to 2 (superbasic). If the number of superbasic variables has already reached the `Superbasics limit`, then variable j will be made nonbasic at the bound nearest to x_j (or at zero if it is a free variable).

```
sqdat2..  ITN       0    Optimal Soln  NINF      0      OBJ  -2.043665038075E+06
OBJ=         RHS=          RNG=          BND=          M=      8 N=      7 SB=    1
033023303133003
       5    4.33461578293999E+02
       0
```

Figure 1: Format of NEW and OLD BASIS files

9.2. PUNCH and INSERT Files

These files provide compatibility with commercial mathematical programming systems. The PUNCH file from a previous run may be used as an INSERT file for a later run on the same problem. It may also be possible to modify the INSERT file and/or problem and still obtain a useful advanced basis.

The standard MPS format has been slightly generalized to allow the saving and reloading of nonbasic solutions. It is illustrated in Figure 2. Apart from the first and last line, each entry has the following form:

Columns	2–3	5–12	15–22	25–36
Contents	*Key*	*Name1*	*Name2*	*Value*

The various keys are best defined in terms of the action they cause on input. It is assumed that the basis is initially set to be the full set of slack variables, and that column variables are initially at their smallest bound in absolute magnitude.

Key	*Action to be taken during* INSERT
`XL`	Make variable *Name1* basic and slack *Name2* nonbasic at its lower bound.
`XU`	Make variable *Name1* basic and slack *Name2* nonbasic at its upper bound.
`LL`	Make variable *Name1* nonbasic at its lower bound.
`UL`	Make variable *Name1* nonbasic at its upper bound.
`SB`	Make variable *Name1* superbasic at the specified *Value*.

Note that *Name1* may be a column name or a row name, but (on `XL` and `XU` lines) *Name2* must be a row name. In all cases, row names indicate the associated slack variable, and if *Name1* is a nonlinear variable then its *Value* is recorded for possible use in defining the initial Jacobian matrix.

The key `SB` is an addition to the standard MPS format to allow for nonbasic solutions.

Notes on PUNCH Data

1. Variables are output in natural order. For example, on the first `XL` or `XU` line, *Name1* will be the first basic column and *Name2* will be the first row whose slack is not basic. (The slack could be nonbasic or superbasic.)
2. `LL` lines are *not* output for nonbasic variables if the corresponding lower bound value is zero.
3. Superbasic slacks are output last.
4. PUNCH and INSERT files deal with the status and values of *slack variables*. This is in contrast to the printed solution and the SOLUTION file, which deal with *rows*.

Notes on INSERT Data

1. Before an INSERT file is read, column variables are made nonbasic at their smallest bound in absolute magnitude, and the slack variables are made basic.
2. Preferably an INSERT file should be an unmodified PUNCH file from an earlier run on the same problem. If some rows have been added to the problem, the INSERT file need not be altered. (The slacks for the new rows will be in the basis.)
3. Entries will be ignored if *Name1* is already basic or superbasic. `XL` and `XU` lines will be ignored if *Name2* is not basic.

4. SB lines may be added before the ENDATA line, to specify additional superbasic columns or slacks.

5. An SB line will not alter the status of *Name1* if the SUPERBASICS LIMIT has been reached. However, the associated *Value* will be retained if *Name1* is a Jacobian variable.

9.3. DUMP and LOAD Files

These files are similar to PUNCH and INSERT files, but they record solution information in a manner that is more direct and more easily modified. In particular, no distinction is made between columns and slacks. Apart from the first and last line, each entry has the form

Columns	2–3	5–12	25–36
Contents	*Key*	*Name*	*Value*

as illustrated in Figure 3. The keys LL, UL, BS and SB mean Lower Limit, Upper Limit, Basic and Superbasic respectively.

Notes on DUMP Data

1. A line is output for every variable, columns followed by slacks.

2. Nonbasic free variables will be output with either LL or UL keys and with *Value* zero.

Notes on LOAD Data

1. Before a LOAD file is read, all columns and slacks are made nonbasic at their smallest bound in absolute magnitude. The basis is initially empty.

2. Each LL, UL or BS line causes *Name* to adopt the specified status. The associated *Value* will be retained if *Name* is a Jacobian variable.

3. An SB line causes *Name* to become superbasic at the specified *Value*.

4. An entry will be ignored if *Name* is already basic or superbasic. (Thus, only the first BS or SB line takes effect for any given *Name*.)

5. An SB line will not alter the status of *Name* if the Superbasics limit has been reached, but the associated *Value* will be retained if *Name* is a Jacobian variable.

6. (*Partial basis*) Let M be the number of rows in the problem. If fewer than M variables are specified to be basic, a tentative basis list will be constructed by adding the requisite number of slacks, starting from the first row and taking those that were not previously specified to be basic or superbasic. (If the resulting basis proves to be singular, the basis factorization routine will replace a number of basic variables by other slacks.) The starting point obtained in this way will not necessarily be "good".

7. (*Too many basics*) If M variables have already been specified as basic, any further BS keys will be treated as though they were SB. This feature may be useful for combining solutions to smaller problems.

```
NAME          sqdat2..  PUNCH/INSERT
 XL ...100    ....01     3.89064E+02
 XU ...101    ....02     6.19233E+02
 LL ...102    ....03     1.00000E+02
 SB ...105    ....04     4.33462E+02
 XL ...107    ....05     3.00048E+02
 XL ...111    ....06     1.58194E+02
ENDATA
```

Figure 2: Format of PUNCH/INSERT files

```
NAME          sqdat2..     DUMP/LOAD
 LL ...100                 0.00000E+00
 BS ...101                 3.89064E+02
 BS ...102                 6.19233E+02
 LL ...103                 1.00000E+02
 SB ...104                 4.33462E+02
 BS ...105                 3.00048E+02
 BS ...106                 1.58194E+02
 LL ...107                 2.00000E+03
 BS ...108                 4.83627E+01
 UL ...109                 1.00000E+02
 BS ...110                 3.24241E+01
 BS ...111                 1.60065E+01
 LL ...112                 1.50000E+03
 LL ...113                 2.50000E+02
 BS ...114                -2.90022E+06
ENDATA
```

Figure 3: Format of DUMP/LOAD files

9.4. Restarting Modified Problems

Sections 9.1–9.3 document three distinct starting methods (OLD BASIS, INSERT and LOAD files), which may be preferable to any of the cold start (CRASH) options. The best choice depends on the extent to which a problem has been modified, and whether it is more convenient to specify variables by number or by name. The following notes offer some rules of thumb.

Protection

In general there is no danger of specifying infinite values. For example, if a variable is specified to be nonbasic at an upper bound that happens to be $+\infty$, it will be made nonbasic at its lower bound. Conversely if its lower bound is $-\infty$. If the variable is *free* (both bounds infinite), it will be made nonbasic at value zero. No warning message will be issued.

Default Status

If the status of a variable is not explicitly given, it will initially be nonbasic at the bound that is smallest in absolute magnitude. Ties are broken in favor of lower bounds, and free variables will again take the value zero.

Restarting with Different Bounds

Suppose that a problem is to be restarted after the bounds on some variable `X` have been altered. Any of the basis files may be used, but the starting point obtained depends on the status of `X` at the time the basis is saved.

If `X` is basic or superbasic, the starting point will be the same as before (all other things being equal). The value of `X` may lie outside its new set of bounds, but there will be minimal loss of feasibility or optimality for the problem as a whole.

If `X` was previously *fixed*, it is likely to be nonbasic at its *lower* bound (which happens to be the same as its upper bound). Increasing its upper bound will not affect the solution.

In contrast, if `X` is nonbasic at its *upper* bound and if that bound is altered, the starting values for an arbitrary number of basic variables could be changed (since they will be recomputed from the nonbasic and superbasic variables). This may not be of great consequence,

but sometimes it may be worthwhile to retain the old solution precisely. To do this, one must make `X` superbasic at the original bound value.

For example, if `x` is nonbasic at an upper bound of 5.0 (which has now been changed), one should insert a line of the form

```
j                    5.0
```

near the end of an OLD BASIS file, or the line

```
SB X                 5.0
```

near the end of an INSERT or LOAD file. Note that the SPECS file must specify a `Superbasics limit` at least as large as the number of variables involved, even for purely linear problems.

Sequences of Problems

Whenever practical, a series of related problems should be ordered so that the *most tightly constrained* cases are solved first. Their solutions will often provide feasible starting points for subsequent relaxed problems, as long the above precautions are taken.