

# 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

February 1999

## Abstract

SNOPT is a general-purpose system for solving optimization problems involving many variables and constraints. It minimizes a linear or nonlinear function subject to bounds on the variables and sparse linear or nonlinear constraints. It is suitable for large-scale linear and quadratic programming and for linearly constrained optimization, as well as for general nonlinear programs.

SNOPT finds solutions that are *locally optimal*, and ideally any nonlinear functions should be smooth and users should provide gradients. It is 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. Unknown gradients are estimated by finite differences.

SNOPT uses a sequential quadratic programming (SQP) algorithm 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 (and their gradients) are expensive to evaluate.

The source code for SNOPT is suitable for any machine with a Fortran compiler. SNOPT may be called from a driver program (typically in Fortran, C or MATLAB). SNOPT can also be used as a stand-alone package, reading data in the MPS format used by commercial mathematical programming systems.

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

pgill@ucsd.edu

<http://sdna3.ucsd.edu/~peg>

walter@sol-walter.stanford.edu

<http://www.stanford.edu/~walter/>

mike@sol-michael.stanford.edu

<http://www.stanford.edu/~saunders/>

## Contents

<b>1. Introduction</b>	<b>3</b>
1.1 Problem types . . . . .	3
1.2 Subroutines . . . . .	4
1.3 Files . . . . .	4
<b>2. Description of the method</b>	<b>5</b>
2.1 Constraints and slack variables . . . . .	5
2.2 Major iterations . . . . .	5
2.3 Minor iterations . . . . .	6
2.4 The merit function . . . . .	6
2.5 Treatment of constraint infeasibilities . . . . .	7
<b>3. Identifying structure in the objective and constraints</b>	<b>8</b>
3.1 Problem dimensions . . . . .	9
<b>4. Subroutine snopt</b>	<b>10</b>
<b>5. User-supplied subroutines</b>	<b>16</b>
5.1 Subroutine funobj . . . . .	17
5.2 Subroutine funcon . . . . .	19
5.3 Constant Jacobian elements . . . . .	20
5.4 Example . . . . .	20
<b>6. The SPECS file</b>	<b>23</b>
6.1 SPECS file checklist and defaults . . . . .	23
6.2 Subroutine snInit . . . . .	26
6.3 Subroutine snSpec . . . . .	27
6.4 Subroutines snset, snseti, snsetr . . . . .	28
6.5 Subroutines sngetc, sngeti, sngetr . . . . .	29
6.6 Description of the optional parameters . . . . .	30
<b>7. Output</b>	<b>46</b>
7.1 The PRINT file . . . . .	46
7.2 The major iteration log . . . . .	46
7.3 The minor iteration log . . . . .	49
7.4 Basis factorization statistics . . . . .	50
7.5 Crash statistics . . . . .	52
7.6 EXIT conditions . . . . .	53
7.7 Solution output . . . . .	57
7.8 The SOLUTION file . . . . .	60
7.9 The SUMMARY file . . . . .	60
<b>8. BASIS files</b>	<b>62</b>
8.1 NEW and OLD BASIS files . . . . .	62
8.2 PUNCH and INSERT files . . . . .	64
8.3 DUMP and LOAD files . . . . .	65





## 1.2. Subroutines

SNOPT is accessed via the following routines:

**snInit** (§6.2) Must be called before any other SNOPT routines.

**snSpec** (§6.3) May be called to input a SPECS file (a list of run-time options).

**snset**, **snseti**, **snsetr** (§6.4) May be called to specify a single option.

**sngetc**, **sngeti**, **sngetr** (§6.5) May be called to obtain an option's current value.

**funcon**, **funobj** (§5) Supplied by the user and called by **snopt**. They define the constraint functions  $F(x)$  and objective function  $f(x)$ .

**snopt** (§4) The main solver.

**snMem** (In distribution file **sn12snzz.f**) Computes the size of the workspace arrays **cw**, **iw**, **rw** required for given problem dimensions. Intended for Fortran 90 drivers that reallocate workspace if necessary.

The user routines **funcon** and **funobj** have a fixed parameter list but may have any convenient name. They are passed to **snopt** as parameters.

The SNOPT routines are intended to be re-entrant (as long as the Fortran compiler allocates local variables dynamically). Hence they may be used in a parallel or multi-thread environment. They may also be called recursively.

## 1.3. Files

SNOPT reads or creates the following files:

**SPECS file.** A list of run-time options, input by **snSpec**.

**PRINT file.** A detailed iteration log, error messages, and optionally the printed solution.

**SUMMARY file.** A brief iteration log, error messages, and the final solution status. Intended for screen output in an interactive environment.

**SOLUTION file.** A separate copy of the printed solution.

**BASIS files.** To allow restarts.

You must define unit numbers for the **specs**, **print** and **summary** files by specifying appropriate parameters for **snInit** and **snSpec**. For a more detailed description of the files that can be created by SNOPT, see §8.

## 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 SQP algorithm is fully described in [6].

### 2.1. Constraints and slack variables

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  $s = (s_1, s_2, \dots, s_m)^T$ . 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

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

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.

### 2.2. Major iterations

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*, whose rows are the first derivatives of  $F(x)$  evaluated at  $x_k$ . The QP constraints therefore comprise the  $m$  linear constraints

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

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

$$\underset{x, s}{\text{minimize}} \quad q(x) \quad \text{subject to} \quad Ax - s = b, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u, \quad (2.1)$$

where  $q(x)$  is a quadratic approximation to a modified Lagrangian function [6]. The linear constraint matrix  $A$  above is input to SNOPT by means of the arguments  $\mathbf{a}(\ast)$ ,  $\mathbf{ha}(\ast)$  and  $\mathbf{ka}(\ast)$  (see §4). Access to this matrix allows the user to specify the pattern of nonzero elements in  $F'(x)$  and  $G$ , and to identify any nonzero elements that remain constant throughout the minimization.

### 2.3. Minor iterations

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 upper 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 the sum of infeasibilities). The basic variables are then adjusted in order to ensure 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*  $\pi$ . 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  $\pi$ . 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 other variables, including superbasics. In practice, an *approximate* QP solution is found by relaxing these conditions on  $d_j$  (see the **Minor optimality tolerance** described in §6.6).

### 2.4. The merit function

After a QP subproblem has been solved, new estimates of the NP solution are computed using a linesearch on the augmented Lagrangian merit function

$$\mathcal{M}(x, s, \pi) = f(x) - \pi^T(F(x) - s_N) + \frac{1}{2}(F(x) - s_N)^T D (F(x) - s_N), \quad (2.2)$$

where  $D$  is a diagonal matrix of penalty parameters. If  $(x_k, s_k, \pi_k)$  denotes the current solution estimate and  $(\hat{x}_k, \hat{s}_k, \hat{\pi}_k)$  denotes the optimal QP solution, the linesearch determines a step  $\alpha_k$  ( $0 < \alpha_k \leq 1$ ) such that the new point

$$\begin{pmatrix} x_{k+1} \\ s_{k+1} \\ \pi_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ s_k \\ \pi_k \end{pmatrix} + \alpha_k \begin{pmatrix} \hat{x}_k - x_k \\ \hat{s}_k - s_k \\ \hat{\pi}_k - \pi_k \end{pmatrix} \quad (2.3)$$

gives a *sufficient decrease* in the merit function (2.2). When necessary, the penalties in  $D$  are increased by the minimum-norm perturbation that ensures descent for  $\mathcal{M}$  [10]. 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 [7, 3].

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

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 the functions can be safely evaluated.) SNOPT proceeds to solve NP as given, using search directions obtained from a sequence of quadratic programming subproblems (2.1).

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

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

where  $\gamma$  is a nonnegative parameter (the *elastic weight*), and  $f(x) + \gamma e^T(v+w)$  is called a *composite objective*. If  $\gamma$  is sufficiently large, this is equivalent to minimizing the sum of the nonlinear constraint violations subject to the linear constraints and bounds. A similar  $\ell_1$  formulation of NP is fundamental to the  $S\ell_1$ QP algorithm of Fletcher [4]. See also Conn [1].

### 3. Identifying structure in the objective and constraints

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

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

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  $x' = (u, v, z)$  and a *linear* function of (potentially) all variables  $x$ .
- The first two constraints are nonlinear, and the third constraint is linear.
- Each nonlinear constraint involves the sum of a *nonlinear* function of the two variables  $x'' = (u, v)$  and a *linear* function of the remaining variables  $y'' = (z, w)$ .

The nonlinear terms are defined by user-written subroutines `funobj` and `funcon`, which involve only  $x'$  and  $x''$ , the appropriate subsets of variables.

For the objective, we define the function  $f(u, v, z) = (u + v + z)^2$  to include only the nonlinear terms. The variables  $x' = (u, v, z)$  are known as *nonlinear objective variables*, and their dimension is specified by the `snopt` input parameter `nnObj` (= 3 here). The linear part  $3z + 5w$  of the objective is treated as an additional linear constraint whose row index is specified by the input parameter `iObj` (= 3 or 4 here). Thus, the full objective has the form

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

where  $x'$  is the first `nnObj` variables,  $f(x')$  is defined by subroutine `funobj`, and  $c$  is a constant vector that forms row `iObj` of the full Jacobian matrix  $A$ .

Similarly for the constraints, we define a vector function  $F(u, v)$  to include just the nonlinear terms. In this example,  $F_1(u, v) = u^2 + v^2$  and  $F_2(u, v) = u^4 + v^4$ . The number of nonlinear constraints (the dimension of  $F$ ) is specified by the input parameter `nnCon` (= 2 here). The variables  $x'' = (u, v)$  are known as *nonlinear Jacobian variables*, with dimension specified by `nnJac` (= 2 here). Thus, the constraint functions and the linear part of the objective have the form

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

where  $x''$  is the first `nnJac` variables,  $F(x'')$  is defined by subroutine `funcon`, and  $y''$  contains the remaining variables. The full Jacobian is of the form

$$A = \begin{pmatrix} J(x'') & A_2 \\ A_3 & A_4 \end{pmatrix}, \quad (3.2)$$

with the Jacobian of  $F$  always appearing in the *top left corner* of  $A$ . The constant matrices  $A_2, A_3, A_4$  and the sparsity pattern of  $J(x'')$  are input column-wise via the array parameters `a`, `ha`, `ka`. (Elements that are identically zero need not be included.)

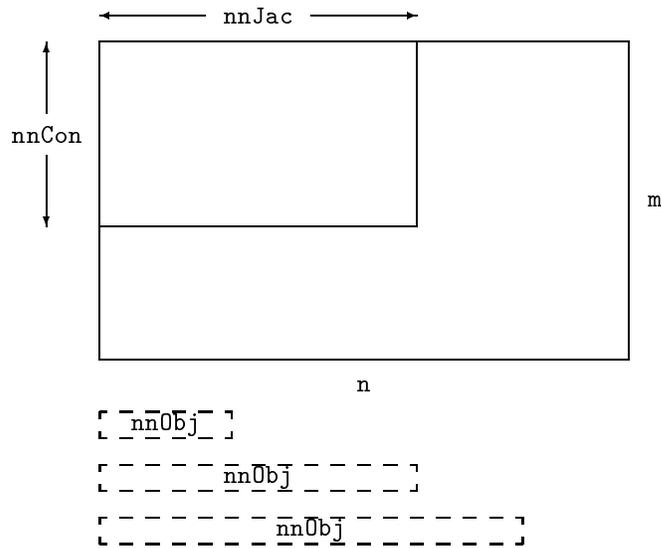
The inequalities  $l_1 \leq F(x'') + A_2 y'' \leq u_1$  and  $l_2 \leq A_3 x'' + A_4 y'' \leq u_2$  implied by the constraint functions (3.1) are known as the *nonlinear* and *linear* constraints respectively. Together, these two sets of inequalities constitute the *general constraints*.

In general, the vectors  $x'$  and  $x''$  have different dimensions, but they *always overlap*, in the sense that the shorter vector is always the beginning of the other. In the example, the nonlinear Jacobian variables  $(u, v)$  are an ordered subset of the nonlinear objective variables  $(u, v, w)$ . In other cases it could be the other way round—whichever is the most convenient—but the first way keeps  $J(x'')$  smaller.

Together the nonlinear objective and nonlinear Jacobian variables comprise the *nonlinear variables*. The number of nonlinear variables is therefore the *larger* of the dimensions of  $x'$  and  $x''$ .

### 3.1. Problem dimensions

The following picture illustrates the problem structure just described:



The dimensions are all input parameters to subroutine `snopt` (see the next section). For linear programs, `nnCon`, `nnJac` and `nnObj` are all zero. If a linear objective term exists, `iObj` points to one of the bottom rows ( $\text{nnCon} < \text{iObj} \leq m$ ).

The dashed boxes indicate that a nonlinear objective function  $f(x')$  may involve either a subset or a superset of the variables in the nonlinear constraint functions  $F(x'')$ , counting from the left. Thus,  $\text{nnObj} \leq \text{nnJac}$  or vice versa.

Sometimes the objective and constraints really involve *disjoint sets of nonlinear variables*. We then recommend ordering the variables so that  $\text{nnObj} > \text{nnJac}$  and  $x' = (x'', x''')$ , where the objective is nonlinear in just the last vector  $x'''$ . Subroutine `funobj` should set  $g(j) = 0.0$  for  $j = 1 : \text{nnJac}$ . It should then set as many remaining gradients as possible—preferably all!

#### 4. 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 constraint matrix  $A$  in (3.2).

*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**, **ha** and **ka** below.

- n** is  $n$ , 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 ( $nnCon \geq 0$ ).
- nnObj** is  $n'_1$ , the number of nonlinear objective variables ( $nnObj \geq 0$ ).
- 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 output BASIS files. A blank name may be used.
- funcon** is the name of a subroutine that calculates the vector of nonlinear constraint functions  $F(x)$  and (optionally) its Jacobian for a specified vector  $x$  (the first **nnJac** elements of **x(\*)**). **funcon** must be declared **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  $x$  (the first **nnObj** elements of **x(\*)**). **funobj** must be declared **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  $A$  (3.2), including the Jacobian matrix associated with nonlinear constraints. The nonzeros are stored column-wise. A pair of values (**a(k)**,**ha(k)**) contains a matrix element and its corresponding row index, and the array **ka(\*)** is a set of pointers to the beginning of each column of  $A$  within **a(\*)** and **ha(\*)**. Thus for  $j = 1 : n$ , the entries of column  $j$  are held in **a(k : l)** and their corresponding row indices are in **ha(k : l)**, where  $k = ka(j)$  and  $l = ka(j + 1) - 1$ ,

*Note: Every element of **a(\*)** must be assigned a value in the calling program.*

In general, elements in the nonlinear part of **a(\*)** (see the notes below) may be any dummy value (e.g., zero) because they are initialized at the first point that is feasible with respect to the linear constraints.

If **Derivative level** = 2 or 3, the nonlinear part of **a(\*)** may be used to define any constant Jacobian elements. If **funcon** does not define all entries of **gCon(\*)**, the missing values will be obtained from **a(\*)**.

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

2. The Jacobian  $J(x)$  forms the top left corner of  $\mathbf{a}$  and  $\mathbf{ha}$  (see §3). If a Jacobian column  $j$  ( $1 \leq j \leq \mathbf{nnJac}$ ) contains any entries  $\mathbf{a}(k)$ ,  $\mathbf{ha}(k)$  associated with nonlinear constraints ( $1 \leq \mathbf{ha}(k) \leq \mathbf{nnCon}$ ), those entries must come before any entries belonging to linear constraints.
3. The row indices  $\mathbf{ha}(k)$  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.
4. If your problem has no constraints, or just bounds on the variables, you may include a dummy “free” row with a single (zero) element by setting  $\mathbf{a}(1) = 0.0$ ,  $\mathbf{ha}(1) = 1$ ,  $\mathbf{ka}(1) = 1$ , and  $\mathbf{ka}(j) = 2$  for  $j = 2 : n + 1$ . This row is made “free” by setting its bounds to be  $\mathbf{bl}(n + 1) = -\mathbf{bigbnd}$  and  $\mathbf{bu}(n + 1) = \mathbf{bigbnd}$ , where  $\mathbf{bigbnd}$  is typically  $1.0\text{e}+20$  (see next paragraph).

$\mathbf{bl}(n+m)$  contains the lower bounds on the variables and slacks  $(x, s)$ .

The first  $n$  entries of  $\mathbf{bl}$ ,  $\mathbf{bu}$ ,  $\mathbf{hs}$  and  $\mathbf{xs}$  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  $\mathbf{bl}(j) \leq -\mathbf{bigbnd}$ , where  $\mathbf{bigbnd}$  is the Infinite Bound, whose default value is  $10^{20}$ .

To fix the  $j$ th variable (say  $x_j = \beta$ , where  $|\beta| < \mathbf{bigbnd}$ ), set  $\mathbf{bl}(j) = \mathbf{bu}(j) = \beta$ .

To make the  $i$ th constraint an *equality* constraint (say  $s_i = \beta$ , where  $|\beta| < \mathbf{bigbnd}$ ), set  $\mathbf{bl}(n + i) = \mathbf{bu}(n + i) = \beta$ .

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

$\mathbf{Names}(nName)$  sometimes contains 8-character names for the variables and constraints. If  $nName = 1$ ,  $\mathbf{Names}$  is not used. The printed solution will use generic names for the columns and row. If  $nName = n + m$ ,  $\mathbf{Names}(j)$  should contain the 8-character name of the  $j$ th variable ( $j = 1 : n + m$ ). If  $j = n + i$ , the  $j$ th variable is the  $i$ th row.

$\mathbf{hs}(n+m)$  sometimes contains a set of initial states for each variable  $x$ , or for each variable and slack  $(x, s)$ . See the following discussion of  $\mathbf{xs}$ .

$\mathbf{xs}(n+m)$  sometimes contains a set of initial values for  $x$  or  $(x, s)$ .

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  $\mathbf{hs}$  and  $\mathbf{xs}$  need not be set at all.
2. Otherwise,  $\mathbf{hs}(1 : n)$  and  $\mathbf{xs}(1 : n)$  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  $\mathbf{hs}(j) = 0$ ,  $\mathbf{xs}(j) = 0.0$  for all  $j = 1 : n$ . All variables will be eligible for the initial basis.

Less trivially, to say that the optimal value of variable  $j$  will probably be equal to one of its bounds, set  $\mathbf{hs}(j) = 4$  and  $\mathbf{xs}(j) = \mathbf{bl}(j)$  or  $\mathbf{hs}(j) = 5$  and  $\mathbf{xs}(j) = \mathbf{bu}(j)$  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  $\mathbf{hs}(j) = 0, 1, 2, 3, 4, 5$  have the following meaning:

$hs(j)$	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  $hs(j) = 2$  are made superbasic. Other entries not selected for the basis are made nonbasic at the value  $xs(j)$  if  $bl(j) \leq xs(j) \leq bu(j)$ , or at the value  $bl(j)$  or  $bu(j)$  closest to  $xs(j)$ . See the description of  $hs$  below (on exit).

4. For Warm starts, all of  $hs(1 : n + m)$  must be 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  $\lambda$ , the vector of Lagrange multipliers (shadow prices) for the nonlinear constraints. The first  $nnCon$  components must be defined. If nothing is known about  $\lambda$ , set  $pi(i) = 0.0$ ,  $i = 1 : nnCon$ .

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

**cu(len<sub>cu</sub>)**, **iu(len<sub>iu</sub>)**, **ru(len<sub>ru</sub>)** are 8-character, integer and real arrays of user workspace. They may be used to pass data or workspace to your function routines **funcon** and **funobj** (which have the same parameters). They are not touched by **snopt**.

If the function routines don't reference these parameters, you may use any arrays of the appropriate type, such as **cw**, **iw**, **rw** (see next paragraph). Alternatively, you should use the latter arrays if **funcon** and **funobj** need to access **snopt**'s workspace.

**cw(len<sub>cw</sub>)**, **iw(len<sub>iw</sub>)**, **rw(len<sub>rw</sub>)** are 8-character, integer and real arrays of workspace for **snopt**.

**len<sub>cw</sub>**, **len<sub>iw</sub>**, **len<sub>rw</sub>** must all be at least 500. In general, **len<sub>cw</sub> = 500** is appropriate but **len<sub>iw</sub>** and **len<sub>rw</sub>** should be as large as possible because it is uncertain how much storage will be needed for the basis factors. As an estimate, **len<sub>iw</sub>** should be about  $100(m + n)$  or larger, and **len<sub>rw</sub>** should be about  $200(m + n)$  or larger.

Appropriate values may be obtained from a preliminary run with **len<sub>cw</sub> = len<sub>iw</sub> = len<sub>rw</sub> = 500**. If **Print level** is positive, the required amounts of workspace are printed before **snopt** terminates with **inform = 42, 43** or **44**. The values are returned in **min<sub>cw</sub>**, **min<sub>iw</sub>** and **min<sub>rw</sub>**.

#### On exit:

**hs(n+m)** is the final state vector. The elements of **hs** have the following meaning:

$hs(j)$	State of variable $j$	Usual value of $xs(j)$
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 **Minor 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 **Minor feasibility tolerance**, and there may be some nonbasics for which  $\mathbf{xs}(j)$  lies strictly between its bounds.

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

$\mathbf{xs}(\mathbf{n+m})$  is the final variables and slacks  $(x, s)$ .

$\mathbf{pi}(\mathbf{m})$  is the vector of dual variables  $\pi$  (a set of Lagrange multipliers for the general constraints).

$\mathbf{rc}(\mathbf{n+m})$  is a vector of reduced costs,  $g - (A - I)^T \pi$ , where  $g$  is the gradient of the objective if  $\mathbf{xs}$  is feasible (or the gradient of the Phase-1 objective otherwise). The last  $m$  entries are  $\pi$ .

**inform** reports the result of the call to **snopt**. Here is a summary of possible values (for a detailed description, see §7.6):

- 0 Optimal solution found, i.e., the primal and dual infeasibilities are negligible.
- 1 The problem is infeasible.
- 2 The problem is unbounded (or badly scaled).
- 3 Too many iterations.
- 4 Feasible solution, but the requested accuracy in the dual infeasibilities could not be achieved.
- 5 The **Superbasics limit** is too small.
- 6 User requested termination by returning  $\mathbf{mode} \leq -2$  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 insufficient 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 superbasic variables.

`nInf`, `sInf` give the number and the sum of the infeasibilities of constraints that lie outside their bounds by more than the **Feasibility tolerance**.

If the *linear* constraints are infeasible, `xs` minimizes the sum of the infeasibilities of the linear constraints subject to the upper and lower bounds being satisfied. In this case `nInf` gives the number of components of  $Gx$  lying outside their upper or lower bounds. The nonlinear constraints are not evaluated.

Otherwise, `xs` minimizes the sum of the infeasibilities of the *nonlinear* constraints subject to the linear constraints and upper and lower bounds being satisfied. In this case `nInf` gives the number of components of  $F(x)$  lying outside their upper or lower bounds.

`Obj` is the value of the objective function, including the constant `ObjAdd`. If `nInf = 0`, `Obj` includes both the linear and nonlinear objective if any. If `nInf > 0`, `Obj` is just the linear objective if any.

## 5. User-supplied subroutines

The user must provide subroutines to define the nonlinear parts of the objective function and nonlinear constraints. They are passed to `snopt` as external parameters `funobj` and `funcon`. (A dummy subroutine must be provided if the objective or constraints are purely linear.)

Be careful when coding the call to `snopt`: the parameters are ordered alphabetically as `funcon`, `funobj`. The first call to each function routine is also in that order.

In general, these subroutines should return all function and gradient values on every entry except perhaps the last. 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 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 reliability, compute all function and gradient values.
2. If the gradients are expensive to compute, specify `Nonderivative linesearch` and use the input parameter `mode` to avoid computing them on certain entries. (Don't compute gradients if `mode = 0`.)
3. If not all gradients are known, you must specify `Derivative level < 3`. You should still compute as many gradients as you can. (It often happens that some of them are constant or even zero.)
4. Again, if the known gradients are expensive, don't compute them if `mode = 0`.
5. Use the input parameter `nState` to test for special actions on the first or last entries.
6. While the function routines are being developed, use the `Verify` option to check the computation of gradient elements that are supposedly known. The `Start` and `Stop` options may also be helpful.
7. The function routines are not called until the linear constraints and bounds on  $x$  are satisfied. This helps confine  $x$  to regions where the nonlinear functions are likely to be defined. However, be aware of the `Minor feasibility tolerance` if the functions have singularities,
8. Set `mode = -1` if the functions are undefined. The linesearch will shorten the step and try again.
9. Set `mode ≤ -2` if you want SNOPT to stop.

### 5.1. Subroutine funobj

This subroutine must calculate the nonlinear objective function  $f(x)$  and (optionally) its gradient  $g(x)$ , where  $x$  is the current value of the objective variables  $x'$ . The  $j$ th component of the gradient is  $\partial f / \partial x_j$ .

---

```

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

integer          mode, nnObj, nState
double precision fObj
double precision x(nnObj), gObj(nnObj)

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

```

#### On entry:

**mode** indicates whether **fObj** or **gObj** or both must be assigned during the present call of **funobj** ( $0 \leq \text{mode} \leq 2$ ).

This parameter can be ignored if **Derivative linesearch** is selected (the default) and if **Derivative level** = 1 or 3. In this case, **mode** will always have the value 2, and **fObj** and all elements of **gObj** must be assigned.

Otherwise, **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 the known components of **gObj**; **fObj** is not required and is ignored.
- If **mode** = 0, only **fObj** need be assigned; **gObj** is ignored.

**nnObj** is the number of variables involved in  $f(x)$  ( $0 < \text{nnObj} \leq n$ ). These must be the first **nnObj** variables in the problem.

**x(nnObj)** contains the nonlinear objective variables  $x$ . *The array **x** must not be altered.*

**nState** indicates the first and last calls to **funobj**.

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. Note that if there are nonlinear constraints, the first call to **funcon** will occur *before* the first call to **funobj**.

If **nState**  $\geq 2$ , **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 + **inform**, where **inform** 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 linesearch 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 the character, integer and real arrays of user workspace provided to `snopt`. They may be used to pass information into the function routines and to preserve data between calls.

In special applications the functions may depend on some of the internal variables stored in `snopt`'s workspace arrays `cw`, `iw`, `rw`. For example, the 8-character problem name `Prob` is stored in `cw(51)`, and the dual variables are stored in `rw(1xMul)` onward, where `1xMul = iw(316)`. These will be accessible to both `funobj` and `funcon` if `snopt` is called with parameters `cu`, `iu`, `ru` the *same* as `cw`, `iw`, `rw`.

If you still require user workspace, elements

```
rw(501:maxru) and rw(maxrw+1:lenru)
```

are set aside for this purpose, where `maxru = iw(2)`. Similarly for workspace in `cw` and `rw`. (See the `Total` and `User` workspace options.)

#### On exit:

`mode` may be used to indicate that you are unable or unwilling to evaluate the objective function at the current  $x$ . (Similarly for the constraint functions.)

During the linesearch, the functions are evaluated at points of the form  $x = x_k + \alpha p_k$  after they have already been evaluated satisfactorily at  $x_k$ . At any such  $\alpha$ , if you set `mode` to  $-1$ , SNOPT will evaluate the functions at some point closer to  $x_k$  (where they are 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  $f(x)$  (except perhaps if `mode = 1`).

`gObj(nnObj)` must contain the known 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`).

---

## 5.2. Subroutine funcon

This subroutine must compute the nonlinear constraint functions  $F(x)$  and (optionally) their gradients  $J(x)$ , where  $x$  is the current value of the Jacobian variables  $x''$ . The  $j$ th column of the Jacobian matrix  $J(x)$  is the vector  $\partial F/\partial x_j$ .

Gradients are stored column-wise in the output array `gCon`.

Recall that  $J(x)$  is the *top left corner* of a larger matrix  $A$  that is stored column-wise in `snopt`'s input arrays `a`, `ha`, `ka` (see (3.2) and §§3.1,4). Jacobian elements must be stored in `gCon` in the same order as the corresponding parts of `a`, `ha`, `ka`.

For small problems (or large dense ones) it is convenient to treat the Jacobian as a dense matrix and declare `gCon` as a two-dimensional array `gCon(*,*)` (which is stored column-wise in Fortran). It is then simple to compute the Jacobian by rows or by columns. For problems with sparse Jacobians, it is essential to use a one-dimensional array `gCon(*)` in order to conserve storage. Thus, `funcon` should use just *one* of the declarations

```
double precision  gCon(nnCon,nnJac)
double precision  gCon(neJac)
```

according to convenience.

---

```

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

integer          mode, nnCon, nnJac, neJac, nState
double precision x(nnJac), fCon(nnCon)

*** Choose ONE of the following:
*   double precision  gCon(nnCon,nnJac)
*   double precision  gCon(neJac)

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

### On entry:

`mode` indicates whether `fCon` or `gCon` or both must be assigned during the present call of `funcon` ( $0 \leq \text{mode} \leq 2$ ).

This parameter can be ignored if `Derivative linesearch` is selected (the default) and if `Derivative level = 2` or `3`. In this case, `mode` will always have the value `2`, and all elements of `fCon` and `gCon` must be assigned (except perhaps constant elements of `gCon`).

Otherwise, `snopt` will call `funcon` with `mode = 0, 1` or `2`. You may test `mode` to decide what to do:

- If `mode = 2`, assign `fCon` and the known components of `gCon`.
- If `mode = 1`, assign the known components of `gCon`; `fCon` is not required and is ignored.
- If `mode = 0`, only `fCon` need be assigned; `gCon` is ignored.

**nnCon** is the number of nonlinear constraints ( $\text{nnCon} > 0$ ). These must be the first **nnCon** constraints in the problem.

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

**neJac** is the value  $\text{nnCon} * \text{nnJac}$ .

**x(nnJac)** contains the nonlinear Jacobian variables  $x$ . *The array **x** must not be altered.*

**nState** is used as in **funobj**.

**cu(len<sub>cu</sub>)**, **iu(len<sub>iu</sub>)**, **ru(len<sub>ru</sub>)** are the same as in **funobj**.

#### On exit:

**fCon(nnCon)** contains the computed constraint vector  $F(x)$  (except perhaps if **mode** = 1).

**gCon(nnCon,nnJac)** or **gCon(neJac)** contains the computed Jacobian  $J(x)$  (except perhaps if **mode** = 0).

These gradient elements must be stored in **gCon** in exactly the same positions as implied by the definitions of **snopt**'s arrays **a**, **ha**, **ka**. There is no internal check for consistency (except indirectly via the **Verify** option), so great care is essential.

**mode** may be set as in **funobj**.

### 5.3. Constant Jacobian elements

If all constraint gradients (Jacobian elements) are known (**Derivative level** = 2 or 3), any *constant* elements may be given to **snopt** in the array **a** if desired. The Jacobian array **gCon** are initialized from the appropriate elements of **a**. If any are constant and have the correct value, **funcon** need not reassign them in **gCon**.

Note that constant *nonzero* elements do affect **fCon**. Thus, if  $J_{ij}$  is assigned correctly in **a(\*)** and is constant, a linear term  $\text{gCon}(i,j) * \mathbf{x}(j)$  or  $\text{gCon}(l) * \mathbf{x}(j)$  must be added to **fCon(i)** (depending on whether **gCon** is a two- or one-dimensional array).

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 example of §3, repeated here for convenience:

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

This problem has 4 variables, 3 nonlinear objective variables, 2 nonlinear Jacobian variables, 2 nonlinear constraints and 1 linear constraint. The objective has some linear terms that we include as an extra “free row” (with infinite bounds). The calling program must assign the values

```

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

```

Subroutine `funobj` works with the nonlinear objective variables  $(x_1, x_2, x_3)$ . Since  $x_3$  occurs only linearly in the constraints, we have placed it *after* the nonlinear Jacobian variables  $(x_1, x_2)$ .

For interest, we test `mode` to economize on gradient evaluations (even though they are cheap here). Note that `No derivative linesearch` would have to be specified, otherwise all entries would have `mode = 2`.

```

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

integer          mode, nnObj, nState
double precision fObj
double precision x(nnObj), gObj(nnObj)

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

* =====
* Toy NLP problem from the SNOPT User's Guide.
* =====
double precision sum

sum = x(1) + x(2) + x(3)

if (mode .eq. 0 .or. mode .eq. 2) then
  fObj = sum*sum
end if

if (mode .eq. 1 .or. mode .eq. 2) then
  sum = 2.0d+0*sum
  gObj(1) = sum
  gObj(2) = sum
  gObj(3) = sum
end if

* end of funobj for toy NLP.
end

```

Subroutine `funcon` involves only  $(x_1, x_2)$ . For convenience we treat the Jacobian as a dense matrix. In Fortran it is preferable to access (large) two-dimensional arrays columnwise, as shown.

Since `funcon` is called before `funobj`, we test `nState` here to print a message on the first and last entries.

```

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

integer         mode, nnCon, nnJac, neJac, nState
double precision x(nnJac), fCon(nnCon), gCon(nnCon,nnJac)

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

*  =====
*  Toy NLP problem from the SNOPT User's Guide.
*  =====
integer         nout

nout = 9

*  -----
*  First entry.  Print something.
*  -----
if (nState .eq. 1) then
  if (nout .gt. 0) write(nout, '(/a)') ' This is problem Toy'
end if

if (mode .eq. 0 .or. mode .eq. 2) then
  fCon( 1) = x(1)**2 + x(2)**2
  fCon( 2) = x(1)**4 + x(2)**4
end if

if (mode .ge. 1) then

*   Jacobian elements for column 1.

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

*   Jacobian elements for column 2.

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

*  -----
*  Last entry.
*  -----
if (nState .ge. 2) then
  if (nout .gt. 0) write(nout, '(/a)') ' Finished problem Toy'
end if

*  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
  Solution                  Yes
End SNOPT options
```

We call such data a SPECS file because it specifies various options. The file starts with the keyword `Begin` and ends with `End`. Each line specifies a single option in free format, 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.

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.

### 6.1. 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  $\epsilon$ , 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      * 1-line major iteration log
  Minor print level          0      * no minor iteration log
  Print file                 9      *
  Summary file               6      * typically the screen
  Print frequency            1      * minor iterations log on PRINT file
  Summary frequency          1      * minor iterations log on SUMMARY file
  Solution                   Yes    * on the PRINT file
* Suppress options listing   * default: options are listed
```

---

* Convergence Tolerances		
Major feasibility tolerance	1.0e-6	* target nonlinear constraint violation
Major optimality tolerance	1.0e-6	* target complementarity gap
Minor feasibility tolerance	1.0e-6	* for satisfying the QP bounds
Minor optimality tolerance	1.0e-6	* target value for reduced gradients
* Derivative checking		
Verify level	0	* 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	*
* Scale Print		
		* default: scales are not printed
* 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	100.0	* used only during elastic mode
Iterations limit	10000	* or $20m$ if that is more
Partial price	1	* 10 for large LPs
* SQP method		
Minimize		* (opposite of Maximize)
* Feasible point		* (alternative to Max or Min)
* Feasible Exit		* (get feasible before exiting)
Major iterations limit	1000	* or $m$ if that is more
Minor iterations limit	500	* or $3m$ if that is more
Major step limit	2.0	*
Superbasics limit	500	* or $n_1 + 1$ if that is less
Derivative level	3	* assumes all gradients are known
Derivative linesearch		*
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) $^{\frac{1}{3}}$
Violation limit	10.0	* unscaled constraint violation limit
Unbounded step size	1.0e+18	*
Unbounded objective	1.0e+15	*
* Hessian approximation		
Hessian	Full memory	* default if $n \leq 75$
Hessian	Limited memory	* default if $n > 75$
Hessian frequency	999999	* for full Hessian (never reset)
Hessian updates	20	* for limited memory Hessian
Hessian flush	999999	* no flushing

---

```
* Frequencies
  Check frequency          60      * test row residuals  $\|Ax - s\|$ 
  Expand frequency        10000   * for anti-cycling procedure
  Factorization frequency  50     * 100 for LPs
  Save frequency          100     * save basis map

* 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       * different from printed solution

* Partitions of cw, iw, rw
  Total character workspace  lencw  *
  Total integer workspace   leniw  *
  Total real workspace      lenrw  *
  User character workspace   500  *
  User integer workspace    500  *
  User real workspace       500  *
End of SPECS file checklist
```

## 6.2. Subroutine snInit

Subroutine `snInit` must be called before any other SNOPT routines. It defines the PRINT and SUMMARY files, prints a title on both files, and sets all user options to be undefined. (`snopt` will later check the options and set undefined ones to default values.)

---

```

subroutine snInit( iPrint, iSumm,
$                cw, lencw, iw, leniw, rw, lenrw )

integer          iPrint, iSumm
integer          lencw, leniw, lenrw
character*8      cw(lencw)
integer          iw(leniw)
double precision rw(lenrw)

```

### On entry:

`iPrint` defines a unit number for the PRINT file. Typically `iPrint = 9`.

On some systems, the file may need to be opened before `snInit` is called. If `iPrint ≤ 0`, there will be no PRINT file output.

`iSumm` defines a unit number for the SUMMARY file. Typically `iSumm = 6`. (In an interactive environment, this usually denotes the screen.)

On some systems, the file may need to be opened before `snInit` is called. If `iSumm ≤ 0`, there will be no SUMMARY file output.

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` must be the same arrays that are passed to `snopt` and other routines. They must all have length 500 or more.

### On exit:

Some elements of `cw`, `iw`, `rw` are given values to indicate that most optional parameters are undefined.

---

### 6.3. Subroutine `snSpec`

Subroutine `snSpec` may be called to input a SPECS file (to specify options for a subsequent call of `snopt`).

---

```
subroutine snSpec( iSpecs, inform,  
$                cw, lencw, iw, leniw, rw, lenrw )  
  
integer          iSpecs, inform  
integer          lencw, leniw, lenrw  
character*8      cw(lencw)  
integer          iw(leniw)  
double precision rw(lenrw)
```

**On entry:**

`iSpecs` is a unit number for the SPECS file (`iSpecs > 0`). Typically `iSpecs = 4`.

On some systems, the file may need to be opened before `snSpec` is called.

**On exit:**

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` contain the specified options.

`inform` is 0 if the SPECS file was successfully read. Otherwise, it returns the number of errors encountered.

---

#### 6.4. Subroutines `snset`, `snseti`, `snsetr`

These routines specify a single option that might otherwise be defined in one line of a SPECS file.

---

```

subroutine snset ( buffer,          iPrint, iSumm, inform,
$               cw, lencw, iw, leniw, rw, lenrw )
subroutine snseti( buffer, ivalue, iPrint, iSumm, inform,
$               cw, lencw, iw, leniw, rw, lenrw )
subroutine snsetr( buffer, rvalue, iPrint, iSumm, inform,
$               cw, lencw, iw, leniw, rw, lenrw )

character*(*)   buffer
integer         ivalue, iPrint, iSumm, inform
double precision rvalue
integer         lencw, leniw, lenrw
character*8     cw(lencw)
integer         iw(leniw)
double precision rw(lenrw)

```

##### On entry:

`buffer` is a string to be decoded. Use `snset` if the string contains all relevant data. For example, if the value 1000 is known at compile time, say

```
call snset ( 'Iterations 1000',    iPrint, iSumm, inform, ... )
```

Restriction:  $\text{len}(\text{buffer}) \leq 72$  (`snset`) or  $\leq 55$  (`snseti` and `snsetr`).

`ivalue` is an integer value associated with the keyword in `buffer`. Use `snseti` if it is convenient to define the value at run time. For example, the following allows the iterations limit to be computed:

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

`rvalue` is a real value associated with the keyword in `buffer`. The following illustrates how the LU stability tolerance could be defined at run time:

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

`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 for the first call to the `snset` routines.

##### On exit:

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

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` record the specified option.

---

### 6.5. Subroutines `sngetc`, `sngeti`, `sngetr`

These routines obtain the current value of a single option.

---

```

subroutine sngetc( buffer, cvalue, inform,
$               cw, lencw, iw, leniw, rw, lenrw )
subroutine sngeti( buffer, ivalue, inform,
$               cw, lencw, iw, leniw, rw, lenrw )
subroutine sngetr( buffer, rvalue, inform,
$               cw, lencw, iw, leniw, rw, lenrw )

character*(*)    buffer
character*8      cvalue
integer          ivalue, inform
double precision rvalue
integer          lencw, leniw, lenrw
character*8      cw(lencw)
integer          iw(leniw)
double precision rw(lenrw)

```

#### On entry:

`buffer` is a string to be decoded. Restriction:  $\text{len}(\text{buffer}) \leq 72$ .

`inform` should be 0 for the first call to the `snget` routines.

#### On exit:

`cvalue` is a string associated with the keyword in `buffer`. Use `sngetc` to obtain the names

...

For example, if ..., say

```
call sngetc( 'Bounds',    inform, ... )
```

`ivalue` is an integer value associated with the keyword in `buffer`. Example:

```
call sngeti( 'Iterations limit', itnlim, inform, ... )
```

`rvalue` is a real value associated with the keyword in `buffer`. Example:

```
call sngetr( 'LU factor tol', factol, inform, ...)
```

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

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` contain the required option value.

---

## 6.6. 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 be interrupted during the next few milliseconds (in the middle of the save). 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 usable 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 =  $\epsilon^{\frac{1}{3}} \approx 6.0\text{e-}6$

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 *j*th 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 *i*th 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>i</i> | <i>Meaning</i>  |
|----------|---|
| 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 <i>linear equalities</i> and <i>linear inequalities</i> 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>i</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>i</i> | <i>Meaning</i>  |
|----------|---|
| 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.)

Derivative linesearch	Default
Nnderivative 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 values to compute estimates of the step  $\alpha_k$ . If some analytic derivatives are not provided, or a **Nnderivative linesearch** is specified, SNOPT employs a linesearch based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

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 **Nnderivative linesearch** is selected, SNOPT signals the evaluation of the linesearch by calling `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 = $\epsilon^{1/2} \approx 1.5\text{e-}8$
---------------------	-------	--

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 8.3.

**Elastic weight**  $\omega$  Default = 100

This keyword determines the initial weight  $\gamma$  associated with problem NP( $\gamma$ ).

At any given major iteration  $k$ , elastic mode is started if the QP subproblem is infeasible, or the QP dual variables are larger in magnitude than  $\omega(1 + \|g(x_k)\|_2)$ , where  $g$  is the objective gradient. In either case, the QP is re-solved in elastic mode with  $\gamma = \omega(1 + \|g(x_k)\|_2)$ .

Thereafter,  $\gamma$  is increased (subject to a maximum allowable value) at any point that is optimal for problem NP( $\gamma$ ), but not feasible for NP. After the  $r$ th increase,  $\gamma = \omega 10^r(1 + \|g(x_{k_l})\|_2)$ , where  $x_{k_l}$  is the iterate at which  $\gamma$  was first needed.

**Expand frequency**  $i$  Default = 10000

This option is part of the EXPAND anti-cycling procedure [9] 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  $\delta$ . Over a period of  $i$  iterations, the tolerance actually used by SNOPT increases from  $0.5\delta$  to  $\delta$  (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.

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.

**Feasibility tolerance**  $t$  Default = 1.0e-6  
 see **Minor feasibility tolerance**

#### Feasible point only

This options means “Ignore the objective function” while finding a feasible point for the linear and nonlinear constraints. It can be used to check that the nonlinear constraints are feasible without altering the call to **snopt**.

#### Feasible exit

**Infeasible exit** ok Default

If SNOPT is about to terminate with nonlinear constraint violations, **Feasible exit** requests further effort to satisfy the nonlinear constraints while ignoring the objective function.

**Function precision**  $\epsilon_R$  Default =  $\epsilon^{0.8} \approx 3.7\text{e-}11$

The *relative function precision*  $\epsilon_R$  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  $\epsilon_R$  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,  $\epsilon_R$  should be the *absolute* precision. For example, if  $f(x) = 1.23456789\text{e-}4$  at some point and if the first 6 significant digits are known to be correct, the appropriate value for  $\epsilon_R$  would be 1.0e-10.)

- The default value of  $\epsilon_R$  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 Full memory** Default = Full if  $n_1 \leq 75$   
**Hessian Limited memory**

These options select the method for storing and updating the approximate Hessian. (SNOPT uses a quasi-Newton approximation to the Hessian of the Lagrangian. A BFGS update is applied after each major iteration.)

If **Hessian Full memory** is specified, the approximate Hessian is treated as a dense matrix and the BFGS updates are applied explicitly. This option is most efficient when the number of nonlinear variables  $n_1$  is not too large (say, less than 75). In this case, the storage requirement is fixed and one can expect 1-step Q-superlinear convergence to the solution.

**Hessian Limited memory** should be used on problems where  $n_1$  is very large. In this case a limited-memory procedure is used to update a diagonal Hessian approximation  $H_r$  a limited number of times. (Updates are accumulated as a list of vector pairs. They are discarded at regular intervals after  $H_r$  has been reset to their diagonal.)

**Hessian frequency**  $i$  Default = 999999

If **Hessian Full** is selected and  $i$  BFGS updates have already been carried out, the Hessian approximation is reset to the identity matrix. (For certain problems, occasional resets may improve convergence, but in general they should not be necessary.)

**Hessian Full memory** and **Hessian frequency** = 20 have a similar effect to **Hessian Limited memory** and **Hessian updates** = 20 (except that the latter retains the current diagonal during resets).

**Hessian updates**  $i$  Default = 20

If **Hessian Limited memory** is selected and  $i$  BFGS updates have already been carried out, all but the diagonal elements of the accumulated updates are discarded and the updating 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 convergence can sometimes be obtained with significantly fewer updates (e.g.,  $i = 5$ ).

**Insert file**  $f$  Default = 0

If  $f > 0$ , this references a file containing basis information in the format of Section 8.2.

- 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.

**Iterations limit**  $k$  Default =  $\max\{10000, 20m\}$

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

**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  $-\text{BigBnd}$ ). If  $r \leq 0$ , the default value is used.

**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  $t = 0.9$  requests just moderate accuracy in the linesearch.
- If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try  $t = 0.1, 0.01$  or  $0.001$ . The number of major iterations might decrease.
- If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. *If all gradients are known*, try  $t = 0.99$ . (The number of major iterations might increase, but the total number of function evaluations may decrease enough to compensate.)
- If not all gradients are known, a moderately accurate search remains appropriate. Each search will require only 1–5 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 8.3.

- 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 (LP) or 5.0 (NLP)  
 LU update tolerance  $r_2$  Default = 10.0 (LP) or 5.0 (NLP)

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  $\mu$  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,  $r_1 = 5.0$  (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

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

both  $r_1$  and  $r_2$  should be in the range  $1.0 \leq r_i < 4.0$ .

---

LU density tolerance	$r_1$	Default = 0.6
LU singularity tolerance	$r_2$	Default = $\epsilon^{2/3} \approx 3.7\text{e-}11$

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 terminated. The remaining matrix is factored by a dense LU procedure. 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  $j$ th 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 a larger tolerance  $r_2 = 1.0\text{e-}5$ , say, may help cause a judicious change of basis.

Major feasibility tolerance	$\epsilon_r$	Default = $1.0\text{e-}6$
-----------------------------	--------------	---------------------------

This specifies how accurately the nonlinear constraints should be satisfied. The default value of  $1.0\text{e-}6$  is appropriate when the linear and nonlinear constraints contain data to about that accuracy.

Let `rowerr` be the maximum nonlinear constraint violation, normalized by the size of the solution. It is required to satisfy

$$\text{rowerr} = \max_i \text{viol}_i / \|x\| \leq \epsilon_r, \quad (6.1)$$

where  $\text{viol}_i$  is the violation of the  $i$ th nonlinear constraint ( $i = 1 : \text{nnCon}$ ).

In the major iteration log, `rowerr` appears as the quantity labeled “Feasibl”. If some of the problem functions are known to be of low accuracy, a larger Major feasibility tolerance may be appropriate.

Major optimality tolerance	$\epsilon_d$	Default = $1.0\text{e-}6$
----------------------------	--------------	---------------------------

This specifies the final accuracy of the dual variables. On successful termination, SNOPT will have computed a solution  $(x, s, \pi)$  such that

$$\text{maxgap} = \max_j \text{gap}_j / \|\pi\| \leq \epsilon_d, \quad (6.2)$$

where  $\text{gap}_j$  is an estimate of the complementarity gap for variable  $j$  ( $j = 1 : n + m$ ). The gaps are computed from the final QP solution using the reduced gradients  $d_j = g_j - \pi^T a_j$  (where  $g_j$  is the  $j$ th component of the objective gradient,  $a_j$  is the associated column of the constraint matrix  $(A \ -I)$ , and  $\pi$  is the set of QP dual variables):

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

In the major iteration log, `maxgap` appears as the quantity labeled “Optimal”.

Major iterations limit  $k$  Default =  $\max\{1000, m\}$

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

Major print level  $p$  Default = 00001

This controls the amount of output to the PRINT and SUMMARY files each major iteration. Major print level 1 gives normal output for linear and nonlinear problems, and Major print level 11 gives addition details of the Jacobian factorization that commences each major iteration.

In general, the value being specified may be 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 as follows:

**s** a single line that gives a summary of each major iteration. (This entry in JFDXbs is not strictly binary since the summary line is printed whenever JFDXbs  $\geq 1$ ).

**b** BASIS statistics, i.e., information relating to the basis matrix whenever it is refactorized. (This output is always provided if JFDXbs  $\geq 10$ ).

**X**  $x_k$ , the nonlinear variables involved in the objective function or the constraints.

**D**  $\pi_k$ , 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 items JFDXbs, 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+01 BS      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 suppresses most output, except for error messages.

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  $\alpha$  over the range  $0 < \alpha \leq \beta$ , where  $\beta$  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  $\tilde{\beta} = r(1 + \|x\|)/\|p\|$  (where  $p$  is the search direction), and the first evaluation of  $f(x)$  is at the potentially smaller steplength  $\alpha_1 = \min(1, \tilde{\beta}, \beta)$ .

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 =  $\max\{1000, 5m\}$

This is the maximum number of minor iterations allowed for each QP subproblem in the SQP algorithm. Current experience is that the major iterations converge more reliably if the QP subproblems are allowed to solve accurately. Thus,  $k$  should be a large value.

In the major iteration log, a  $\tau$  at the end of a line indicates that the corresponding QP was terminated by the limit  $k$ .

Note that `Iterations limit` defines an independent limit on the *total* number of minor iterations (summed over all QP subproblems).

`Minor feasibility tolerance`  $t$  Default =  $1.0\text{e-}6$

SNOPT tries to ensure that all variables eventually satisfy their upper and lower bounds to within the tolerance  $t$ . This includes slack variables. Hence, general linear constraints should also be satisfied to within  $t$ .

Feasibility with respect to nonlinear constraints is judged by the `Major feasibility tolerance` (not by  $t$ ).

- If the bounds and linear constraints cannot be satisfied to within  $t$ , the problem is declared *infeasible*. 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.
- Nonlinear functions will be evaluated only at points that satisfy the bounds and linear constraints. If there are regions where a function 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  $t = 1.0\text{e-}6$ , 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.)
- If `Scale option`  $\geq 1$ , feasibility is defined in terms of the *scaled* problem (since it is then more likely to be meaningful).
- In reality, SNOPT uses  $t$  as a feasibility tolerance for satisfying the bounds on  $x$  and  $s$  in each QP subproblem. If the sum of infeasibilities cannot be reduced to zero, the QP subproblem is declared infeasible. SNOPT is then in *elastic mode* thereafter (with only the linearized nonlinear constraints defined to be elastic). See the `Elastic` options.

Minor optimality tolerance  $t$  Default = 1.0e-6

This is used to judge optimality for each QP subproblem. Let the QP reduced gradients be  $d_j = g_j - \pi^T a_j$ , where  $g_j$  is the  $j$ th component of the QP gradient,  $a_j$  is the associated column of the QP constraint matrix, and  $\pi$  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

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

respectively, and if  $|d_j/\|\pi\| \leq t$  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 large scale factor on the objective function. The quantity actually used is defined by

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

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

Minor print level  $k$  Default = 0

This controls the amount of output to the PRINT and SUMMARY files during solution of the QP subproblems. The value of  $k$  has the following effect:

- 0 No minor iteration output except error messages.
- $\geq 1$  A single line of output each minor iteration (controlled by **Print frequency** and **Summary frequency**).
- $\geq 10$  Basis factorization statistics generated during the periodic refactorization of the basis (see **Factorization frequency**). Statistics for the *first factorization* each major iteration are controlled by the **Major print level**.

New Basis file  $f$  Default = 0

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

The first line of the file will contain the word **Proceeding** if the run is still in progress. At the end of a run a further basis map will be saved, 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 8.1.

The file will usually have been output previously as a **New Basis file**. It 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 Minor 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 ( $A - I$ ) 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 =  $\epsilon^{2/3} \approx 3.7\text{e-}11$

During solution of QP subproblems, 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  $x+ap$  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.
- Elements of  $p$  are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance  $r$ .
- 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 file  $f$  Default = 15  
 Print frequency  $k$  Default = 1

If  $f > 0$  and **Minor print level**  $> 0$ , a line of the QP iteration log will be printed on file  $f$  every  $k$ th minor iteration.

Punch file  $f$  Default = 0

If  $f > 0$ , the final solution obtained will be output to file  $f$  in the format described in Section 8.2. 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  $k$ th iteration. A BACKUP BASIS file will also be saved if specified.

Scale option  $i$  Default = 2 (LP) or 1 (NLP)

Scale tolerance  $r$  Default = 0.9

Scale Print

Three scale options are available as follows:

$i$             *Meaning*

0 No scaling. This is recommended if it is known that  $x$  and the constraint matrix (and Jacobian) never have very large elements (say, larger than 1000).

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 [5]). This will sometimes improve the performance of the solution procedures.

2 All constraints and variables are scaled by the iterative procedure. Also, an additional scaling is performed that takes into account columns of  $(A - I)$  that are fixed or have positive lower bounds or negative upper bounds.

If nonlinear constraints are present, the scales depend on the Jacobian at the first point that satisfies the linear constraints. **Scale option 2** should therefore be used only if (a) a good starting point is provided, and (b) the problem is not highly nonlinear.

**Scale tolerance** 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:

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

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.

**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  $j > n$ .

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 first and third 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 `1p, e16.6` format, including “infinite” bounds, which will have magnitude `1.000000e+20`.
- To see more significant digits in the printed solution, it is sometimes useful to make `f` refer to the `PRINT` file (i.e., make it the same as the number specified by `Print file`).

<code>Start Objective Check at Column</code>	<code>k</code>	Default = 1
<code>Start Constraint Check at Column</code>	<code>k</code>	Default = 1
<code>Stop Objective Check at Column</code>	<code>l</code>	Default = $n'_1$
<code>Stop Constraint Check at Column</code>	<code>l</code>	Default = $n''_1$

If `Verify level > 0`, 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.
- 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.

<code>Summary file</code>	<code>f</code>	Default = 6
<code>Summary frequency</code>	<code>k</code>	Default = 100

If `f > 0` and `Minor print level > 0`, a line of the QP iteration log will be output to file `f` every `k`th minor iteration.

<code>Superbasics limit</code>	<code>i</code>	Default = $\min\{500, n_1 + 1\}$
--------------------------------	----------------	----------------------------------

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.

### Suppress Parameters

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

Total real workspace	maxrw	Default = lenrw
Total integer workspace	maxiw	Default = leniw
Total character workspace	maxcw	Default = lencw
User real workspace	maxru	Default = 500
User integer workspace	maxiu	Default = 500
User character workspace	maxcu	Default = 500

These options may be used to confine SNOPT to certain parts of its workspace arrays `iw`, `rw`. (The arrays are defined by the last six parameters of `snopt`.)

The `Total ...` options place an *upper* limit on `snopt`'s workspace. They may be useful on machines with virtual memory. For example, some systems allow a very large array `rw(lenrw)` to be declared at compile time 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 restrict SNOPT to the lower end of `rw` in order 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.)

If `snopt`'s "user" parameters `ru`, `lenru` happen to be the same as `rw`, `lenrw`, the nonlinear function routines will be free to use `ru(maxrw + 1 : lenru)` for their own purpose. Similarly for the other work arrays.

The `User ...` options place a *lower* limit on `snopt`'s workspace (not counting the first 500 elements). Again, if `snopt`'s parameters `ru`, `lenru` happen to be the same as `rw`, `lenrw`, the function routines will be free to use `ru(501 : maxru)` for their own purpose. Similarly for the other work arrays.

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 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  $\alpha$  varies. If  $|f|$  exceeds  $f_{\max}$  or  $\alpha$  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	$l$	Default = 0
--------------	-----	-------------

This option refers to finite-difference checks on the gradients computed by the user routines `funobj` and `funcon`. Gradients are checked at the first point that satisfies all bounds and linear constraints.

$l$                       *Meaning*

- 0 Only a "cheap" test will be performed, requiring 2 calls to `funcon` and 3 calls to `funobj`.
- 1 Individual objective gradients will be checked (with a more reliable test). A key of the form "OK" or "Bad?" indicates whether or not each component appears to be correct.
- 2 Individual columns of the Jacobian (constraint gradients) will be checked.
- 3 Options 2 and 1 will both occur (in that order).

-1 No gradient checking will occur.

`Verify level 3` should be specified whenever a new function routine is being developed. The `Start` and `Stop` keywords may be used to limit the number of nonlinear variables checked. Missing gradients are not checked, so they result in no overhead.

`Violation limit`  $\tau$  Default = 10

This keyword defines an absolute limit on the magnitude of the maximum constraint violation after the line search. On completion of the line search, the new iterate  $x_{k+1}$  satisfies the condition

$$v_i(x_{k+1}) \leq \tau \max\{1, v_i(x_0)\}, \quad (6.3)$$

where  $x_0$  is the point at which the nonlinear constraints are first evaluated and  $v_i(x)$  is the  $i$ th nonlinear constraint violation  $v_i(x) = \max(0, l_i - F_i(x), F_i(x) - u_i)$ .

The effect of this violation limit is to restrict the iterates to lie in an *expanded* feasible region whose size depends on the magnitude of  $\tau$ . This makes it possible to keep the iterates within a region where the objective is expected to be well-defined and bounded below. If the objective is bounded below for all values of the variables, then  $\tau$  may be any large positive value.

## 7. Output

Subroutine `snInit` specifies unit numbers for the PRINT and SUMMARY files described in this section. The files can be redirected with the `Print file` and `Summary file` options (or suppressed).

### 7.1. The PRINT file

If `Print file > 0`, the following information is output to the PRINT file during the solution process. All printed lines are less than 125 characters.

- A listing of the SPECS file, if any.
- A listing of the options that were or could have been set in the SPECS file.
- An estimate of the working storage needed and the amount available.
- Some statistics about the problem being solved.
- The storage available for the LU factors of the basis matrix.
- A summary of the scaling procedure, if `Scale option > 0`.
- 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.

### 7.2. The major iteration log

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

<i>Label</i>	<i>Description</i>
<code>Maj</code>	The current major iteration number.
<code>Mnr</code>	is the number of iterations required by both the feasibility and optimality phases of the QP subproblem. Generally, <code>Mnr</code> will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see §2).
<code>Step</code>	The step length $\alpha$ 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.
<code>nObj</code>	The number of times subroutine <code>funobj</code> has been called to evaluate the nonlinear objective function. Evaluations needed for the estimation of the gradients by finite differences are not included. <code>nObj</code> is printed as a guide to the amount of work required for the linesearch.
<code>nCon</code>	The number of times subroutine <code>funcon</code> has been called to evaluate the nonlinear constraint functions.
<code>Merit</code>	is the value of the augmented Lagrangian merit function (see (2.2)). 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 the constraints are linear, this item is labeled **Objective**, the value of the objective function. It will decrease monotonically to its optimal value.

**Feasbl** is the value of **rowerr**, the maximum component of the scaled nonlinear constraint residual (6.1). The solution is regarded as acceptably feasible if **Feasbl** is less than the **Major feasibility tolerance**.

If the constraints are linear, all iterates are feasible and this entry is not printed.

**Optimal** is the value of **maxgap**, the maximum complementarity gap (6.2). It is an estimate of the degree of nonoptimality of the reduced costs. Both **Feasbl** and **Optimal** are small 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,  $LU = \text{lenL} + \text{lenU}$ , 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 will tend to increase. As the solution is approached and the minor iterations decrease towards zero, **LU** will reflect the number of nonzeros in the  $LU$  factors at the start of the QP subproblem.

If the constraints are linear, refactorization is subject only to the **Factorize frequency**, and **LU** will tend to increase between factorizations.

**Swp** The number of columns of the basis matrix  $B$  that were swapped with columns of  $S$  to improve the condition of  $B$ . The swaps are determined by an  $LU$  factorization of the rectangular matrix  $B_S = (B \ S)^T$  with stability being favored more than sparsity.

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

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.

PD is a two-letter indication of the status of the convergence tests involving primal and dual feasibility of the iterates (see 6.1) and (6.2) 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.

The summary line may include additional code characters that indicate what happened during the course of the iteration.

<i>Code</i>	<i>Meaning</i>
c	Central differences have been used to compute the unknown components of the objective and constraint gradients. A switch to central differences is made if either the linesearch gives a small step, or $x$ is close to being optimal. In some cases, it may be necessary to re-solve the QP subproblem with the central-difference gradient and Jacobian.
d	During the linesearch it was necessary to decrease the step in order to obtain a maximum constraint violation conforming to the value of <code>Violation limit</code> .
l	The norm-wise change in the variables was limited by the value of the <code>Major step limit</code> . If this output occurs repeatedly during later iterations, it may be worthwhile increasing the value of <code>Major step limit</code> .
i	If SNOPT is not in elastic mode, an “i” signifies that the QP subproblem is infeasible. This event triggers the start of nonlinear elastic mode, which remains in effect for all subsequent iterations. Once in elastic mode, the QP subproblems are associated with the elastic problem $NP(\gamma)$ . If SNOPT is already in elastic mode, an “i” indicates that the minimizer of the elastic subproblem does not satisfy the linearized constraints. (In this case, a feasible point for the usual QP subproblem may or may not exist.)
M	An extra evaluation of the problem functions was needed to define an acceptable positive-definite quasi-Newton update to the Lagrangian Hessian. This modification is only done when there are nonlinear constraints.
m	This is the same as “M” except that it was also necessary to modify the update to include an augmented Lagrangian term.
R	The approximate Hessian has been reset by discarding all but the diagonal elements. This reset will be forced periodically by the <code>Hessian frequency</code> and <code>Hessian updates</code> keywords. However, it may also be necessary to reset an ill-conditioned Hessian from time to time.
r	The approximate Hessian was reset after ten consecutive major iterations in which no BFGS update could be made. The diagonals of the approximate Hessian are retained if at least one update has been done since the last reset. Otherwise, the approximate Hessian is reset to the identity matrix.
s	A self-scaled BFGS update was performed. This update is always used when the Hessian approximation is diagonal, and hence always follows a Hessian reset.
S	This is the same as a “s” except that it was necessary to modify the self-scaled update to maintain positive definiteness.
n	No positive-definite BFGS update could be found. The approximate Hessian is unchanged from the previous iteration.

---

f	the Hessian was reset to the identity matrix under control of the keyword <code>Hessian flush</code> . This can happen only once in the course of the minimization.
t	The minor iterations were terminated at the <code>Minor iteration limit</code> .
u	The QP subproblem was unbounded.
w	A weak solution of the QP subproblem was found.

### 7.3. The minor iteration log

If `Minor print level`  $> 0$ , one line of information is output to the PRINT file every  $k$ th 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 a nonbasic variable is selected to become superbasic (in addition to those already in the superbasic set). The selected variable is denoted by `jq`. Variable `jq` often becomes basic immediately. Otherwise it remains superbasic, unless it reaches its opposite bound and returns to the nonbasic set.

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

<i>Label</i>	<i>Description</i>
<code>Itn</code>	The current iteration number.
<code>pp</code>	The Partial Price indicator. The variable selected by the last PRICE operation came from the $pp$ th partition of $A$ and $-I$ . <code>pp</code> is set to zero when the basis is refactored.
<code>dj</code>	This is the reduced cost (or reduced gradient) of the variable <code>jq</code> selected by PRICE at the start of the present iteration. Algebraically, <code>dj</code> is $d_j = g_j - \pi^T a_j$ for $j = jq$ , where $g_j$ is the gradient of the current objective function, $\pi$ is the vector of dual variables for the QP subproblem, and $a_j$ is the $j$ th column of $(A - I)$ .  Note that <code>dj</code> is the 1-norm of the reduced-gradient vector at the start of the iteration, just after the PRICE operation.
<code>+SBS</code>	The variable <code>jq</code> selected by PRICE to be added to the superbasic set.
<code>-SBS</code>	The variable chosen to leave the set of superbasics. It has become basic if the entry under <code>-B</code> is nonzero; otherwise it has become nonbasic.
<code>-BS</code>	The variable removed from the basis (if any) to become nonbasic.
<code>-B</code>	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.
<code>Step</code>	The step length $\alpha$ 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 ( <code>+SBS</code> $> 0$ ), <code>Step</code> 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.
<code>Pivot</code>	If column $a_q$ replaces the $r$ th column of the basis $B$ , <code>Pivot</code> is the $r$ th element of a vector $y$ satisfying $By = a_q$ . Wherever possible, <code>Step</code> is chosen to avoid extremely small values of <code>Pivot</code> (since they cause the basis to be nearly singular). In rare cases, it may be necessary to increase the <code>Pivot tolerance</code> to exclude very small elements of $y$ from consideration during the computation of <code>Step</code> .

- 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 *after* 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 after the present iteration. It usually decreases at each nonzero `Step`, but if `nInf` decreases by 2 or more, `sInf` may occasionally increase.

In elastic mode, the heading is changed to `Composite Obj`, and the value printed decreases monotonically.

If `nInf` = 0, the value printed is `Objective`, the QP objective function after the present iteration.

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

<i>Label</i>	<i>Description</i>
<b>Norm rg</b>	The norm of the reduced-gradient vector at the start of the iteration. (It is the 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.
<b>nS</b>	The current number of superbasic variables.
<b>cond Hz</b>	See the major iteration log.

#### 7.4. Basis factorization statistics

If `Major print level`  $\geq 10$ , the following items are output to the PRINT file whenever the basis  $B$  or the rectangular matrix  $B_S = (B \ S)^T$  is factorized before solution of the next QP subproblem.

Note that  $B_S$  may be factorized at the start of just some of the major iterations. It is immediately followed by a factorization of  $B$  itself.

Gaussian elimination is used to compute a sparse LU factorization of  $B$  or  $B_S$ , where  $PLP^T$  and  $PUQ$  are lower and upper triangular matrices for some permutation matrices  $P$  and  $Q$ . Stability is ensured as described under `LU factor tolerance` in §6.6.

If `Minor print level`  $\geq 10$ , the same items are printed during the QP solution whenever the current  $B$  is factorized.

<i>Label</i>	<i>Description</i>
--------------	--------------------

---

<b>Factorize</b>	The number of factorizations since the start of the run.														
<b>Demand</b>	A code giving the reason for the present factorization.														
	<table> <thead> <tr> <th><i>Code</i></th> <th><i>Meaning</i></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>First LU factorization.</td> </tr> <tr> <td>1</td> <td>The number of updates reached the <b>Factorization Frequency</b>.</td> </tr> <tr> <td>2</td> <td>The nonzeros in the updated factors have increased significantly.</td> </tr> <tr> <td>7</td> <td>Not enough storage to update factors.</td> </tr> <tr> <td>10</td> <td>Row residuals too large (see the description of <b>Check Frequency</b>).</td> </tr> <tr> <td>11</td> <td>Ill-conditioning has caused inconsistent results.</td> </tr> </tbody> </table>	<i>Code</i>	<i>Meaning</i>	0	First LU factorization.	1	The number of updates reached the <b>Factorization Frequency</b> .	2	The nonzeros in the updated factors have increased significantly.	7	Not enough storage to update factors.	10	Row residuals too large (see the description of <b>Check Frequency</b> ).	11	Ill-conditioning has caused inconsistent results.
<i>Code</i>	<i>Meaning</i>														
0	First LU factorization.														
1	The number of updates reached the <b>Factorization Frequency</b> .														
2	The nonzeros in the updated factors have increased significantly.														
7	Not enough storage to update factors.														
10	Row residuals too large (see the description of <b>Check Frequency</b> ).														
11	Ill-conditioning has caused inconsistent results.														
<b>Iteration</b>	The current iteration number.														
<b>Infeas</b>	The current number of infeasibilities.														
<b>Objective</b>	If <b>Infeas</b> > 0, this is the current sum of infeasibilities. If <b>Infeas</b> = 0, it is the current QP objective function.														
<b>Nonlinear</b>	The number of nonlinear variables in the current basis $B$ (not printed if $B_S$ is factorized).														
<b>Linear</b>	The number of linear variables in $B$ (not printed if $B_S$ is factorized).														
<b>Slacks</b>	The number of slack variables in $B$ (not printed if $B_S$ is factorized).														
<b>Elms</b>	The number of nonzero matrix elements in $B$ . (not printed if $B_S$ is factorized).														
<b>Density</b>	The percentage nonzero density of $B$ , $100 \times \text{Elms} / (m^2)$ , where $m$ is the number of rows in the problem ( $m = \text{Nonlinear} + \text{Linear} + \text{Slacks}$ ).														
<b>Comprsns</b>	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.														
<b>Merit</b>	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. <b>Merit</b> is the average of $m$ such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization.														
<b>lenL</b>	The number of nonzeros in $L$ . On most machines, each nonzero is represented by one 8-byte <b>real</b> and two 4-byte <b>integer</b> data types.														
<b>lenU</b>	The number of nonzeros in $U$ . The storage required for each nonzero is the same as for the nonzeros of $L$ .														
<b>Increase</b>	The percentage increase in the number of nonzeros in $L$ and $U$ relative to the number of nonzeros in $B$ ; i.e., $100 \times (\text{lenL} + \text{lenU} - \text{Elms}) / \text{Elms}$ .														
<b>m</b>	is $m$ , the number of rows in the problem. Note that $m = \text{Ut} + \text{Lt} + \text{bp}$ .														
<b>Ut</b>	is the number of triangular rows of $B$ at the top of $U$ .														
<b>d1</b>	is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.														

---

<b>Lmax</b>	The maximum subdiagonal element in the columns of $L$ . This will be no larger than the <code>LU factor tolerance</code> .
<b>Bmax</b>	The maximum nonzero element in $B$ .
<b>Umax</b>	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 <b>Umax</b> . If the basis is strictly triangular, <i>none</i> of the elements of $B$ will contribute, and <b>Umax</b> will be zero.)  Ideally, <b>Umax</b> should not be substantially larger than <b>Bmax</b> . If it is several orders of magnitude larger, it may be advisable to reduce the <code>LU factor tolerance</code> to some value nearer 1.0.  <b>Umax</b> is not printed if $B_S$ is factorized.
<b>Umin</b>	The smallest <i>diagonal</i> element of $PUQ$ in absolute magnitude.
<b>Growth</b>	The ratio $Umax/Bmax$ , which should not be too large (see above).  As long as <b>Lmax</b> is not large (say 10.0 or less), $\max\{Bmax, Umax\} / Umin$ gives an estimate of the condition number of $B$ . If this number is extremely large, the basis is nearly singular and some numerical difficulties might occur. (However, an effort is made to avoid near-singularity by using slacks to replace columns of $B$ that would have made <b>Umin</b> extremely small. Messages are issued to this effect, and the modified basis is refactored.)
<b>Lt</b>	is the number of triangular columns of $B$ at the left of $L$ .
<b>bp</b>	is the size of the “bump” or block to be factorized nontrivially after the triangular rows and columns of $B$ have been removed.
<b>d2</b>	is the number of columns remaining when the density of the basis matrix being factorized reached 0.6.

### 7.5. Crash statistics

If `Major print level`  $\geq 10$ , the following items are output to the PRINT file when `Start = 'Cold'` and no basis file is loaded. They refer to the number of columns that the CRASH procedure selects during several passes through  $A$  while searching for a triangular basis matrix.

<i>Label</i>	<i>Description</i>
<b>Slacks</b>	is the number of slacks selected initially.
<b>Free cols</b>	is the number of free columns in the basis, including those whose bounds are rather far apart.
<b>Preferred</b>	is the number of “preferred” columns in the basis (i.e., $hs(j) = 3$ for some $j \leq n$ ). It will be a subset of the columns for which $hs(j) = 3$ was specified.
<b>Unit</b>	is the number of unit columns in the basis.
<b>Double</b>	is the number of columns in the basis containing 2 nonzeros.
<b>Triangle</b>	is the number of triangular columns in the basis with 3 or more nonzeros.
<b>Pad</b>	is the number of slacks used to pad the basis (to make it a nonsingular triangle).

## 7.6. EXIT conditions

When the solution procedure terminates, an `EXIT --` message is printed to summarize the final result. Here we describe each message and suggest possible courses of action.

The number associated with each EXIT is the output value of the integer variable `inform`.

The following messages arise when a solution exists (though it may not be optimal). A BASIS file may be saved, and the solution will be output to the PRINT or SOLUTION files 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. 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 input during 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 modelers 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. Some of the variables or slacks may lie outside their bounds more than desired, especially if scaling was requested. `Max Primal infeas` refers to the largest bound infeasibility and which variable is involved. If it is too large, consider restarting with a smaller `Minor feasibility tolerance` (say 10 times smaller) and perhaps `Scale` option 0.

Similarly, `Max Dual infeas` indicates which variable is most likely to be at a non-optimal value. Broadly speaking, if

$$\text{Max Dual infeas}/\text{Norm of pi} = 10^{-d},$$

then the objective function would probably change in the *d*th significant digit if optimization could be continued. If *d* seems too large, consider restarting with smaller `Major` and `Minor optimality tolerances`.

Finally, `Nonlinear constraint violn` shows the maximum infeasibility for nonlinear rows. If it seems too large, consider restarting with a smaller `Major feasibility tolerance`.

**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 and slacks. Among all the points satisfying the general constraints  $Ax - s = 0$ , there is apparently no point that satisfies the bounds on  $x$  and  $s$ . Violations as small as the **Minor 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 each 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. The subproblem includes the original QP objective and the sum of the infeasibilities—suitably weighted using the **Elastic weight** parameter. In elastic mode, some of the bounds on the nonlinear rows “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)**

**EXIT -- violation limit exceeded -- the problem may be unbounded**

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 §6.6), 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.

The second message indicates an abnormal termination while enforcing the limit on the constraint violations. This exit implies that the objective is not bounded below in the feasible region defined by expanding the bounds by the value of the **Violation limit**.

**3 EXIT -- major iteration limit exceeded**

**EXIT -- minor iteration limit exceeded**

**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 accuracy could not be achieved

A feasible solution has been found, but the requested accuracy in the dual infeasibilities could not be achieved. An abnormal termination has occurred, but SNOPT is within  $10^{-2}$  of satisfying the Major optimality tolerance. Check that the Major optimality tolerance is not too small.

#### 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 (about  $\frac{1}{2}s^2$  double words).

#### 6 EXIT -- constraint and objective values could not be calculated

This exit occurs if a value `mode`  $\leq -1$  is set 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 SNOPT. If the default versions of `funobj` and `funcon` are ever called, they issue a warning message and then set `mode` to terminate the run.

#### 7 EXIT -- subroutine funobj seems to be giving incorrect gradients

A check has been made on some individual elements of the objective gradient array at the first point that satisfies the linear constraints. 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* in `funobj`. A simple omission (such as forgetting to divide `fObj` 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 `gObj(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 may have difficulty.

#### 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 `funcon`.

#### 9 EXIT -- the current point cannot be improved upon

Several circumstances could lead to this exit.

1. Subroutines `funobj` or `funcon` could be returning accurate function values but inaccurate gradients (or vice versa). This is the most likely cause. Study the comments given for EXIT 7 and 8, and do your best to ensure that the coding is correct.
2. The function and gradient values could be consistent, but their precision could be too low. For example, accidental use of a `real` data type when `double precision` was intended 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

$$\begin{array}{ll} \text{Function precision} & t \\ \text{Major optimality tolerance} & \sqrt{t} \end{array}$$

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 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  $Ax - s = 0$  sufficiently well.

This probably means that the current basis is very ill-conditioned. Try `Scale option 1` if scaling has not yet been used and there are some 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 §7.4, and set the `LU factor tolerance` to 2.0 (or possibly even smaller, but not less than 1.0).

#### 12 EXIT -- terminated from subroutine s1User

The user has set the value `iAbort = 1` in subroutine `s1User`. SNOPT assumes that you want the problem to be abandoned forthwith.

If the following exits occur during the *first* basis factorization, the primal and dual variables `x` and `pi` will have their original input values. BASIS files will be saved if requested, but certain values in the printed solution will not be meaningful.

#### 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 `iw` and `rw` should be recompiled with a larger dimensions for those arrays. The new values should also be assigned to `leniw` and `lenrw`.

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  $U$  were respectively zero or smaller than a certain tolerance.) The associated variables are

replaced by slacks and the modified basis is refactorized, but singularity persists. This must mean that the problem is badly scaled, or the `LU factor tolerance` is too much larger than 1.0.

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 labeled `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 another 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 an 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 line, or some of the 2's in the map did not have a corresponding "`j xj`" entry following the map.

32 EXIT -- system error. Wrong no. of basic variables: nnn

This exit should never happen. It may indicate that the wrong SNOPT source files have been compiled, or incorrect parameters have been used in the call to subroutine `snopt`.

Check that all integer variables and arrays are declared `integer` in your calling program (including those beginning with `h!`), and that all "real" variables and arrays are declared consistently. (They should be `double precision` on most machines.)

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 storage array `rw(*)` is not large enough to provide workspace for the optimization procedure. Be sure that the `Superbasics limit` is not unreasonably large. Otherwise, see the advice for EXIT 20.

### 7.7. Solution output

At the end of a run, the final solution is 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 certain commercial systems, though there is no industry standard.

An example of the printed solution is given in §7. In general, numerical values are output with format `f16.5`. The maximum record length is 111 characters, including the first (carriage-control) character.

To reduce clutter, a dot “.” is printed for any numerical value that is exactly zero. The values  $\pm 1$  are also printed specially as `1.0` and `-1.0`. Infinite bounds ( $\pm 10^{20}$  or larger) are printed as `None`.

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

### The ROWS section

General linear constraints take the form  $l \leq Ax \leq u$ . The  $i$ th constraint is therefore of the form

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

and the value of  $a^T x$  is called the *row activity*. Internally, the linear constraints take the form  $Ax - s = 0$ , where the slack variables  $s$  should satisfy the bounds  $l \leq s \leq u$ . For the  $i$ th “row”, it is the slack variable  $s_i$  that is directly available, and it is sometimes convenient to refer to its state. Slacks may be basic or nonbasic (but not superbasic).

Nonlinear constraints  $\alpha \leq F_i(x) + a^T x \leq \beta$  are treated similarly, except that the row activity and degree of infeasibility are computed directly from  $F_i(x) + a^T x$  rather than from  $s_i$ .

<i>Label</i>	<i>Description</i>
<b>Number</b>	The value $n + i$ . This is the internal number used to refer to the $i$ th slack in the iteration log.
<b>Row</b>	The name of the $i$ th row.
<b>State</b>	The state of the $i$ th row relative to the bounds $\alpha$ and $\beta$ . The various states possible are as follows.
LL	The row is at its lower limit, $\alpha$ .
UL	The row is at its upper limit, $\beta$ .
EQ	The limits are the same ( $\alpha = \beta$ ).
BS	The constraint is not binding. $s_i$ is basic.
A key is sometimes printed before the <b>State</b> to give some additional information about the state of the slack variable.	
A	<i>Alternative optimum possible.</i> The slack is nonbasic, but its reduced gradient is essentially zero. This means that if the slack were allowed to start moving from its current value, there would be no change in the objective function. The values of the basic and superbasic variables <i>might</i> change, giving a genuine alternative solution. The values of the dual variables <i>might</i> also change.
D	<i>Degenerate.</i> The slack is basic, but it is equal to (or very close to) one of its bounds.
I	<i>Infeasible.</i> The slack is basic and is currently violating one of its bounds by more than the <b>Feasibility tolerance</b> .
N	<i>Not precisely optimal.</i> The slack is nonbasic. Its reduced gradient is larger than the <b>Optimality tolerance</b> .

*Note:* If **Scale option**  $> 0$ , the tests for assigning A, D, I, N are made on the scaled problem, since the keys are then more likely to be meaningful.

- 
- Activity** The row value  $a^T x$  (or  $F_i(x) + a^T x$  for nonlinear rows).
- 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**  $\alpha$ , the lower bound on the row.
- Upper limit**  $\beta$ , the upper bound on the row.
- Dual activity** The value of the dual variable  $\pi_i$ , often called the shadow price (or simplex multiplier) for the  $i$ th constraint. The full vector  $\pi$  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”  $x_j$ ,  $j = 1 : n$ . We assume that a typical variable has bounds  $\alpha \leq x_j \leq \beta$ .

- | <i>Label</i>  | <i>Description</i>   |
|---------------|--|
| <b>Number</b> | The column number, $j$ . This is the internal number used to refer to $x_j$ in the iteration log.            |
| <b>Column</b> | The name of $x_j$ .  |
| <b>State</b>  | The state of $x_j$ relative to the bounds $\alpha$ and $\beta$ . The various states possible are as follows. |
| LL            | $x_j$ is nonbasic at its lower limit, $\alpha$ .   |
| UL            | $x_j$ is nonbasic at its upper limit, $\beta$ .  |
| EQ            | $x_j$ is nonbasic and fixed at the value $\alpha = \beta$ .  |
| FR            | $x_j$ is nonbasic at some value strictly between its bounds: $\alpha < x_j < \beta$ .                        |
| BS            | $x_j$ is basic. Usually $\alpha < x_j < \beta$ .   |
| SBS           | $x_j$ is superbasic. Usually $\alpha < x_j < \beta$ .  |

A key is sometimes printed before the **State** to give some additional information about the state of  $x_j$ .

- A** *Alternative optimum possible.* The variable is nonbasic, but its reduced gradient is essentially zero. This means that if  $x_j$  were allowed to start moving from its current value, there would be no change in the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the dual variables *might* also change.
- D** *Degenerate.*  $x_j$  is basic, but it is equal to (or very close to) one of its bounds.
- I** *Infeasible.*  $x_j$  is basic and is currently violating one of its bounds by more than the **Feasibility tolerance**.
- N** *Not precisely optimal.*  $x_j$  is nonbasic. Its reduced gradient is larger than the **Optimality tolerance**.

*Note:* If **Scale option**  $> 0$ , the tests for assigning A, D, I, N are made on the scaled problem, since the keys are then more likely to be meaningful.

- Activity** The value of the variable  $x_j$ .

**Obj Gradient**  $g_j$ , the  $j$ th component of the gradient of the (linear or nonlinear) objective function. (If any  $x_j$  is infeasible,  $g_j$  is the gradient of the sum of infeasibilities.)

**Lower limit**  $\alpha$ , the lower bound on  $x_j$ .

**Upper limit**  $\beta$ , the upper bound on  $x_j$ .

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

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

### 7.8. The SOLUTION file

The information in a printed solution (§7.7) may be output as a SOLUTION file, according to the **Solution file** option (which may refer to the PRINT file if so desired). Infinite bounds appear as  $\pm 10^{20}$  rather than **None**. Other numerical values are output with format **1p, e16.6**.

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 for **backspace** statements.)

### 7.9. The SUMMARY file

If **Summary file**  $> 0$ , the following information is output to the SUMMARY file. (It is a brief form of the PRINT file.) All output lines are less than 72 characters.

- The **Begin** line from the SPECS file, if any.
- The basis file loaded, if any.
- A brief Major iteration log.
- A brief Minor iteration log.
- The **EXIT** condition and a summary of the final solution.

The following SUMMARY file is from the example of §3, using **Major print level 1** and **Minor print level 0**.

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

Begin Toy NLP problem

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 = 4.0000000000000000
```

## 8. 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
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 BASIS files are saved in that order every  $k$ th iteration, where  $k$  is the `Save frequency`.

NEW BASIS, 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:

```
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 options` 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. (The problem has 10 nonlinear constraints, 10 linear constraints, and 30 variables.) Selected column numbers are included to define significant data fields.

### 8.1. 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,  $M = m$ , the number of rows in the constraint matrix,  $N = n$ , the number of columns in the constraint matrix, and `SB` = the number of superbasic variables.
3. A set of  $(n + m - 1)/80 + 1$  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  $j = 1 : n + m$  as follows, written with `format(80i1)`.

<code>hs(j)</code>	State of the $j$ th variable
0	Nonbasic at lower bound
1	Nonbasic at upper bound
2	Superbasic
3	Basic

If variable  $j$  is nonbasic, it may be *fixed* (lower bound = upper bound), or *free* (infinite bounds), or it may be strictly between its bounds. In such cases, `hs(j) = 0`. (Free variables will almost always be basic.)

4. A set of lines of the form

$$j \quad x_j$$

written with `format(i8, 1p, e24.14)` and terminated by an entry with  $j = 0$ , where  $j$  denotes the  $j$ th variable and  $x_j$  is a real value. The  $j$ th variable is either the  $j$ th column or the  $(j - n)$ th slack, if  $j > n$ . 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. The list also includes nonbasic variables that lie strictly between their bounds.

### 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 labeled M and N on the second line must agree with  $m$  and  $n$  for the problem that has just been defined. The value labeled 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) ≠ 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 its current value (even if it is not equal to one of its bounds).

```
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

## 8.2. 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	<i>Key</i>	<i>Name1</i>	<i>Name2</i>	<i>Value</i>

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, or zero for free variables.

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

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.
2. LL lines are *not* output for nonbasic variables if the corresponding lower bound value is zero.

### 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 ENDDATA 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.

### 8.3. 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	<i>Key</i>	<i>Name</i>	<i>Value</i>

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 variables between their bounds will be output with key LL and their current value.

#### 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. BS causes *Name* to become basic.
3. SB causes *Name* to become superbasic at the specified *Value*.
4. LL or UL cause *Name* to be nonbasic at the specified *Value*.
5. 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*.)
6. 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.
7. (*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.)
8. (*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

```

```

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 2: Format of PUNCH/INSERT files

Figure 3: Format of DUMP/LOAD files

---

## References

- [1] A. R. CONN, *Constrained optimization using a nondifferentiable penalty function*, SIAM J. Numer. Anal., 10 (1973), pp. 760–779.
- [2] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.
- [3] S. K. ELDERSVELD, *Large-scale sequential quadratic programming algorithms*, PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1991.
- [4] R. FLETCHER, *An  $\ell_1$  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.
- [5] R. FOURER, *Solving staircase linear programs by the simplex method. 1: Inversion*, Math. Prog., 23 (1982), pp. 274–313.
- [6] P. E. GILL, W. MURRAY, AND M. A. SAUNDERS, *SNOPT: An SQP algorithm for large-scale constrained optimization*, Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.
- [7] 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.
- [8] ———, *Maintaining LU factors of a general sparse matrix*, Linear Algebra and its Applications, 88/89 (1987), pp. 239–270.
- [9] ———, *A practical anti-cycling procedure for linearly constrained optimization*, Math. Prog., 45 (1989), pp. 437–474.
- [10] ———, *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.
- [11] B. A. MURTAGH AND M. A. SAUNDERS, *Large-scale linearly constrained optimization*, Math. Prog., 14 (1978), pp. 41–72.
- [12] ———, *A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints*, Math. Prog. Study, 16 (1982), pp. 84–117.
- [13] ———, *MINOS 5.4 User's Guide*, Report SOL 83-20R, Department of Operations Research, Stanford University, Stanford, CA, Revised 1995.

## Acknowledgement

We are grateful to Alan Brown of the Numerical Algorithms Group Ltd for helpful comments on the documentation for SNOPT. We also thank Rocky Nelson, Arne Drud and Ulf Ringertz for their feedback while running SNOPT on numerous examples.