User's Guide For Snopt Version 6, 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 MS&E
Stanford University
Stanford, California 94305-4026

December 2002

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://www.scicomp.ucsd.edu/~peg
walter@stanford.edu
http://www.stanford.edu/~walter/
saunders@stanford.edu
http://www.stanford.edu/~saunders/

*Partially supported by National Science Foundation grants DMI-9204208 and DMI-9500668, and Office of Naval Research grant N00014-96-I-0274.

Contents

1. Introduction **3**
- 1.1 Problem types . . . 3
- 1.2 The SNOPT interfaces . . . 3
- 1.3 Files . . . 4
- 1.4 Overview of the package . . . 4
- 1.5 Subroutine snInit . . . 5

2. Description of the SQP method **6**
- 2.1 Constraints and slack variables . . . 6
- 2.2 Major iterations . . . 6
- 2.3 Minor iterations . . . 7
- 2.4 The merit function . . . 7
- 2.5 Treatment of constraint infeasibilities . . . 8

3. The `snOptA` interface **9**
- 3.1 Subroutines used by `snOptA` . . . 9
- 3.2 Getting Started . . . 10
- 3.3 Exploiting problem structure . . . 12
- 3.4 Subroutine `snOptA` . . . 14
- 3.5 Subroutine `snJac` . . . 20
- 3.6 Subroutine `usrfun` . . . 22
- 3.7 Specification of `usrfun` . . . 23
- 3.8 Example . . . 25
- 3.9 Subroutine `snMemA` . . . 28
- 3.10 Specification of `snMemA` . . . 28
- 3.11 Using `snMemA` to estimate storage . . . 28

4. The `snOptB` interface **31**
- 4.1 Subroutines used by `snOptB` . . . 31
- 4.2 Identifying structure in the objective and constraints . . . 31
- 4.3 Problem dimensions . . . 33
- 4.4 Subroutine `snOptB` . . . 34
- 4.5 User-supplied routines required by `snOptB` . . . 39
- 4.6 Subroutine funobj . . . 41
- 4.7 Subroutine funcon . . . 43
- 4.8 Constant Jacobian elements . . . 44
- 4.9 Example . . . 44
- 4.10 Subroutine `snMemB` . . . 47
- 4.11 Specification of `snMemB` . . . 47
- 4.12 Using `snMemB` to estimate storage . . . 48

5. The `snOptC` interface **50**
- 5.1 Subroutine snOptC . . . 50
- 5.2 Subroutine usrfun . . . 50

6. The `npOpt` interface **54**
6.1 Subroutines used by `npOpt` 55
6.2 Subroutine `npOpt` 55
6.3 User-supplied subroutines for `npOpt` 59
6.4 Subroutine funobj 59
6.5 Subroutine funcon 60
6.6 Constant Jacobian elements 61

7. Optional parameters **62**
7.1 The SPECS file 62
7.2 SPECS file checklist and defaults 62
7.3 Subroutine snSpec 65
7.4 Subroutines snSet, snSeti, snSetr 66
7.5 Subroutines snGetc, snGeti, snGetr 67
7.6 Description of the optional parameters 68

8. Output **85**
8.1 The PRINT file 85
8.2 The major iteration log 85
8.3 The minor iteration log 88
8.4 Basis factorization statistics 89
8.5 Crash statistics 91
8.6 EXIT conditions 92
8.7 Solution output 97
8.8 The SOLUTION file 99
8.9 The SUMMARY file 99

9. BASIS files **102**
9.1 NEW and OLD BASIS files 102
9.2 PUNCH and INSERT files 104
9.3 DUMP and LOAD files 105

1. Introduction

The SNOPT package 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 is designed to solve a *nonlinear programming problem*, which is formally stated in the form

$$\text{(SparseNP)} \qquad \underset{x}{\text{minimize}} \quad f_0(x) \qquad \text{subject to} \quad l \le \begin{pmatrix} x \\ f(x) \\ A_L x \end{pmatrix} \le u,$$

where l and u are constant lower and upper bounds, f_0 is a smooth scalar objective function, A_L is a sparse matrix, and $f(x)$ is a vector of smooth nonlinear constraint functions $\{f_i(x)\}$. (The optional parameter `maximize` may be used to specify a problem in which f_0 is maximized instead of minimized.)

Ideally, the first derivatives (gradients) of f_0 and f_i should be known and coded by the user. If only some gradients are known, SNOPT will estimate the missing ones with finite differences.

Note that upper and lower bounds are specified for all variables and constraints. This form allows full generality in specifying various types of constraint. Special values are used to indicate absent bounds ($l_j = -\infty$ or $u_j = +\infty$ for appropriate j). Free variables and free constraints ("free rows") are ones that have both bounds infinite. Fixed variables and equality constraints have $l_j = u_j$.

Routines in the SNOPT package 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-threaded environment. They may also be called recursively.

1.1. Problem types

If the nonlinear functions are absent, the problem is a *linear program* (LP) and SNOPT applies the primal simplex method [2]. Sparse basis factors are maintained by LUSOL [8] as in MINOS [14].

If only the objective is nonlinear, the problem is *linearly constrained* (LC) and tends to solve more easily than the general case with nonlinear constraints (NC). For both cases, SNOPT applies a sparse sequential quadratic programming (SQP) method [6], using limited-memory quasi-Newton approximations to the Hessian of the Lagrangian. The merit function for steplength control is an augmented Lagrangian, as in the dense SQP solver NPSOL [7, 10].

In general, SNOPT requires less matrix computation than NPSOL and fewer evaluations of the functions than the nonlinear algorithms in MINOS [12, 13]. It is suitable for nonlinear problems with thousands of constraints and variables, but not thousands of degrees of freedom. (Thus, for large problems there should be many constraints and bounds, and many of them should be active at a solution.)

1.2. The SNOPT interfaces

The SNOPT distribution contains several user *interfaces* or *wrappers* that allow different formats for the specification of the problem. For efficiency reasons, the underlying solver

routines require the problem to be arranged so that the nonlinear variables and constraints occur before linear variables and constraints (for a definition of these terms, see Section 2). This ordering is imposed on the problem format used by the conventional interface `snOptB`. Pre-ordering the data by hand is time-consuming, error prone and difficult to modify. New users are therefore recommended to use the `snOptA` interface, which *automatically* orders the constraints and variables and hence does not impose these restrictions on the problem format. The `snOptA` interface may be used in conjunction with the SnadiOpt package to provide automatic differentiation of the problem functions.

Two other interfaces are available: `snOptC` allows the user to call `snOptB` with the objective and constraints computed in the same routine, and `npOpt` is an interface that accepts problem data written in the same format as the dense SQP code NPSOL.

A complete list of SNOPT interfaces follows:

`snOptA` (§3) is recommended for new users of the SNOPT package. The variables and constraints may be coded in any order.

`snOptB` (§4) is the conventional interface used by versions of SNOPT prior to Version 6. The data must be ordered so that nonlinear constraints and variables come first.

`snOptC` (§5) is the same as `snOptB` except that the user may provides one routine to compute both constraint and objective functions.

`npOpt` (§6) is an interface that accepts the same problem format as NPSOL. It is only appropriate for small dense problems (as is NPSOL!).

1.3. Files

Every SNOPT interface 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 the SNOPT interfaces, see §9.

1.4. Overview of the package

The usual way to use the optimization package SNOPT is via a sequence of subroutine calls. For example, the interface `snOptA` may be invoked via the statements:

```
call snInit( iPrint, iSumm, ... )
call snoptA( Start, neF, n, ... )
```

The routine `snInit` must be called once before any other SNOPT routine is called. It defines the PRINT and SUMMARY files, prints a title on both files, and sets all user-defined optional parameters to be undefined. (Each SNOPT interface will later check the options and set undefined ones to default values.)

If you wish to change the values of one or more of the `snOptA` optional parameters, you may call one or more of the option-setting routines (see §§7.3–7.4). In this case a typical invocation would be of the form:

```
call snInit( iPrint, iSumm, ... )
call snSpec( ... )
call snSet ( ... )
call snOptA( Start, neF, n, ... )
```

where `snSpec` reads a file of optional parameter definitions and `snSet` defines a individual option.

1.5. Subroutine snInit

Subroutine `snInit` must be called before any other SNOPT routine. It defines the PRINT and SUMMARY files, prints a title on both files, and sets all user options to be undefined. (Each SNOPT interface 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, lencw, leniw, lenrw, iw(leniw)
 character*8
&     cw(lencw)
 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` $\leq$ 0, there will be no SUMMARY file output.

`cw(lencw), iw(leniw), rw(lenrw)` must be the same arrays that are passed to other routines in the SNOPT package. 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.

---

2. Description of the SQP method

Here we briefly describe the main features of the SQP algorithm used in the SNOPT package and introduce some terminology used in the description of the subroutine and its arguments. The SQP algorithm is fully described by Gill, Murray and Saunders [6].

2.1. Constraints and slack variables

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

$$\begin{array}{ll} \underset{x,s}{\text{minimize}} & f_0(x) \\ \text{subject to} & \begin{pmatrix} f(x) \\ A_L x \end{pmatrix} - s = 0, \quad l \le \begin{pmatrix} x \\ s \end{pmatrix} \le u. \end{array}$$

The linear and nonlinear general constraints become equalities of the form $f(x) - s_N = 0$ and $A_L x - s_L = 0$, 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 $A_L x - s_L = 0$ 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 elements are the first derivatives of $f(x)$ evaluated at x_k. The QP constraints therefore comprise the m linear constraints

$$\begin{array}{rrrcl} f'(x_k)x & -s_N & & = & -f(x_k) + f'(x_k)x_k, \\ A_L x & & -s_L & = & 0, \end{array}$$

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

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

then the QP subproblem can be written as

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

where $q(x)$ is a quadratic approximation to a modified Lagrangian function [6].

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* π. Similarly, each variable in (x, s) has an associated *reduced gradient* d_j. The reduced gradients for the variables x are the quantities $g - A^T\pi$, where g is the gradient of the QP objective, and the reduced gradients for the slacks are the dual variables π. The QP subproblem is optimal if $d_j \geq 0$ for all nonbasic variables at their lower bounds, $d_j \leq 0$ for all nonbasic variables at their upper bounds, and $d_j = 0$ for 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 §7.6).

2.4. The merit function

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

$$\mathcal{M}(x, s, \pi) = f_0(x) - \pi^T\big(f(x) - s_N\big) + \tfrac{1}{2}\big(f(x) - s_N\big)^T D\big(f(x) - s_N\big), \qquad (2.2)$$

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

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

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

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

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

If the linear constraints are infeasible ($v \ne 0$ or $w \ne 0$), 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 (NPsparse) 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 π for the nonlinear constraints become large), SNOPT enters "elastic" mode and solves the problem

$$\text{NP}(\gamma) \qquad \underset{x,v,w}{\text{minimize}} \quad f_0(x) + \gamma e^T(v+w) \qquad \text{subject to} \quad l \le \begin{pmatrix} x \\ f(x) - v + w \\ A_L x \end{pmatrix} \le u, \quad v \ge 0, \quad w \ge 0,$$

where γ is a nonnegative parameter (the *elastic weight*), and $f_0(x) + \gamma e^T(v+w)$ is called a *composite objective*. If γ is sufficiently large, this is equivalent to minimizing the sum of the nonlinear constraint violations subject to the linear constraints and bounds. A similar ℓ_1 formulation of (NPsparse) is fundamental to the Sℓ_1QP algorithm of Fletcher [4]. See also Conn [1].

3. The snOptA interface

The wrapper **snOptA** accepts a format that allows the constraints and variables to defined in any order, irrespective of whether or not they occur nonlinearly in the objective or constraints. **snOptA** is designed to solve an optimization problem stated in the form

$$\text{(NPA)} \qquad \underset{x}{\text{minimize}} \quad F_{obj}(x) \qquad \text{subject to} \quad l_x \le x \le u_x, \qquad l_F \le F(x) \le u_F,$$

where l and u are constant lower and upper bounds, $F(x)$ is a vector of smooth linear and nonlinear constraint functions $\{F_i(x)\}$, and $F_{obj}(x)$ is the component of F to be minimized. (The optional parameter **maximize** may be used to specify a problem in which $F_{obj}(x)$ is maximized instead of minimized.) The **snOptA** interface reorders the variables and constraints so that the problem is in the form (SparseNP).

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

Note that upper and lower bounds are specified for all variables and functions. This form allows full generality in specifying various types of constraint. Special values are used to indicate absent bounds ($l_j = -\infty$ or $u_j = +\infty$ for appropriate j). Free variables and free constraints ("free rows") are ones that have both bounds infinite. Fixed variables and equality constraints have $l_j = u_j$.

In general, the components of F are *structured* in the sense that they are formed from sums of linear and nonlinear functions. This structure can be exploited by **snOptA** (see Section 3.3).

3.1. Subroutines used by snOptA

snOptA is accessed via the following routines:

snInit (§1.5) Must be called before any other **snOptA** routines.

snSpec (§7.3) May be called to input a SPECS file (a list of run-time options).

snSet, snSeti, snSetr (§7.4) May be called to specify a single option.

snGetc, snGeti, snGetr (§7.5) May be called to obtain an option's current value.

snLog, snLog2 are called every major and minor iteration to print details of the progress of the optimization.

snJac (§3.5) May be called to find the coordinate stucture of the Jacobian.

usrfun (§3.6) Supplied by the user and called by **snOptA** to define the functions $F_i(x)$.

snOptA (§3) The main solver.

snMemA (§3.9) 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 routine **usrfun** has a fixed parameter list but may have any convenient name. It is passed to **snOptA** as a parameter.

3.2. Getting Started

Consider the following simple nonlinear optimization problem with two variables $x = (x_1, x_2)$ and three inequality constraints:

$$\begin{array}{ll} \underset{x_1,x_2}{\text{minimize}} & x_2 \\ \text{subject to} & x_1^2 + 4x_2^2 \le 4, \\ & (x_1 - 2)^2 + x_2^2 \le 5, \\ & x_1 \ge 0, \end{array} \tag{3.1}$$

If this problem is written to conform to the format of Problem (NPA), we have $l_x \le x \le u_x$, with

$$l_x = \begin{pmatrix} 0 \\ -\infty \end{pmatrix} \le \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} +\infty \\ +\infty \end{pmatrix} = u_x,$$

and $l_F \le F(x) \le u_F$, where

$$l_F = \begin{pmatrix} -\infty \\ -\infty \\ -\infty \end{pmatrix} \le \begin{pmatrix} x_2 \\ x_1^2 + 4x_2^2 \\ (x_1 - 2)^2 + x_2^2 \end{pmatrix} \le \begin{pmatrix} +\infty \\ 4 \\ 5 \end{pmatrix} = u_F.$$

The function $F(x)$ has three components $F_1(x) = x_2$, $F_2(x) = x_1^2 + 4x_2^2$, and $F_3(x) = (x_1 - 2)^2 + x_2^2$. Since the function corresponding to the first row of F is to be minimized, we define the objective row as `ObjRow` = 1. In this case, the objective function is the first row of F, but it can be *any* row, provided that its index is recorded in `ObjRow`.

Note that the upper and lower bounds on the objective must be chosen large enough to make it a "free" row of F.

The Jacobian matrix $F'(x)$ of first partial derivatives of F will be denoted by $\{G_{ij}(x)\}$, where $G_{ij}(x) = \partial F_i(x)/\partial x_j$. For our simple two-dimensional problem we have

$$F(x) = \begin{pmatrix} x_2 \\ x_1^2 + 4x_2^2 \\ (x_1 - 2)^2 + x_2^2 \end{pmatrix} \quad \text{and} \quad G(x) = \begin{pmatrix} 0 & 1 \\ 2x_1 & 8x_2 \\ 2(x_1 - 2) & 2x_2 \end{pmatrix}.$$

The user must provide `snOptA` the following information about the problem:

1. The upper and lower bounds on x and F, and the index `Objrow` of the objective row.

2. A subroutine `usrfun` that computes F. In the example above, the body of the subroutine `usrfun` must contain the assignments:

```
F(1) =                         x(2)     ! The objective row
F(2) =   x(1)**2         + 4.0*x(2)**2
F(3) =  (x(1) - 2.0)**2 +      x(2)**2
```

   The subroutine `usrfun` should also compute as many elements of G as possible, although the user can choose to let `snOptA` compute them (see the definition of the optional parameter `Derivative option` in §7). `snOptA` requires the user to provide a description of the pattern of nonzeros elements of the Jacobian G. This takes the form of a list of the coordinates $\{(i, j)\}$ of the elements G_{ij}. The *values* of the G_{ij} can be defined in the user-defined subroutine `usrfun` or calculated automatically by `snOptA` (a feature instigated by setting the optional parameter `Derivative option`

to 0). The kth coordinate is defined by the values of the integer arrays `iGfun(k)` and `jGvar(k)` (i.e., if $i =$ `iGfun(k)` and $j =$ `jGvar(k)`, then `G(k)` holds G_{ij}, the (i,j)th element of $F'(x)$.) The coordinates of elements that are zero at every x need not be included.

For example, the list of coordinates for the Jacobian above is:

```
iGfun(1) = 1   ! row co-ordinate of g(1) =  0.0
jGvar(1) = 1   ! col co-ordinate of g(1) (can be omitted)

iGfun(2) = 1   ! row co-ordinate of g(2) =  1.0
jGvar(2) = 2   ! col co-ordinate of g(2)

iGfun(3) = 2   ! row co-ordinate of g(3) =  2.0* x(1)
jGvar(3) = 1   ! col co-ordinate of g(3)

iGfun(4) = 2   ! row co-ordinate of g(4) =  8.0* x(2)
jGvar(4) = 2   ! col co-ordinate of g(4)

iGfun(5) = 3   ! row co-ordinate of g(5) =  2.0*(x(1) - 2.0)
jGvar(5) = 2   ! col co-ordinate of g(5)

iGfun(6) = 3   ! row co-ordinate of g(6) =  2.0* x(2)
jGvar(6) = 3   ! col co-ordinate of g(6)
```

In the example above, the body of subroutine `usrfun` would contain the following assignments:

```
F(1) =                         x(2)     ! The objective row
F(2) =   x(1)**2        + 4.0*x(2)**2
F(3) =  (x(1) - 2.0)**2 +      x(2)**2
G(1) =  0.0                             ! G(1,1)  (can be omitted)
G(2) =  1.0                             ! G(1,2)
G(3) =  2.0* x(1)                       ! G(2,1)
G(4) =                      8.0*x(2)    ! G(2,2)
G(5) =  2.0*(x(1) - 2.0)                ! G(3,1)
G(6) =                      2.0*x(2)    ! G(3,2)
```

The elements of G can be stored in any order, (e.g., by rows or by columns). In the example above, they have been provided by rows, i.e., in the order of row 1 (elements $(1,1)$ and $(1,2)$); row 2 (elements $(2,1)$ and $(2,2)$); and finally row 3 (elements $(3,1)$ and $(3,2)$).

If the *values* of the Jacobian elements are calculated in the user-defined subroutine `usrfun`. It is crucial that these elements be defined in the same order as the list of coordinates $\{(i,j)\}$.

Note that the arrays `iGfun(k)` and `jGvar(k)` must be provided regardless of whether or not `snOptA` is used to approximate derivatives automatically. However, if the derivatives are approximated, `iGfun(k)` and `jGvar(k)` may be computed automatically by invoking the routine `snJac`.

3.3. Exploiting problem structure

In many cases, each component of the vector $F(x)$ is the sum of linear and nonlinear functions. `snOptA` allows these terms to be specified separately, so that the linear part is defined just once on input using the input arguments `iAfun`, `jAvar` and `A`. Only the nonlinear part is recomputed at each new iterate.

Suppose that the ith component of $F(x)$ is of the form

$$F_i(x) = f_i(x) + \sum_{j=1}^{n} A_{ij} x_j,$$

where $f_i(x)$ is a nonlinear function (possibly zero) and the elements A_{ij} are constant. The $\mathtt{nF} \times \mathtt{n}$ Jacobian of $F(x)$ is the sum of two $\mathtt{nF} \times \mathtt{n}$ sparse matrices $G(x)$ and A, i.e., $F'(x) = G(x) + A$, where $G(x) = f'(x)$ and A is the $\mathtt{nF} \times \mathtt{n}$ matrix with elements $\{A_{ij}\}$. The matrices $G(x)$ and A are said to be "non-overlapping" if every element of the Jacobian $F'(x) = G(x) + A$ is either an element of G, or an element of A, but *not both* (i.e., there is no Jacobian entry (i, j) with both $A_{ij} \neq 0$ and $G_{ij} \neq 0$).

For example, the function

$$F(x) = \begin{pmatrix} 3x_1 + e^{x_2}x_4 + x_2^2 + 4x_4 - x_3 + x_5 \\ x_2 + x_3^2 + \sin x_4 - 3x_5 \\ x_1 \end{pmatrix}$$

can be written as

$$F(x) = f(x) + Ax = \begin{pmatrix} e^{x_2}x_4 + x_2^2 + 4x_4 \\ x_3^2 + \sin x_4 \\ 0 \end{pmatrix} + \begin{pmatrix} 3x_1 - x_3 + x_5 \\ x_2 - 3x_5 \\ x_1 \end{pmatrix}.$$

in which case

$$F'(x) = \begin{pmatrix} 3 & e^{x_2}x_4 + 2x_2 & -1 & e^{x_2} + 4 & 1 \\ 0 & 1 & 2x_3 & \cos x_4 & -3 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

can be written as $F'(x) = f'(x) + A = G(x) + A$, where

$$G(x) = \begin{pmatrix} 0 & e^{x_2}x_4 + 2x_2 & 0 & e^{x_2} + 4 & 0 \\ 0 & 0 & 2x_3 & \cos x_4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad A = \begin{pmatrix} 3 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 & -3 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The important property of non-overlapping functions is that any variable variable x_j appearing explicitly in $f_i(x)$ does not appear explicitly in $\sum_j A_{ij} x_j$, i.e., $A_{ij} = 0$. (Equivalently, any variable with a nonzero A_{ij} must not appear explicitly in $f_i(x)$.)

The nonzero elements of A and G must be provided in coordinate form. In coordinate form, a nonzero element G_{ij} of a matrix G is stored as the triple (i, j, G_{ij}). The kth coordinate is defined by `iGfun(k)` and `jGvar(k)` (i.e., if i = `iGfun(k)` and j = `jGvar(k)`, then `G(k)` is the ijth element of G.) Any known values of `G(k)` must be assigned by the user in the routine `usrfun`.

The following restrictions apply:

1. If the elements of G cannot be provided because they are either too expensive or too complicated to evaluate, it is still necessary to specify the position of the nonzeros as specified by the arrays `iGfun` and `jGvar`.

2. If an element of G happens to be zero at a given point, it must still be loaded in `usrfun`. (The order of the list of coordinates (triples) is meaningful in `snOptA`.)

The elements of A and G can be stored in any order, (e.g., by rows or by columns). Duplicate entries are ignored.

3.4. Subroutine snOptA

Problem (NPA) is solved by a call to subroutine `snOptA`, 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 routine `usrfun`.

```
      subroutine snOptA
     &   ( Start, nF, n, nxname, nFname,
     &     ObjAdd, ObjRow, Prob, usrfun, snLog,
     &     iAfun, jAvar, lenA, neA, A,
     &     iGfun, jGvar, lenG, neG,
     &     xlow, xupp, xnames, Flow, Fupp, Fnames,
     &     x, xstate, xmul, F, Fstate, Fmul,
     &     inform, mincw, miniw, minrw,
     &     nS, nInf, sInf,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )
      external
     &     snLog, usrfun
      integer
     &     inform, lenA, lencu, lencw, lenG, leniu, leniw, lenru, lenrw,
     &     mincw, miniw, minrw, n, neA, neG, nF, nFname, nInf, nS,
     &     nxname, ObjRow, Start, iAfun(lenA), iGfun(lenG), iu(leniu),
     &     iw(leniw), jAvar(lenA), jGvar(lenG), xstate(n), Fstate(nF)
      double precision
     &     ObjAdd, sInf, A(lenA), F(nF), Fmul(nF), Flow(nF), Fupp(nF),
     &     ru(lenru), rw(lenrw), x(n), xlow(n), xmul(n), xupp(n)
      character*8
     &     Prob, cu(lencu), cw(lencw), Fnames(nFname), xnames(nxname)
```

On entry:

`Start` is an integer that specifies how a starting basis (and certain other items) are to be obtained.

- `Start = 0` 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.
- `Start = 1` is the same as `start = 0` (Cold start) but is more meaningful when a basis file is given.
- `Start = 2` means that a basis is already defined in `xstate` and `Fstate` (probably from an earlier call).

`nF` is n_F, the number of problem functions, including the objective function (if any) and the linear and nonlinear constraints. Simple upper and lower bounds on x can be defined using the parameters `xLow` and `xUpp` defined below and should not be included in F ($\texttt{nF} > 0$).

This is the dimension of the problem vector F.

Since F includes the objective (if any) and all the general linear and nonlinear constraints, F must include at least one row for the problem to be meaningful.

`n` is n, the number of variables. This is the number of columns of G and A ($\texttt{n} > 0$).

`iAfun(lenA), jAvar(lenA), A(lenA)` define the coordinates (i, j) of the nonzero elements of the linear part A of the function $F(x) = f(x) + Ax$.

In particular, the `neA` triples (`iAfun(k)`, `jAvar(k)`, `A(k)`), define the row and column indices i = `iAfun(k)` and j = `jAvar(k)` of the element A_{ij} = `A(k)`.

The coordinates can define the elements of A in any order.

`lenA` is the dimension of the coordinate arrays `iAfun`, `jAvar` and `A` that hold (i, j, A_{ij}) (`lenA` ≥ 1).

`neA` is the number of nonzeros in A such that $F(x) = f(x) + Ax$ (`neA` ≥ 0).

`iGfun(lenG), jGvar(lenG)` define the coordinates (i, j) of the nonzero elements of the nonlinear part G of the derivatives $J(x) = G(x) + A$ of the function $F(x) = f(x) + Ax$. The actual elements of G are assigned in the subroutine `usrfun`.

The coordinates can define the elements of G in any order. However, subroutine `usrfun` must define the actual elements of `G` in the exactly the same order as defined by the coordinates (`iGfun`, `jGvar`).

`lenG` is the dimension of the coordinate arrays `iGfun` and `jGvar` that define the varying Jacobian elements (i, j, G_{ij}) (`lenG` ≥ 1).

`neG` is the number of nonzero entries in G (`neG` ≥ 0).

`nxname, nFname` give the number of variable and constraint names provided in the character arrays `xname` and `Fname`. If `nxname` $= 1$ and `nFname` $= 1$, *no* variable and constraint names are provided. Generic names will be used in the printed solution. Otherwise, `nxname` and `nFname` must be given the values `nxname` $= n$ and `nFname` = `nF`, and all names must be provided.

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

`ObjRow` says which row of F is to act as the objective function. If there is no such vector, `ObjRow` $= 0$ and `snOptA` will attempt to find a point such that $l_F \leq F(x) \leq u_F$ and $l_x \leq x \leq u_x$ ($0 \leq$ `ObjRow` $\leq$ `nF`).

`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.

`usrfun` is the name of a subroutine that calculates the vector of problem functions $F(x)$ and (optionally) its Jacobian $F'(x)$ for a specified vector x. `usrfun` must be declared `external` in the routine that calls `snOptA`. For a detailed description of `usrfun`, see §3.6.

`snLog` is the name of a routine that prints the details of a major iteration. `snLog` must be declared `external` in the routine that calls `snOptA`.

`xlow(n), xupp(n)` contain the lower and upper bounds l_x and u_x on the variables x.

To specify a non-existent lower bound ($[l_x]_j = -\infty$), set `xlow`$(j) \leq -$`bigbnd`, where `bigbnd` is the `Infinite Bound`, whose default value is 10^{20}. To specify a non-existent upper bound ($[u_x]_j = \infty$), set `xupp`$(j) \geq$ `bigbnd`.

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

`Flow(nF), Fupp(nF)` contain the lower and upper bounds l_F and u_F on the problem functions F.

To specify a non-existent lower bound ($[l_F]_j = -\infty$), set Flow(j) $\le$ −bigbnd, where bigbnd is the Infinite Bound, whose default value is 10^{20}. For a non-existent upper bound ($[u_F]_j = -\infty$), set Fupp(j) $\ge$ bigbnd.

To make the ith constraint an *equality* constraint (say $F_i = \beta$, where $|\beta| <$ bigbnd), set Flow(i) = Fupp(i) = β.

xnames(nxname), Fnames(nFname) sometimes contain 8-character names for the variables and problem functions. If nxname = 1 or nFname = 1, then names are not used. The printed solution will use generic names for the columns and row. If nxname = n and nFname = nF, the elements xnames(i) and Fnames(j) should contain the 8-character names of the jth variable and i row of F.

xstate(n) sometimes contains a set of initial states for each variable x. See the following discussion of x.

x(n) contains a set of initial values for x.

1. If Start = 0 (cold start) or Start = 1 (basis file), and a BASIS file of some sort is to be input (an OLD BASIS file, INSERT file or LOAD file), then xstate and x need not be set at all.
2. Otherwise, xstate and x 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 xstate(j) = 0, x(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 xstate(j) = 4 and x(j) = xlow(j) or xstate(j) = 5 and x(j) = xupp(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 xstate(j) = 0, 1, 2, 3, 4, 5 have the following meaning:

xstate(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 xstate(j) = 2 are made superbasic. Other entries not selected for the basis are made nonbasic at the value x(j) if xlow(j) $\le$ x(j) $\le$ xupp(j), or at the value xlow(j) or xupp(j) closest to x(j). See the description of xstate below (on exit).
4. For Warm starts, xstate(1: n) must be 0, 1, 2 or 3 (probably from some previous call).

F(nF) sometimes contains a set of initial values for the problem functions F. See the following discussion of Fstate.

Fstate(nF) sometimes contains a set of initial states for the problem functions F.

1. If Start = 0 (cold start) or Start = 1 (basis file), and a BASIS file of some sort is to be input (an OLD BASIS file, INSERT file or LOAD file), then Fstate and F need not be set at all.

2. Otherwise, `Fstate` and `F` 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 `Fstate`$(i) = 0$, `F`$(i) = 0.0$ for all $i = 1$: `nF`. All rows will be eligible for the initial basis.

   Less trivially, to say that the optimal value of row i will probably be equal to one of its bounds, set `Fstate`$(i) = 4$ and `F`$(i) =$ `Flow`(i) or `Fstate`$(i) = 5$ and `F`$(i) =$ `Fupp`(i) 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 `xstate`$(i) = 0, 1, 2, 3, 4, 5$ have the following meaning:

`Fstate`(i)	State of row i during CRASH
$\{0, 1, 3\}$	Eligible for the basis. 3 is given preference
$\{2, 4, 5\}$	Ignored

   After CRASH, rows for which `Fstate`$(i) = 2$ are made superbasic. Other entries not selected for the basis are made nonbasic at the value `F`(i) if `Flow`$(i) \le$ `F`$(i) \le$ `Fupp`(i), or at the value `Flow`(i) or `Fupp`(i) closest to `F`(i). See the description of `Fstate` below (on exit).

4. For Warm starts, `Fstate`(1: `nF`) must be 0, 1, 2 or 3 (probably from some previous call).

`Fmul(nF)` contains an estimate of λ, the vector of Lagrange multipliers (shadow prices) for the constraints $l_F \le F(x) \le u_F$. All `nF` components must be defined. If nothing is known about λ, set `Fmul`$(i) = 0.0$, $i = 1$: `nF`.

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

`cu(lencu), iu(leniu), ru(lenru)` are 8-character, integer and real arrays of user workspace. They may be used to pass data or workspace to your function routine `usrfun` (which has the same parameters). They are not touched by `snOptA`.

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 `cw`, `iw`, `rw` arrays if `usrfun` needs to access `snOptA`'s workspace.

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

`lencw`, `leniw`, `lenrw` must all be at least 500. If variable and function names are specified in `xnames` and `Fnames`, then `lencw` must be at least 500 + `n` + `nF`, otherwise `lencw` = 500 is appropriate. Arguments `leniw` and `lenrw` should be as large as possible because it is uncertain how much storage will be needed for the basis factors. As an estimate, `leniw` should be about $100(m + n)$ or larger, and `lenrw` should be about $200(m + n)$ or larger.

Appropriate values may be obtained from a call to the routine `snMemA` (see Section 3.9, or from a preliminary run of `snOptA` with `lencw` = `leniw` = `lenrw` = 500. If the value of `Print level` is positive, the required amounts of workspace are printed before `snOptA` terminates with `inform` = 42, 43 or 44. The values are returned in `mincw`, `miniw` and `minrw`.

On exit:

`xstate(n)` gives the final state of the variables. The elements of `xstate` have the following meaning:

`xstate`(j)	State of variable j	Usual value of `x`(j)
0	nonbasic	`xlow`(j)
1	nonbasic	`xupp`(j)
2	superbasic	Between `xlow`(j) and `xupp`(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 `x`(j) lies strictly between its bounds.

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

`x(n)` contains the final values of the variables x.

`xmul(ne)` is the vector of dual variables for the simple bound constraints $l_x \le x \le u_x$.

`F(nF)` is the final value of the vector F of problem functions.

`Fmul(nF)` is the vector of dual variables (Lagrange multipliers) for the general constraints $l_F \le F(x) \le u_F$.

`inform` reports the result of the call to `snOptA`. Here is a summary of possible values (for a detailed description, see §8.6):

- 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 Termination has been requested by the user.
- 7 `usrfun` seems to be giving incorrect objective derivatives.
- 8 `usrfun` seems to be giving incorrect constraint derivatives.
- 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.
- 11 The user signaled undefined functions, but no recovery was possible.
- 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.

40 Some input arguments have invalid values.

41 The work arrays each must have at least 500 elements.

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 `snOptA` 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. `snOptA` 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. `snOptA` 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, `x` 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 $A_L x$ lying outside their upper or lower bounds. The nonlinear constraints are not evaluated.

Otherwise, `x` 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.

3.5. Subroutine snJac

If you are not providing derivatives, you can call the routine snJac to estimate the input arrays iAfun, jAvar, A, iGfun and jGvar that define the pattern of nonzeros. A typical sequence of calls would be as follows:

```
call snInit( iPrint, iSumm,  ... )
call snJac ( inform, iPrint, ... )
call snSet ( 'Derivative option = 0', ... )
call snOptA( Start, nF, n,  ... )
```

Note that the optional parameter Derivative option = 0 must be used to force snOptA to compute the derivatives.

The routine snJac automatically determines the sparsity pattern for the Jacobian and identifies the constant elements automatically. To make this determination, snJac approximates $F'(x)$ at three random perturbations of a user-supplied initial point x. If an element of the approximate Jacobian is the same at all three points, then it is taken to be constant. If it is zero, it is taken to be identically zero. The random points are not chosen close together, so the heuristic will correctly classify the Jacobian elements in the vast majority of cases.

It is possible to fool this heuristic. In particular, snJac may not work correctly on functions for which the sparsity pattern or linearity pattern in a (relatively large) region around x is not representative of the sparsity or linearity pattern of the function as a whole. Computing a sparsity pattern for such a function would require significant additional user intervention.

It is also worth bearing in mind that a knowledgeable user can often do better than snJac at identifying the constant Jacobian elements. For example, snJac will classify a constant element in a row and column containing other nonlinear elements as nonlinear. (This is because snJac "removes" linear variables from the calculation of f by setting them to zero before calling usrfun.) A knowledgeable user would be able to define this type of element as linear and physically remove it from the calculation of F.

Note that most machines use double precision declarations as shown, but some machines use real. The same applies to the user routine usrfun.

```
      subroutine snJac
     &   ( inform, iPrint, iSumm, nF, n, usrfun,
     &     iAfun, jAvar, lenA, neA, A,
     &     iGfun, jGvar, lenG, neG,
     &     x, xlow, xupp, mincw, miniw, minrw,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )

      external
     &     usrfun
      integer
     &     inform, iPrint, iSumm, nF, n, neA, lenA, neG, lenG, mincw,
     &     miniw, minrw, lencu, lencw, leniu, leniw, lenru, lenrw,
     &     iAfun(lenA), jAvar(lenA), iGfun(lenG), jGvar(lenG),
     &     iu(leniu), iw(leniw)
      double precision
     &     A(lenA), x(n), xlow(n), xupp(n), ru(lenru), rw(lenrw)
      character*8
```

```
     &     cu(lencu), cw(lencw)
```

On entry:

Most arguments are identical to those of `snOptA`.

`lenA` is the dimension of the coordinate arrays `iAfun`, `jAvar` and `A` that hold the coordinates (i, j, A_{ij}). ($\texttt{lenA} \geq 1$).

`lenA` should be an *overestimate* of the number of elements in the linear part of the Jacobian. The value $\texttt{lenA} = \texttt{nF} \times \texttt{n}$ is an upper bound on the length of `A`.

`lenG` is the dimension of the coordinate arrays `iGfun` and `jGvar` that define the nonlinear Jacobian elements (i, j, G_{ij}). ($\texttt{lenG} \geq 1$).

`lenG` should be an overestimate of the number of elements in the nonlinear part of the Jacobian. The value $\texttt{lenG} = \texttt{nF} \times \texttt{n}$ is an upper bound on the length of `iGfun` and `jGvar`.

On exit:

`iAfun(lenA), jAvar(lenA), A(lenA)` define the coordinates (i, j) of the nonzero elements of the linear part A of the function $F(x) = f(x) + Ax$.

The `neA` triples (`iAfun(k)`, `jAvar(k)`, `A(k)`), define the row and column indices $i = \texttt{iAfun(k)}$ and $j = \texttt{jAvar(k)}$ of the element $A_{ij} = \texttt{A(k)}$.

`neA` is the number of nonzeros in A such that $F(x) = f(x) + Ax$. ($\texttt{neA} \geq 0$).

`iGfun(lenG), jGvar(lenG)` define the coordinates (i, j) of the nonzero elements of the nonlinear part G of the derivatives $F'(x) = f'(x) + A = G(x) + A$ of the function $F(x) = f(x) + Ax$. The actual elements of G are assigned in the subroutine `usrfun`.

`neG` is the number of nonzero entries in G. ($\texttt{neG} \geq 0$).

`inform` reports the result of the call to `snJac`.

- 0 Satisfactory coordinates were found.
- 6 User requested termination by returning $\texttt{mode} < -1$ from `usrfun`.
- 11 User indicates that the functions are undefined at the initial point.
- 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 build the arrays (i, j, A_{ij}) and (i, j, G_{ij}). If `snJac` 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. `snJac` 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. `snJac` may be called again with `leniw` or `lenrw` suitably larger than `miniw` or `minrw`.

3.6. Subroutine `usrfun`

The user must provide a subroutine that defines the values and derivatives of the nonlinear portion $f(x)$ of the problem functions $F(x) = f(x) + Ax$. This subroutine is passed to `snOptA` as the external parameter `usrfun`. (A dummy subroutine must be provided even if $f \equiv 0$ and all functions are linear.)

In general, this subroutine should return all function and derivative values on every entry except perhaps the last. This provides maximum reliability and corresponds to the default setting, `Derivative option = 1`.

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

As a compromise, `snOptA` allows you to code *as many derivatives as you like*. This option is implemented as follows. Just before the function routine is called, each element of the derivative 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 derivatives.
2. If the derivatives are expensive to compute, specify `Nonderivative linesearch` and use the value of the input parameter `needG` to avoid computing them on certain entries. (There is no need to compute derivatives if `needG` $= 0$ on entry to `usrfun`.)
3. If not all derivatives are known, you must specify `Derivative option` $= 0$. You should still compute as many derivatives as you can. (It often happens that some of them are constant or even zero.)
4. Again, if the known derivatives are expensive, don't compute them if `needG` $= 0$ on entry to `usrfun`.
5. Use the input parameter `Status` to test for special actions on the first or last entries.
6. While the function routine are being developed, use the `Verify` option to check the computation of derivatives 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 on the constraint boundaries.
8. Set `Status` $= -1$ if the functions are undefined. The line search will shorten the step and try again.
9. Set `Status` < -1 if you want `snOptA` to stop.

3.7. Specification of usrfun

This subroutine must compute the nonlinear portion $f(x)$ and (optionally) its Jacobian matrix of first derivatives $G(x) = f'(x)$. The jth column of $G(x)$ is the vector $\partial f/\partial x_j$.

The elements of $G(x)$ are stored in the array `G(1:lenG)` in the order specified by `snOptA`'s input arrays `iGFun` and `jGvar` (see Section 3.8).

```
      subroutine usrfun
     &   ( Status, n, x,
     &     needf, nF, f,
     &     needG, lenG, G,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     lencu, lenG, leniu, lenru, n, needf, needG, nF, Status,
     &     iu(leniu)
      double precision
     &     f(nF), G(lenG), ru(lenru), x(n)
      character*8
     &     cu(lencu)
```

On entry:

`Status` indicates the first and last calls to `usrfun`.

If `Status` $= 0$, there is nothing special about the current call to `usrfun`.

If `Status` $= 1$, `snOptA` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `Status` ≥ 2, `snOptA` is calling your subroutine for the *last* time. You may wish to perform some additional computation on the final solution.

In general, the last call is made with `Status` $= 2 +$ `inform`, where `inform` indicates the status of the final solution. In particular, if `Status` $= 2$, the current `x` is *optimal*; if `Status` $= 3$, the problem appears to be infeasible; if `Status` $= 4$, the problem appears to be unbounded; and if `Status` $= 5$, the iterations limit was reached. In some cases, the solution may be *nearly* optimal if `Status` $= 11$; 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 (Status .ge. 2) return
```

at the start of the subroutine.

`n` is n, the number of variables, as defined in the call to `snOptA`.

`x(n)` contains the variables x at which the problem functions are to be calculated. *The array `x` must not be altered.*

`needf, f(nF)` concern the calculation of the nonlinear part of the problem functions.

`nF` is the length of the full vector $F(x) = f(x) + Ax$ as defined in the call to `snOptA`.

The value of `needf` indicates if `f` must be assigned during this call of `usrfun`.

- If `needf` $= 0$, `f` is not required and is ignored.

- If `needf` > 0, the components of $f(x)$ corresponding to the nonlinear part of F must be calculated and assigned to `f`.

  If $F_i(x)$ is linear and completely defined by A_i', then the associated value of $f_i(x)$ is ignored and need not be assigned. However, if $F_i(x)$ has a nonlinear portion f_i that happens to be zero at x, then it is still necessary to set $f_i(x) = 0$. If the linear part A_i' of a nonlinear $F_i(x)$ is provided using the `snOptA` arrays (`iAFun`, `jAvar`, `A`), then it must not be computed as part of $f_i(x)$.

To simpify the code, the user can choose to ignore the value of `needf` and compute f on every entry to `usrfun`.

The parameter `needf` also can be ignored if `Derivative linesearch` is selected (the default) and if `Derivative option` $= 1$. In this case, `needf` will always be 1, and `f` will always need to be assigned.

`needG, G(lenG)` concern the calculation of the derivatives of the function f such that $F(x) = f(x) + Ax$.

`lenG` is the length of the coordinate arrays `jGfun` and `iGvar` as defined in the call to `snOptA`.

`needG` indicates if `G` must be assigned during this call `usrfun`.

- If `needG` $= 0$, `G` is not required and is ignored.
- If `needG` > 0, the partial derivatives of $f(x)$ must be calculated and assigned to `G`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snOptA`. 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 `snOptA`'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(lFMul)` onward, where `lFMul` = `iw(329)`. These will be accessible to `usrfun` if `snOptA` 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:

`Status` may be used to indicate that you are unable or unwilling to evaluate the problem function at the current x (e.g., the problem functions may not be defined there).

During the line search, 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 α, if you set `Status` to -1, `snOptA` 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 `Status` to a negative value (other than -1).

`f(nF)` contains the computed functions $f(x)$ (except perhaps if `needf` $= 0$).

`G(neG)` contains the computed derivatives $G(x)$ (except perhaps if `needG` $= 0$).

These derivative elements must be stored in `G` in exactly the same positions as implied by the definitions of `snOptA`'s arrays `iGfun`, `jGvar`. There is no internal check for consistency (except indirectly via the `Verify` option), so great care is essential.

3.8. Example

Here we give the routine `usrfun` for the problem

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

This problem has 4 variables, 2 nonlinear constraints and 1 linear constraint.

$$F'(x) = \begin{pmatrix} 3 + 2(x_1 + x_3 + x_4) & 5 & 2(x_1 + x_3 + x_4) & 2(x_1 + x_3 + x_4) \\ 1 & 0 & 2x_3 & 2x_4 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 4x_3^3 & 4x_4^3 \end{pmatrix}$$

can be written as $F'(x) = f'(x) + A = G(x) + A$, where

$$G(x) = \begin{pmatrix} 3 + 2(x_1 + x_3 + x_4) & 0 & 2(x_1 + x_3 + x_4) & 2(x_1 + x_3 + x_4) \\ 0 & 0 & 2x_3 & 2x_4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 4x_3^3 & 4x_4^3 \end{pmatrix}$$

and

$$A = \begin{pmatrix} 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

This problem has 4 variables, 2 nonlinear constraints and 1 linear constraint. The calling program must assign the values

```
nF     =  4
n      =  4
ObjRow =  1
```

Subroutine `usrfun` works with the nonlinear variables (x_1, x_3, x_4). For illustration, we test `needF` and `needG` to economize on function and gradient evaluations (even though they are cheap here). Note that `No derivative linesearch` would have to be specified, otherwise all entries would have `needG` $= 1$. We also test `State` to print a message on the first and last entries.

```
      subroutine usrfun
     &   ( Status, n, x,
     &     needF, nF, F,
     &     needG, lenG, G,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     lencu, lenG, leniu, lenru, n, needF, needG, nF, Status,
     &     iu(leniu)
      double precision
     &     F(nF), G(lenG), x(n), ru(lenru)
      character*8
     &     cu(lencu)

*     ==================================================================
*     Computes the nonlinear objective and constraint terms for the toy
*     problem featured in the SNOPT users guide.
*     ==================================================================
      integer
     &     neG, Obj, Out
      double precision
     &     sum
*     ------------------------------------------------------------------
      Out = 15
      Obj = 1                  ! Objective row of F

*     --------------------------------------------
*     Print something on the first and last entry.
*     -------------------------------------------
      if (Status .eq. 1) then       ! First
         if (Out .gt. 0) write(Out, '(/a)') ' This is problem  Toy'
      else  if (Status .ge. 2) then ! Last
         if (Out .gt. 0) write(Out, '(/a)') ' Finished problem Toy'
         return
      end if

      sum = x(1) + x(3) + x(4)

      if (needF .gt. 0) then
         F(Obj) =   3.0d0*x(1) + sum**2
         F(2)   =         x(3)**2 +   x(4)**2
*        F(3)   =       4*x(3)    + 2*x(4)     ! Linear constraint omitted
         F(4)   =         x(3)**4 +   x(4)**4
      end if

      neG = 0
      if (needG .gt. 0) then
         neG        =  neG + 1
         G(neG)     =  2.0D0*sum + 3.0d0
*        iGfun(neG) =  Obj
*        jGvar(neG) =  1
         neG        =  neG + 1
         G(neG)     =  2.0D0*sum
*        iGfun(neG) =  Obj
*        jGvar(neG) =  3
         neG        =  neG + 1
```

```
         G(neG)      =  2.0D0*sum
*        iGfun(neG) =  Obj
*        jGvar(neG) =  4
         neG         =  neG + 1
         G(neG)      =  2.0D0*x(3)
*        iGfun(neG) =  2
*        jGvar(neG) =  3
         neG         =  neG + 1
         G(neG)      =  2.0D0*x(4)
*        iGfun(neG) =  2
*        jGvar(neG) =  4
         neG         =  neG + 1
         G(neG)      =  4.0d0*x(3)**3
*        iGfun(neG) =  4
*        jGvar(neG) =  3
         neG         =  neG + 1
         G(neG)      =  4.0d0*x(4)**3
*        iGfun(neG) =  4
*        jGvar(neG) =  4
      end if
      end ! subroutine usrfun
```

3.9. Subroutine snMemA

This routine computes the size of the workspace arrays `cw`, `iw`, `rw` for a given optimization problem. `snMemA` is not strictly needed in f77 because all workspace must be defined explicitly in the driver program at compile time. `snMemA` is included in the SNOPT distribution for users wishing to allocate storage dynamically using f90 or C.

The actual storage required at solve time depends on the values of the optional parameters: `Superbasics limit`, `Quasi-Newton updates`, and `Hessian limited memory` (or `Hessian full`) (see Section 7). If these options have not been set when `snMemA` is called, default values are assumed. If the default values are unsatisfactory, it is strongly recommended that the correct values be set *before* the call to `snMemA`.

3.10. Specification of snMemA

```
      subroutine snMemA
     &   ( nF, n, nxname, nFname, neA, neG,
     &     mincw, miniw, minrw,
     &     cw, lencw, iw, leniw, rw, lenrw )

      implicit
     &     none
      integer
     &     lencw, leniw, lenrw, mincw, miniw, minrw, n, neA, neG, nF,
     &     nFname, nxname, iw(leniw)
      double precision
     &     rw(lenrw)
      character*8
     &     cw(lencw)
```

The arguments `nF`, `n`, `nxname`, `nFname`, `neA`, `neG`, define the problem being solved and are identical to the arguments used in the call to `snOptA` (see Section 3.4). If a sequence of problems are being solved, then `snMemA` may be called once with overestimates of these quantities.

On entry:

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

`lencw`, `leniw` and `lenrw` must be of length at least 500.

On exit:

`mincw, miniw, minrw` estimate how much character, integer and real storage is needed to solve the problem.

3.11. Using snMemA to estimate storage

The first step is to allocate the work arrays used by `snMemA` to hold various constants and pointers. These may be either temporary arrays, say, `tmpcw`, `tmpiw` and `tmprw`, or the `snOptA` arrays `cw`, `iw` and `rw` that are reallocated after the storage limits are known. Here

we illustrate the use of `snMemA` using the same arrays for both `snMemA` and `snOptA`. Note that the `snMemA` arrays are used to store the optional parameters, and so any temporary arrays must be copied into the final `cw`, `iw` and `rw` in order to retain the options.

The `snMemA` work arrays must have length at least 500, so we define

```
ltmpcw = 500
ltmpiw = 500
ltmprw = 500
```

As is the case with all SNOPT routines, `snInit` *must* be called to initialize the optional parameters to their default values.

```
      call snInit
     &   ( iPrint, iSumm, cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

Note that the call to `snInit` installs `ltmpcw`, `ltmpiw` and `ltmprw` as the default internal upper limits on the `snOptA` workspace (see the description of `Total real workspace` in Section 7.6). (They are used to compute the boundaries of any user-defined workspace in `cw`, `iw` or `rw`.)

The next step is to compute an estimate of the storage needed by `snOptA`. The required estimates are `mincw`, `miniw` and `minrw`, which are defined by the call:

```
      call snMemA
     &   ( nF, n, nxname, nFname, neA, neG,
     &     mincw, miniw, minrw,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

The output values of `mincw`, `miniw` and `minrw` may now be used to define the lengths of the `snOptA` work arrays:

```
lencw = mincw
leniw = miniw
lenrw = minrw
```

These values may be used in f90 to allocate the work arrays for this problem.

One last step is needed before calling `snOptA`. The current default upper limits `ltmpcw`, `ltmpiw` and `ltmprw` must be replaced with the estimates `mincw`, `miniw` and `minrw`. This is done using the option setting routine `snSeti` as follows:

```
      iPrt   = 0                ! Suppress print   output
      iSum   = 0                ! Suppress summary output
      call snSeti
     &   ( 'Total character workspace', lencw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
      call snSeti
     &   ( 'Total integer   workspace', leniw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
      call snSeti
     &   ( 'Total real      workspace', lenrw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

An alternative way to reset the default workspace optional parameters is to call `snInit` again with arguments `lencw`, `leniw` and `lenrw`:

```
      call snInit
     &   ( iPrint, iSumm, cw, lencw, iw, leniw, rw, lenrw )
```

However, this will have the twin effects of resetting all options to their default values, and reprinting the SNOPT banner (unless `iPrint` $= 0$ and `iSumm` $= 0$ are set as the default print and summary files).

4. The snOptB interface

snOptB is the "basic" user interface with arguments identical to versions of SNOPT up to 5.3-4. This interface requires the data to be ordered so that nonlinear constraints and variables come first. A typical invocation of snOptB would be:

```
call snInit( iPrint, iSumm, ... )
call snSpec( ... )
call snOptB( Start, m, n, ne, ... )
```

where, as before, snSpec reads a file of optional parameter definitions.

The routine snOpt is identical to snOptB and the call

```
call snOpt ( Start, m, n, ne, ... )
```

provides compatibility with previous versions of SNOPT.

4.1. Subroutines used by snOptB

snOptB is accessed via the following routines:

snInit (§1.5) Must be called before any other snOptB routines.

snSpec (§7.3) May be called to input a SPECS file (a list of run-time options).

snSet, snSeti, snSetr (§7.4) May be called to specify a single option.

snGetc, snGeti, snGetr (§7.5) May be called to obtain an option's current value.

funcon, funobj (§4.5) Supplied by the user and called by snOptB. They define the constraint functions $f(x)$ and objective function $f_0(x)$.

snOptB, snOpt (§4.4) The main solvers. snOpt is identical to snOptB.

snMemB (§4.10) 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 snOptB and snOpt as parameters.

4.2. Identifying structure in the objective and constraints

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

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

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

- 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 $f_0(u, v, z) = (u + v + z)^2$ 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 `snOptB` 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

$$f_0(x') + d^T x,$$

where x' is the first `nnObj` variables, $f_0(x')$ is defined by subroutine `funobj`, and d 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

$$\begin{pmatrix} f(x'') + A_2 y'' \\ A_3 x'' + A_4 y'' \end{pmatrix}, \tag{4.1}$$

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

$$A = \begin{pmatrix} f'(x'') & A_2 \\ A_3 & A_4 \end{pmatrix}, \tag{4.2}$$

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 $f'(x'')$ are input column-wise via the array parameters `Acol`, `indA`, `locA`. (Elements that are identically zero need not be included.)

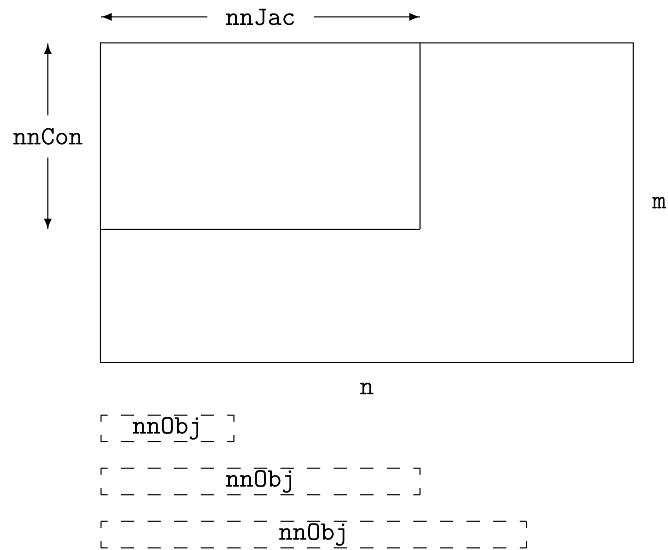
The inequalities $l_1 \le f(x'') + A_2 y'' \le u_1$ and $l_2 \le A_3 x + A_4 y'' \le u_2$ implied by the constraint functions (4.1) are known as the *nonlinear* and *linear* constraints respectively. Together, these two sets of inequalities constitute the *general constraints*.

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''.

4.3. Problem dimensions

The following picture illustrates the problem structure just described:



The dimensions are all input parameters to subroutine `snOptB` (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 (`nnCon` $<$ `iObj` $\leq$ `m`).

The dashed boxes indicate that a nonlinear objective function $f_0(x')$ may involve either a subset or a superset of the variables in the nonlinear constraint functions $f(x'')$, counting from the left. Thus, `nnObj` $\leq$ `nnJac` or vice versa.

Sometimes the objective and constraints really involve *disjoint sets of nonlinear variables*. We then recommend ordering the variables so that `nnObj` $>$ `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 $j = 1$: `nnJac`. It should then set as many remaining gradients as possible—preferably all!

4.4. Subroutine snOptB

Problem (SparseNP) is solved by a call to subroutine **snOptB**, 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,
     &     fgcon, fgobj,
     &     Jcol, indJ, locJ, bl, bu, Names,
     &     hs, x, pi, rc,
     &     inform, mincw, miniw, minrw,
     &     nS, nInf, sInf, Obj,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )
      external
     &     fgcon, fgobj
      integer
     &     inform, iObj, lencu, lencw, leniu, leniw, lenru, lenrw, m,
     &     mincw, miniw, minrw, n, ne, nInf, nName, nnCon, nnJac, nnObj,
     &     nS, hs(n+m), indJ(ne), iu(leniu), iw(leniw), locJ(n+1)
      double precision
     &     Obj, ObjAdd, sInf, Jcol(ne), bl(n+m), bu(n+m), pi(m),
     &     rc(n+m), ru(lenru), rw(lenrw), x(n+m)
      character*(*)
     &     Start
      character*8
     &     Prob, cu(lencu), cw(lencw), Names(nName)
```

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 (4.2).

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 **Acol**, **indA** and **locA** 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 ≥ 0).

nnObj is n'_1, the number of nonlinear objective variables (nnObj ≥ 0).

nnJac is n''_1, the number of nonlinear Jacobian variables. If nnCon $= 0$, nnJac $= 0$. If nnCon > 0, nnJac > 0.

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

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

Prob is an 8-character name for the problem. Prob is used in the printed solution and in some routines that 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 snOptB. For a detailed description of funcon, see §4.7.

funobj is the name of a subroutine that calculates the objective function $f_0(x)$ 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 snOptB. For a detailed description of funobj, see §4.6.

Acol(ne), indA(ne), locA(n+1) define the nonzero elements of the constraint matrix A (4.2), including the Jacobian matrix associated with nonlinear constraints. The nonzeros are stored column-wise. A pair of values (Acol(k),indA(k)) contains a matrix element and its corresponding row index, and the array locA(*) is a set of pointers to the beginning of each column of A within Acol(*) and indA(*). Thus for $j = 1\colon n$, the entries of column j are held in Acol($k\colon l$) and their corresponding row indices are in indA($k\colon l$), where $k =$ locA(j) and $l =$ locA($j+1$) $- 1$,

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

In general, elements in the nonlinear part of Acol(*) (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 Derivativelevel $= 2$ or 3, the nonlinear part of Acol(*) 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 Acol(*).

1. It is *essential* that locA$(1) = 1$ and locA$(n+1) =$ ne $+ 1$.
2. The Jacobian $f'(x)$ forms the top left corner of Acol and indA (see §4.2). If a Jacobian column j ($1 \leq j \leq$ nnJac) contains any entries Acol(k), indA(k) associated with nonlinear constraints ($1 \leq$ indA(k) $\leq$ nnCon), those entries must come before any entries belonging to linear constraints.

3. The row indices `indA(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 `Acol`$(1) = 0.0$, `indA`$(1) = 1$, `locA`$(1) = 1$, and `locA`$(j) = 2$ for $j = 2\colon n+1$. This row is made "free" by setting its bounds to be `bl`$(n+1) = -$`bigbnd` and `bu`$(n+1) =$ `bigbnd`, where `bigbnd` is typically `1.0e+20` (see the next paragraph).

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

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

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

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

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

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

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

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

`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 `hs` and `xs` need not be set at all.
2. Otherwise, `hs`$(1\colon n)$ and `xs`$(1\colon n)$ 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 `hs`$(j) = 0$, `xs`$(j) = 0.0$ for all $j = 1\colon n$. 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 `hs`$(j) = 4$ and `xs`$(j) =$ `bl`(j) or `hs`$(j) = 5$ and `xs`$(j) =$ `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 `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`$(j) \le$ `xs`$(j) \le$ `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`$(1\colon n+m)$ must be 0, 1, 2 or 3 (probably from some previous call) and all of `xs`$(1\colon n+m)$ must have values.

`pi` contains an estimate of λ, the vector of Lagrange multipliers (shadow prices) for the nonlinear constraints. The first `nnCon` components must be defined. If nothing is known about λ, set `pi`$(i) = 0.0$, $i = 1\colon$ `nnCon`.

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

`cu(lencu), iu(leniu), ru(lenru)` are 8-character, integer and real arrays of user workspace. 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 `snOptB`.

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 `snOptB`'s workspace.

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

`lencw`, `leniw`, `lenrw` must all be at least 500. In general, `lencw` = 500 is appropriate but `leniw` and `lenrw` should be as large as possible because it is uncertain how much storage will be needed for the basis factors. As an estimate, `leniw` should be about $100(m+n)$ or larger, and `lenrw` should be about $200(m+n)$ or larger.

Appropriate values may be obtained from a preliminary run with `lencw` = `leniw` = `lenrw` = 500. If `Print level` is positive, the required amounts of workspace are printed before `snOptB` terminates with `inform` = 42, 43 or 44. The values are returned in `mincw`, `miniw` and `minrw`.

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 `xs`(j) lies strictly between its bounds.

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

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

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

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

`inform` reports the result of the call to `snOptB`. Here is a summary of possible values (for a detailed description, see §8.6):

- 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 Termination has been requested by the user.
- 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.
- 11 The user signaled undefined functions, but no recovery was possible.
- 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.
- 40 Some input arguments have invalid values.
- 41 The work arrays each must have at least 500 elements.
- 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 `snOptB` 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. `snOptB` 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. `snOptB` 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 $A_L x$ 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.

4.5. User-supplied routines required by snOptB

The user must provide subroutines to define the nonlinear parts of the objective function and nonlinear constraints. They are passed to `snOptB` 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 `snOptB`: 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. `snOptB` 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, `snOptB` 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 `snOptB` to stop.

4.6. Subroutine funobj

This subroutine must calculate the nonlinear objective function $f_0(x)$ and (optionally) its gradient $g(x)$, where x is the current value of the objective variables x'. The jth component of the gradient is $\partial f_0/\partial x_j$.

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

      integer
     &     lencu, leniu, lenru, mode, nnObj, nState, iu(leniu)
      double precision
     &     fObj, gObj(nnObj), ru(lenru), x(nnObj)
      character*8
     &     cu(lencu)
```

On entry:

`mode` indicates whether `fObj` or `gObj` or both must be assigned during the present call of `funobj` ($0 \le$ `mode` ≤ 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, `snOptB` 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 $f_0(x)$ ($0 <$ `nnObj` $\le n$). 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, `snOptB` 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` ≥ 2, `snOptB` 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 `snOptB`. 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 `snOptB`'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(lxMul)` onward, where `lxMul` = `iw(316)`. These will be accessible to both `funobj` and `funcon` if `snOptB` 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 α, if you set `mode` to -1, `snOptB` 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 $f_0(x)$ (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 $\partial f_0/\partial x_j$ (except perhaps if `mode` = 0).

4.7. Subroutine funcon

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

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

Recall that $f'(x)$ is the *top left corner* of a larger matrix A that is stored column-wise in `snOptB`'s input arrays `Acol`, `indA`, `locA` (see (4.2) and §§4.3,4.4). Jacobian elements must be stored in `gCon` in the same order as the corresponding parts of `Acol`, `indA`, `locA`.

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
     &     lencu, leniu, lenru, mode, neJac, nnCon, nnJac, nState,
     &     iu(leniu)
      double precision
     &     fCon(nnCon), ru(lenru), x(nnJac)
      character*8
     &     cu(lencu)

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

On entry:

`mode` indicates whether `fCon` or `gCon` or both must be assigned during the present call of `funcon` ($0 \le$ `mode` ≤ 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, `snOptB` 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 (`nnCon` > 0). These must be the first `nnCon` constraints in the problem.

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

`neJac` is the number of nonzero elements in `gCon`. If `gCon` is stored as a two-dimensional array, then `neJac` = `nnCon` × `nnJac`.

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

`nState` is used as in `funobj`.

`cu(lencu), iu(leniu), ru(lenru)` 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 $f'(x)$ (except perhaps if `mode` $= 0$).

These gradient elements must be stored in `gCon` in exactly the same positions as implied by the definitions of `snOptB`'s arrays `Acol`, `indA`, `locA`. 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`.

4.8. Constant Jacobian elements

If all constraint gradients (Jacobian elements) are known (`Derivative level` = 2 or 3), any *constant* elements may be given to `snOptB` in the array `Acol` if desired. The Jacobian array `gCon` are initialized from the appropriate elements of `Acol`. 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 `gCon(`i`,`j`)*x(`j`)` or `gCon(`l`)*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.

4.9. Example

Here we give the subroutines `funobj` and `funcon` for the example of §4.2, repeated here for convenience:

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

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, lencu, leniu, lenru, iu(leniu)
      double precision
     &     fObj, x(nnObj), gObj(nnObj),  ru(lenru)
      character*8
     &     cu(lencu)

*     ==================================================================
*     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 ! subroutine funobj
```

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 column-wise, 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
     &     lencu, leniu, lenru, mode, neJac, nnCon, nnJac, nState,
     &     iu(leniu)
      double precision
     &     fCon(nnCon), gCon(nnCon,nnJac), ru(lenru), x(nnJac)
      character*8
     &     cu(lencu)
*     ==================================================================
*     Toy NLP problem.
*     ==================================================================
      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 ! subroutine funcon
```

4.10. Subroutine snMemB

This routine computes the size of the workspace arrays cw, iw, rw for a given optimization problem. snMemB is not strictly needed in f77 because all workspace must be defined explicitly in the driver program at compile time. snMemB is included in the SNOPT distribution for users wishing to allocate storage dynamically using f90 or C.

The actual storage required at solve time depends on the values of the optional parameters: Superbasics limit, Quasi-Newton updates, and Hessian limited memory (or Hessian full) (see Section 7). If these options have not been set when snMemB is called, default values are assumed. If the default values are unsatisfactory, it is strongly recommended that the correct values be set *before* the call to snMemB.

4.11. Specification of snMemB

```
      subroutine snMemB
     &   ( m, n, ne, neGcon,
     &     nnCon, nnJac, nnObj,
     &     mincw, miniw, minrw,
     &     cw, lencw, iw, leniw, rw, lenrw )

      implicit
     &     none
      integer
     &     lencw, leniw, lenrw, m, mincw, miniw, minrw, n, ne, neGcon,
     &     nnCon, nnJac, nnObj, iw(leniw)
      double precision
     &     rw(lenrw)
      character*8
     &     cw(lencw)
```

The arguments m, n, ne, nnCon, nnJac and nnObj, define the problem being solved and are identical to the arguments used in the call to snOptB (see Section 4.4). If a sequence of problems are being solved, then snMemB may be called once with overestimates of these quantities.

On entry:

neGcon is the number of nonzeros in the Jacobian gCon (neGcon $\le$ nnCon $*$ nnJac).

cw(lencw), iw(leniw), rw(lenrw) are 8-character, integer and real arrays of workspace for snMemB.

lencw, leniw and lenrw must be of length at least 500.

On exit:

mincw, miniw, minrw estimate how much character, integer and real storage is needed to solve the problem.

4.12. Using snMemB to estimate storage

The first step is to allocate the work arrays used by snMemB to hold various constants and pointers. These may be either temporary arrays, say, tmpcw, tmpiw and tmprw, or the snOptB arrays cw, iw and rw that are reallocated after the storage limits are known. Here we illustrate the use of snMemB using the same arrays for both snMemB and snOptB. Note that the snMemB arrays are used to store the optional parameters, and so any temporary arrays must be copied into the final cw, iw and rw in order to retain the options.

The snMemB work arrays must have length at least 500, so we define

```
ltmpcw = 500
ltmpiw = 500
ltmprw = 500
```

As is the case with all SNOPT routines, snInit *must* be called to initialize the optional parameters to their default values.

```
      call snInit
     &   ( iPrint, iSumm, cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

Note that the call to snInit installs ltmpcw, ltmpiw and ltmprw as the default internal upper limits on the snOptB workspace (see the description of Total real workspace in Section 7.6). (They are used to compute the boundaries of any user-defined workspace in cw, iw or rw.)

The next step is to compute an estimate of the storage needed by snOptB. The required estimates are mincw, miniw and minrw, which are defined by the call:

```
      call snMemB
     &   ( m, n, ne, neGcon,
     &     nnCon, nnJac, nnObj,
     &     mincw, miniw, minrw,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

The output values of mincw, miniw and minrw may now be used to define the lengths of the snOptB work arrays:

```
lencw = mincw
leniw = miniw
lenrw = minrw
```

These values may be used in f90 to allocate the work arrays for this problem.

One last step is needed before calling snOptB. The current default upper limits ltmpcw, ltmpiw and ltmprw must be replaced with the estimates mincw, miniw and minrw. This is done using the option setting routine snSeti as follows:

```
      iPrt   = 0                ! Suppress print   output
      iSum   = 0                ! Suppress summary output
      call snSeti
     &   ( 'Total character workspace', lencw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
      call snSeti
     &   ( 'Total integer   workspace', leniw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
      call snSeti
     &   ( 'Total real      workspace', lenrw, iPrt, iSum, inform,
     &     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

An alternative way to reset the default workspace optional parameters is to call `snInit` again with arguments `lencw`, `leniw` and `lenrw`:

```
      call snInit
     &   ( iPrint, iSumm, cw, lencw, iw, leniw, rw, lenrw )
```

However, this will have the twin effects of resetting all options to their default values, and reprinting the SNOPT banner (unless `iPrint` = 0 and `iSumm` = 0 are set as the default print and summary files).

5. The snOptC interface

The snOptC interface is identical to snOptB except that both objective and constraint functions are provided in a single routine usrfun instead of being computed separately in funobj and funcon. This arrangement may be more convenient when the objective and constraint functions depend on common data that is difficult to share between separate routines.

5.1. Subroutine snOptC

Problem (SparseNP) is solved by a call to subroutine snOptC, 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 snOptC
     &   ( Start, m, n, ne, nName,
     &     nnCon, nnObj, nnJac,
     &     iObj, ObjAdd, Prob,
     &     usrfun,
     &     Acol, indA, locA, bl, bu, Names,
     &     hs, x, pi, rc,
     &     inform, mincw, miniw, minrw,
     &     nS, nInf, sInf, Obj,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )

      implicit
     &     none
      external
     &     usrfun
      integer
     &     inform, iObj, lencu, leniu, lenru, lencw, leniw, lenrw,
     &     mincw, miniw, minrw, m, n, ne, nName, nS, nInf, nnCon,
     &     nnObj, nnJac, indA(ne), hs(n+m), locA(n+1), iu(leniu),
     &     iw(leniw)
      double precision
     &     Obj, ObjAdd, sInf, Acol(ne), bl(n+m), bu(n+m), pi(m),
     &     rc(n+m), ru(lenru), rw(lenrw), x(n+m)
      character*(*)
     &     Start
      character*8
     &     Prob, Names(nName), cu(lencu), cw(lencw)
```

All arguments except usrfun are the same as those for the snOptB interface. The description of usrfun follows.

5.2. Subroutine usrfun

This subroutine must calculate the nonlinear problem functions $f_0(x)$ and $f(x)$, and (optionally) their derivatives $g(x)$ and $f'(x)$.

The objective derivatives are stored in the output array gObj. Constraint derivatives are stored column-wise in the output array gCon. Recall that $f'(x)$ is the *top left corner*

of a larger matrix A that is stored column-wise in snOptB's input arrays Acol, indA, locA (see (4.2) and §§4.3,4.4). Jacobian elements must be stored in gCon in the same order as the corresponding parts of Acol, indA, locA.

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 usrfun(
     &     mode, nnObj, nnCon, nnJac, nnL, neJac,
     &     x, fObj, gObj, fCon, gCon,
     &     nState, cu, lencu, iu, leniu, ru, lenru )
      integer
     &     lencu, leniu, lenru, mode, nnObj, nnCon, nnJac, nnL, neJac,
     &     nState, iu(leniu)
      double precision
     &     fObj, fCon(nnCon), gObj(nnObj), ru(lenru), x(nnL)
      character*8
     &     cu(lencu)

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

On entry:

mode — indicates which combination of fCon, gCon, fObj and gObj, must be assigned during the present call of usrfun. snOptC will call usrfun with mode = 0, 1 or 2. You may test mode to decide what to do:

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

nnObj — is the number of variables involved in $f_0(x)$ ($0 <$ nnObj $\le n$). These must be the first nnObj variables in the problem.

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

nnJac — is the number of variables involved in $f(x)$ ($0 <$ nnJac $\le n$). These must be the first nnJac variables in the problem.

nnL — is max{nnObj, nnJac}, the number of nonlinear variables. These must be the first nnL variables in the problem.

x(nnL) — contains the nonlinear variables x. *The array* x *must not be altered.*

`neJac` is the number of nonzero elements in `gCon`. If `gCon` is stored as a two-dimensional array, then `neJac` = `nnCon` × `nnJac`.

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

`nState` indicates the first and last calls to `usrfun`.

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

If `nState` = 1, `snOptC` 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` ≥ 2, `snOptC` 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 `snOptC`. 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 `snOptC`'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(lxMul)` onward, where `lxMul` = `iw(316)`. These will be accessible to `usrfun` if `snOptC` 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 problem functions at the current x.

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 α, if you set `mode` to -1, `snOptC` 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 $f_0(x)$ (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 $\partial f_0/\partial x_j$ (except perhaps if `mode` $= 0$).

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

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

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

6. The npOpt interface

The `npOpt` interface is designed for the solution of small dense problems. The calling sequences of `npOpt` and its associated user-defined functions are designed to be similar to those of the dense SQP code NPSOL (Gill *et al.* [7]). For the case of `npOpt` it is convenient to restate the problem (NPsparse) with the constraints reordered as:

$$\text{(DenseNP)} \qquad \underset{x}{\text{minimize}} \quad f_0(x) \qquad \text{subject to} \quad l \le \begin{pmatrix} x \\ A_L x \\ f(x) \end{pmatrix} \le u.$$

where l and u are constant lower and upper bounds, f_0 is a smooth scalar objective function, A_L is a matrix, and $f(x)$ is a vector of smooth nonlinear constraint functions $\{f_i(x)\}$. The interface `npOpt` is designed to handle problems for which the objective and constraint gradients are *dense*, i.e., they do not have a significant number of elements that are identically zero.

A typical invocation of `npOpt` would be:

```
call npInit( iPrint, iSumm, ... )
call npSpec( ... )
call npOpt ( n, nclin, ncnln, ... )
```

where `npSpec` reads a file of optional parameter definitions.

Figure 1 illustrates the feasible region for the jth pair of constraints $\ell_j \le r_j(x) \le u_j$. The quantity δ is the optional parameter `feasibility tolerance`, which can be set by the user (see §7). The constraints $\ell_j \le r_j \le u_j$ are considered "satisfied" if r_j lies in Regions 2, 3 or 4, and "inactive" if r_j lies in Region 3. The constraint $r_j \ge \ell_j$ is considered "active" in Region 2, and "violated" in Region 1. Similarly, $r_j \le u_j$ is active in Region 4, and violated in Region 5. For equality constraints ($\ell_j = u_j$), Regions 2 and 4 are the same and Region 3 is empty.

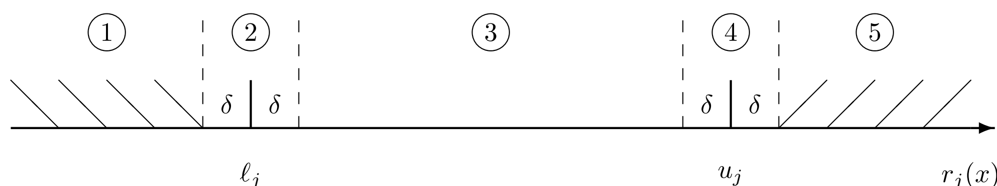


Figure 1: Illustration of the constraints $\ell_j \le r_j(x) \le u_j$. The bounds ℓ_j and u_j are considered "satisfied" if $r_j(x)$ lies in Regions 2, 3 or 4, where δ is the feasibility tolerance. The constraints $r_j(x) \ge \ell_j$ and $r_j(x) \le u_j$ are both considered "inactive" if $r_j(x)$ lies in Region 3.

6.1. Subroutines used by npOpt

npOpt is accessed via the following routines:

npInit (§1.5) Must be called before any other npOpt routines.

npSpec (§7.3) May be called to input a SPECS file (a list of run-time options).

npSet, npSeti, npSetr (§7.4) May be called to specify a single option.

npGetc, npGeti, npGetr (§7.5) May be called to obtain an option's current value.

funcon, funobj (§6.3) Supplied by the user and called by npOpt. They define the constraint functions $f(x)$ and objective function $f_0(x)$.

npOpt (§6) The main solver.

npMem (In distribution file np02lib.f) Computes the size of the workspace arrays iw and 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 npOpt as parameters.

6.2. Subroutine npOpt

In the following specification of npOpt, we define $r(x)$ as the vector of combined constraint functions $r(x) = (\,x \quad A_L x \quad f(x)\,)$, and use nctotl to denote a variable that holds its dimension $\text{nctotl} = \text{n} + \text{nclin} + \text{ncnln}$. Note that most machines use double precision declarations as shown, but some machines use real. The same applies to the user routines funcon and funobj.

```
      subroutine npOpt
     &   ( n, nclin, ncnln, ldA, ldg, ldH,
     &     A, bl, bu, funcon, funobj,
     &     inform, majIts, iState,
     &     fCon, gCon, cMul, fObj, gObj, Hess, x,
     &     iw, leniw, rw, lenrw )
      external
     &     funcon, funobj
      integer
     &     inform, ldA, ldg, ldH, leniw, lenrw, majIts, n, nclin,
     &     ncnln, iState(n+nclin+ncnln), iw(leniw)
      double precision
     &     fObj, A(ldA,*), bl(n+nclin+ncnln), bu(n+nclin+ncnln),
     &     cMul(n+nclin+ncnln), fCon(*), gCon(ldg,*), gObj(n),
     &     Hess(ldH,*), rw(lenrw), x(n)
```

On entry:

n is n, the number of variables in the problem $(\text{n} > 0)$.

nclin is m_L, the number of general linear constraints $(\text{nclin} > 0)$.

ncnln is m_N, the number of nonlinear constraints $(\text{ncnln} > 0)$.

`ldA` is the row dimension of the array `A` (`ldA` ≥ 1, `ldA` $\geq$ `nclin`).

`ldg` is the row dimension of the array `gCon` (`ldg` ≥ 1, `ldg` $\geq$ `ncnln`).

`ldH` is the row dimension of the array `Hess` (`ldH` $\geq$ `n`).

`A` is an array of dimension (`ldA`,k) for some $k \geq$ `n`. It contains the matrix A_L for the linear constraints. If `nclin` $= 0$, `A` is not referenced. (In that case, `A` may be dimensioned `(ldA,1)` with `ldA` $= 1$, or it could be any convenient array.)

`bl(nctotl), bu(nctotl)` contain the lower and upper bounds for $r(x)$ in problem (NP). To specify a non-existent lower bound ($\ell_j = -\infty$), set `bl`$(j) \leq -$`bigbnd`, where `bigbnd` is the `Infinite Bound`, whose default value is 10^{20}. Similarly, a non-existent upper bound ($u_j = \infty$), can be any `bu`$(j) \geq$ `bigbnd`. To specify an *equality* constraint (say $r_j(x) = \beta$), set `bl`$(j) =$ `bu`$(j) = \beta$, where $|\beta| <$ `bigbnd`.

For the data to be meaningful, it is required that `bl`$(j) \leq$ `bu`(j) for all j.

`funcon, funobj` are the names of subroutines that calculate the nonlinear constraint functions $f(x)$, the objective function $f_0(x)$ and (optionally) their gradients for a specified n-vector x. The arguments `funcon` and `funobj` must be declared as `external` in the routine that calls `npOpt`. For a detailed description of `funcon` and `funobj`, see Section 6.3.

`istate(nctotl)` is an integer array that need not be initialized if `npOpt` is called with a `Cold Start` (the default option).

For a `Warm start`, every element of `istate` must be set. If `npOpt` has just been called on a problem with the same dimensions, `istate` already contains valid values. Otherwise, `istate`(j) should indicate whether either of the constraints $r_j(x) \geq \ell_j$ or $r_j(x) \leq u_j$ is expected to be active at a solution of (DenseNP).

The ordering of `istate` is the same as for `bl`, `bu` and $r(x)$, i.e., the first `n` components of `istate` refer to the upper and lower bounds on the variables, the next `nclin` refer to the bounds on $A_L x$, and the last `ncnln` refer to the bounds on $f(x)$. Possible values for `istate`(j) follow.

0 Neither $r_j(x) \geq \ell_j$ nor $r_j(x) \leq u_j$ is expected to be active.

1 $r_j(x) \geq \ell_j$ is expected to be active.

2 $r_j(x) \leq u_j$ is expected to be active.

3 This may be used if $\ell_j = u_j$. Normally an equality constraint $r_j(x) = \ell_j = u_j$ is active at a solution.

The values 1, 2 or 3 all have the same effect when `bl`$(j) =$ `bu`(j). If necessary, `npOpt` will override the user's specification of `istate`, so that a poor choice will not cause the algorithm to fail.

`gCon(ldg,*)` is an array of dimension (`ldg`,k) for some $k \geq$ `n`. If `ncnln` $= 0$, `gCon` is not referenced. (In that case, `gCon` may be dimensioned `(ldg,1)` with `ldg` $= 1$.)

In general, `gCon` need not be initialized before the call to `npOpt`. However, if `Derivative level` $= 3$, any constant elements of `gCon` may be initialized. Such elements need not be reassigned on subsequent calls to `funcon` (see §6.6).

`cMul(nctotl)` is an array that need not be initialized if `npOpt` is called with a `Cold start` (the default).

Otherwise, the ordering of `cMul` is the same as for `bl`, `bu` and `istate`. For a `Warm start`, the components of `cMul` corresponding to nonlinear constraints must

contain a multiplier estimate. The sign of each multiplier should match `istate` as follows. If the ith nonlinear constraint is defined as "inactive" via the initial value `istate`$(j) = 0$, $j =$ `n` $+$ `nclin` $+ i$, then `cMul`(j) should be zero. If the constraint $r_j(x) \geq \ell_j$ is active (`istate`$(j) = 1$), `cMul`(j) should be non-negative, and if $r_j(x) \leq u_j$ is active (`istate`$(j) = 2$), `cMul`(j) should be non-positive.

If necessary, `npOpt` will change `cMul` to match these rules.

`Hess(ldH,*)` is an array of dimension `(ldH,`k`)` for some $k \geq$ `n`. `Hess` need not be initialized if `npOpt` is called with a `Cold Start` (the default), and will be taken as the identity. For a `Warm Start`, `Hess` provides the initial approximation of the Hessian of the Lagrangian, i.e., `H`$(i,j) \approx \partial^2 \mathcal{L}(x,\lambda)/\partial x_i \partial x_j$, where $\mathcal{L}(x,\lambda) = f_0(x) - f(x)^T\lambda$ and λ is an estimate of the optimal Lagrange multipliers. `Hess` must be a positive-definite matrix.

`x(n)` is an initial estimate of the solution.

`iw(leniw), rw(lenrw)` are integer and real arrays of workspace for `npOpt`.

Both `leniw` and `lenrw` must be at least 500. In general, `leniw` and `lenrw` should be as large as possible because it is uncertain how much storage will be needed for the basis factors. As an estimate, `leniw` should be about $100(m+n)$ or larger, and `lenrw` should be about $200(m+n)$ or larger.

Appropriate values may be obtained from a preliminary run with `leniw` = `lenrw` = 500. If `Print level` is positive, the required amounts of workspace are printed before `npOpt` terminates with `inform` = 43 or 44.

On exit:

`inform` reports the result of the call to `npOpt`. Here is a summary of possible values (for a detailed description, see §8.6):

- 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 Termination has been requested by the user.
- 7 `usrfun` seems to be giving incorrect objective derivatives.
- 8 `usrfun` seems to be giving incorrect constraint derivatives.
- 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.
- 11 The user signaled undefined functions, but no recovery was possible.
- 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.

40 Some input arguments have invalid values.

43 Not enough integer workspace to solve the problem.

44 Not enough real workspace to solve the problem.

`iter` is the number of major iterations performed.

`istate` describes the status of the constraints $\ell \le r(x) \le u$ in problem (NP). For the jth lower or upper bound, $j = 1$ to `nctotl`, the possible values of `istate`(j) are as follows (see Figure 1). δ is the appropriate feasibility tolerance.

-2 (Region 1) The lower bound is violated by more than δ.

-1 (Region 5) The upper bound is violated by more than δ.

0 (Region 3) Both bounds are satisfied by more than δ.

1 (Region 2) The lower bound is active (to within δ).

2 (Region 4) The upper bound is active (to within δ).

3 (Region 2 = Region 4) The bounds are equal and the equality constraint is satisfied (to within δ).

These values of `istate` are labeled in the printed solution according to the table in Figure 2.

Region	1	2	3	4	5	$2 \equiv 4$
`istate`(j)	-2	1	0	2	-1	3
Printed solution	`--`	`LL`	`FR`	`UL`	`++`	`EQ`

Figure 2: Labels used in the printed solution for the regions of Figure 1.

`fCon` is an array of dimension at least `ncnln`. If `ncnln` $= 0$, `fCon` is not accessed, and may then be declared to be of dimension `(1)`, or the actual parameter may be any convenient array. If `ncnln` > 0, `fCon` contains the values of the nonlinear constraint functions $f_i(x)$, $i = 1$: `ncnln`, at the final iterate.

`gCon` contains the Jacobian matrix of the nonlinear constraints at the final iterate, i.e., `gCon`(i, j) contains the partial derivative of the ith constraint function with respect to the jth variable, $i = 1$: `ncnln`, $j = 1$: `n`. (See the discussion of `gCon` under `funcon` in §6.5.)

`cMul` contains the QP multipliers from the last QP subproblem. `cMul`(j) should be non-negative if `istate`$(j) = 1$ and non-positive if `istate`$(j) = 2$.

`fObj` is the value of the objective $f_0(x)$ at the final iterate.

`gObj(n)` contains the objective gradient (or its finite-difference approximation) at the final iterate.

`Hess(ldH,*)` contains information about H, the Hessian of the Lagrangian. `Hess` is an estimate of the Hessian of the Lagrangian at `x`.

`x` contains the final estimate of the solution.

6.3. User-supplied subroutines for `npOpt`

The user must provide subroutines that define the objective function and nonlinear constraints. The objective function is defined by subroutine `funobj`, and the nonlinear constraints are defined by subroutine `funcon`. *On every call*, these subroutines must return appropriate values of the objective and nonlinear constraints in `fObj` and `fCon`. The user should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see §7 for a discussion of the optional parameter `Derivative level`. Just before either `funobj` or `funcon` is called, each element of the current gradient array `g` or `gCon` is initialized to a special value. On exit, any element that retains the given value is estimated by finite differences.

For maximum reliability, it is preferable for the user to provide *all* partial derivatives (see Chapter 8 of Gill, Murray and Wright [11], for a detailed discussion). If all gradients cannot be provided, it is similarly advisable to provide as many as possible. While developing the subroutines `funobj` and `funcon`, the `Verify` parameter (see §7) should be used to check the calculation of any known gradients.

6.4. Subroutine funobj

This subroutine must calculate the objective function $f_0(x)$ and (optionally) the gradient $g(x)$.

```
      subroutine funobj
     &   ( mode, n, x, fObj, gObj, nState )
      integer
     &     mode, n, nState
      double precision
     &     fObj, x(n), gObj(n)
```

On entry:

`mode` is set by `npOpt` to indicate which values are to be assigned during the call of `funobj`. If `Derivative level` = 1 or `Derivative level` = 3, then all components of the objective gradient are defined by the user and `mode` will always have the value 2. If some gradient elements are unspecified, `npOpt` will call `funobj` with `mode` = 0, 1 or 2.

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

`n` is the number of variables, i.e., the dimension of `x`. The actual parameter `n` will always be the same Fortran variable as that input to `npOpt`, and *must not be altered by `funobj`*.

`x(n)` is an array containing the values of the variables x for which f_0 must be evaluated. *The array `x` must not be altered by `funobj`.*

`nState` allows the user to save computation time if certain data must be read or calculated only once. If `nState` = 1, `npOpt` is calling `funobj` for the first time. If there are nonlinear constraints, the first call to `funcon` will occur before the first call to `funobj`.

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 α, if you set `mode` to -1, `npOpt` 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 $f_0(x)$ (except perhaps if `mode` $= 1$).

`gObj` must contain the assigned components of the gradient vector $g(x)$, i.e., `gObj`(j) contains the partial derivative $\partial f_0(x)/\partial x_j$ (except perhaps if `mode` $= 0$).

6.5. Subroutine funcon

This subroutine must compute the nonlinear constraint functions $\{f_i(x)\}$ and (optionally) their derivatives. (A dummy subroutine `funcon` must be provided if there are no nonlinear constraints.) The ith row of the Jacobian `gCon` is the vector $(\partial f_i/\partial x_1, \partial f_i/\partial x_2, \ldots, \partial f_i/\partial x_n)$.

```
      subroutine funcon
     &   ( mode, ncnln, n, ldg,
     &     needc, x, fCon, gCon, nState )

      integer
     &     mode, ncnln, n, ldg, nState, needc(*)
      double precision
     &     x(n), fCon(*), gCon(ldg,*)
```

On entry:

`mode` is set by `npOpt` to request values that must be assigned during each call of `funcon`. `mode` will always have the value 2 if all elements of the Jacobian are available, i.e., if `Derivative level` is either 2 or 3 (see §7). If some elements of `gCon` are unspecified, `npOpt` will call `funcon` with `mode` $= 0$, 1, or 2:

- If `mode` $= 2$, only the elements of `fCon` corresponding to positive values of `needc` need to be set (and similarly for the known components of `gCon`).
- If `mode` $= 1$, the knowm components of the rows of `gCon` corresponding to positive values in `needc` must be set. Other rows of `gCon` and the array `fCon` will be ignored.
- If `mode` $= 0$, the components of `fCon` corresponding to positive values in `needc` must be set. Other components and the array `gCon` are ignored.

`ncnln` is the number of nonlinear constraints, i.e., the dimension of `fCon`. The actual parameter `ncnln` is the same Fortran variable as that input to `npOpt`, and *must not be altered by* `funcon`.

`n` is the number of variables, i.e., the dimension of `x`. The actual parameter `n` is the same Fortran variable as that input to `npOpt`, and *must not be altered by funcon.*

`ldg` is the leading dimension of the array `gCon` ($\texttt{ldg} \geq 1$ and $\texttt{ldg} \geq \texttt{ncnln}$).

`needc` is an array of dimension at least `ncnln` containing the indices of the elements of `fCon` or `gCon` that *must* be evaluated by `funcon`. `needc` can be ignored if every constraint is provided.

`x` is an array of dimension at least `n` containing the values of the variables `x` for which the constraints must be evaluated. *x must not be altered by funcon.*

`nState` has the same meaning as for `funobj`.

On exit:

`mode`

`fCon` is an array of dimension at least `ncnln` that contains the appropriate values of the nonlinear constraint functions. If `needc`(i) is nonzero and `mode` $= 0$ or 2, the value of the ith constraint at `x` must be stored in `fCon`(i). (The other components of `fCon` are ignored.)

`gCon` is an array of declared dimension `(ldg,`k`)`, where $k \geq$ `n`. It contains the appropriate elements of the Jacobian evaluated at `x`. (See the discussion of `mode` and `gCon` above.)

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

6.6. Constant Jacobian elements

If all constraint gradients (Jacobian elements) are known (i.e., `Derivative Level = 2` or `3`, any *constant* elements may be assigned to `gCon` one time only at the start of the optimization. An element of `gCon` that is not subsequently assigned in `funcon` will retain its initial value throughout. Constant elements may be loaded into `gCon` either *before* the call to `npOpt` or during the the first call to `funcon` (signalled by the value `nState` $= 1$). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case `gCon` may be initialized to zero and nonzero elements may be reset by `funcon`.

Note that constant *nonzero* elements do affect the values of the constraints. Thus, if `gCon`(i,j) is set to a constant value, it need not be reset in subsequent calls to `funcon`, but the value `gCon`(i,j)`*x`(j) must nonetheless be added to `fCon`(i).

It must be emphasized that, if `Derivative level` < 2, unassigned elements of `gCon` are *not* treated as constant; they are estimated by finite differences, at non-trivial expense.

7. Optional parameters

The performance of each SNOPT interface 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. This options may be defined in a file called a *SPECS file*, or may be defined in the calling program using a call to one of the option-setting routines `snSet`, `snSeti` and `snSetr` (see Section 7.4). At any stage of the computation, the current value of an optional parameter may be examined by calling one of the routines `snGet`, `snGeti` and `snGetr` (see Section 7.5).

7.1. The SPECS file

The specs file contains a list of option defintions, using data in the following general form:

```
Begin options
   Iterations limit             500
   Minor feasibility tolerance  1.0e-7
   Solution                     Yes
End 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.

7.2. SPECS file checklist and defaults

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

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

```
BEGIN checklist of SPECS file parameters and their default values
* Printing
  Major print level               1         * one-line major iteration log
  Minor print level               1         * one-line minor iteration log
  Print   file                    9         *
  Summary file                    6         * typically the screen
  Print   frequency             100         * minor iterations log on PRINT file
  Summary frequency             100         * minor iterations log on SUMMARY file
  Solution                      Yes         * on the PRINT file
* Suppress options listing                  * default: options are listed

* Problem specification
  Objective row                   1         * Objective row of F
  Minimize                                  * (opposite of Maximize)
* Feasible point                            * (alternative to Max or Min)

* 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

* Derivative checking
  Verify level                   0        * cheap check on gradients
  Start objective  check at col  1
  Stop  objective  check at col  n
  Start constraint check at col  1
  Stop  constraint check at col  n

* 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
  Pivot tolerance                3.7e-11 * ϵ^(2/3)

* QP subproblems
  Crash option                   3        * first basis is essentially triangular
  Elastic weight                 2.0e+4   * used only during elastic mode
  Iterations limit               10000    * or 20m if that is more
  Partial price                  1        * 10 for large LPs

* SQP method
  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 if that is less
  Reduced Hessian dimension      500      * or Superbasics limit
  Derivative level                        * NOT ALLOWED IN snOptA
  Derivative option              1        * assumes all gradients are known
  Derivative linesearch                   *
  Function precision             3.0e-13 * ϵ^0.8 (almost full accuracy)
  Difference interval            5.5e-7   * (Function precision)^(1/2)
```

```
  Central difference interval    6.7e-5  * (Function precision)^(1/3)
  New superbasics limit          99      * controls early termination of QPs
  Objective row                  ObjRow  * row number of objective in F(x)
  Penalty parameter               0.0    * initial penalty parameter
  Proximal point method             1    * satisfies linear constraints near x0
  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 <= 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

* LU options
  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  *
  LU partial  pivoting                   * default pivot strategy
  LU rook     pivoting                   * use rook pivoting for the LU
  LU complete pivoting                   * use complete pivoting for the LU

* 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     *

* Miscellaneous
  Debug  level                   0       * for developers
  Timing level                   3       * prints cpu times
End of SPECS file checklist
```

7.3. 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, lencw, leniw, lenrw, iw(leniw)
      double precision
     &     rw(lenrw)
      character*8
     &     cw(lencw)
```

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.

7.4. 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 )

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

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: `len(buffer)` ≤ 72 (`snSet`) or ≤ 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.

7.5. 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
      integer
     &     ivalue, inform, lencw, leniw, lenrw, iw(leniw)
      character*8
     &     cvalue, cw(lencw)
      double precision
     &     rvalue, rw(lenrw)
```

On entry:

`buffer` is a string to be decoded. Restriction: `len(buffer)` ≤ 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 associated with an MPS file. For example, for the name of the bounds section use

```
call snGetc( 'Bounds', MyBounds, inform, cw, ... )
```

`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.

7.6. 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. In the description of the options we use the notation of the problem format (SparseNP) to refer to the objective and constraint functions.

`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 = $\epsilon^{\frac{1}{3}} \approx$ `6.0e-6`

When `Derivative option` = 0 with the `snOptA` interface, or `Derivative level` < 3) with `snOptB` or `snOptC`, the central-difference interval r is used near an optimal solution to obtain more accurate (but more expensive) estimates of gradients. Twice as many function evaluations are required compared to forward differencing. The interval used for the jth variable is $h_j = r(1 + |x_j|)$. The resulting derivative estimates should be accurate to $O(r^2)$, unless the functions are badly scaled.

`Check frequency` i Default = 60

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

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

`Crash option`	i	Default = 3
`Crash tolerance`	r	Default = 0.1

Except on restarts, a CRASH procedure is used to select an initial basis from certain rows and columns of the constraint matrix $(\,A \quad -I\,)$. The `Crash option` i determines which rows and columns of A are eligible initially, and how many times CRASH is called. Columns of $-I$ are used to pad the basis where necessary.

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

If $i \ge 1$, certain slacks on inequality rows are selected for the basis first. (If $i \ge 2$, 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 $|a_{ij}| \le a_{\max} \times r$. (To be meaningful, r should be in the range $0 \le r < 1$.)

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

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

`Derivative level`	i	Default = 3

This keyword is used by the **`snOptB`**, **`snOptC`** and **`npOpt`** interfaces. *It should not be used when calling* ***`snOptA`***. The keyword `Derivative level` specifies which nonlinear function gradients are known analytically and will be supplied to SNOPT by the user subroutines `funobj` and `funcon`.

i	*Meaning*

3 All objective and constraint gradients are known.

2 All constraint gradients are known, but some or all components of the objective gradient are unknown.

1 The objective gradient is known, but some or all of the constraint gradients are unknown.

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

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

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

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

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

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

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

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

At each major iteration a line search 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 α_k. If some analytic derivatives are not provided, or a `Nonderivative linesearch` is specified, SNOPT employs a line search based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

A nonderivative line search 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 line search may give a significant decrease in computation time.

If `Nonderivative linesearch` is selected, `snOptA` signals the evaluation of the line search by calling `usrfun` with `needG` = 0. Once the line search is completed, the problem functions are called again with `needF` = 0 and `needG` = 0. If the potential savings provided by a nonderivative line search are to be realized, it is essential that `usrfun` be coded so that the derivatives are not computed when `needG` = 0.

The selection of `Nonderivative linesearch` for `snOptB` means that `funobj` and `funcon` are called with `mode` $= 0$ in the line search. 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` $= 0$.

`Derivative option` i Default $= 1$

This option is intended for `snOptA` only and *should not be used with any other interface.* The `Derivative option` specifies which nonlinear function gradients are known analytically and will be supplied to `snOptA` by the user subroutine `usrfun`.

i	*Meaning*
0	Some problem derivatives are unknown.
1	All problem derivatives are known.

The value $i =$ `1` should be used whenever possible. It is the most reliable and will usually be the most efficient.

If $i =$ `0` `snOptA` will *estimate* the missing components of $G(x)$ using finite differences. This may simplify the coding of subroutine `usrfun`. However, it could increase the total run-time substantially (since a special call to `usrfun` is required for each column of the Jacobian that has a 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).

For each column of the Jacobian, one call to `usrfun` is needed to estimate all missing elements in that column, if any. If the sparsity pattern of the Jacobian happens to be

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

where `*` indicates known derivatives and `?` indicates unknown elements, `snOptA` will use one call to `usrfun` 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 `usrfun` are then needed. (This is not under the user's control.)

`Difference interval` h_1 Default $= \epsilon^{1/2} \approx$ `1.5e-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 problem derivatives.
- For verifying the problem derivatives.
- For estimating missing derivatives.

In all 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 $f_0(x)$ or $f(x)$ at the perturbed point. The resulting gradient estimates should be accurate to $O(h_1)$ unless the functions are badly scaled. Judicious alteration of h_1 may sometimes lead to greater accuracy.

`Dump file` i Default = 0

If $i > 0$, the last solution obtained will be output to the file with unit number i in the format described in Section 9.3.

`Elastic weight` ω Default = 10^4

This keyword determines the initial weight γ associated with problem NP(γ).

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 $\omega(1 + \|g(x_k)\|_\infty)$, where g is the objective gradient. In either case, the QP is re-solved in elastic mode with $\gamma = \omega(1 + \|g(x_k)\|_\infty)$.

Thereafter, γ is increased (subject to a maximum allowable value) at any point that is optimal for problem NP(γ), but not feasible for (NP). After the rth increase, $\gamma = \omega 10^r(1 + \|g(x_{k1})\|_\infty)$, where x_{k1} is the iterate at which γ 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 δ. Over a period of i iterations, the tolerance actually used by SNOPT increases from 0.5δ to δ (in steps of $0.5\delta/i$).

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

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`
see `Minimize`

`Function precision` ϵ_R Default = $\epsilon^{0.8} \approx$ `3.7e-11`

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

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

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

- In some cases the function values will be the result of extensive computation, possibly involving an iterative procedure that can provide rather few digits of precision at reasonable cost. Specifying an appropriate `Function precision` may lead to savings, by allowing the line search 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 9.2.

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

`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 −`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 line search 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 line search.
- 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 9.3.

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

`Log frequency` k Default = 100

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

- For large and relatively dense problems, $r_1 = 5.0$ (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 Partial  Pivoting` Default
`LU Rook     Pivoting`
`LU Complete Pivoting`

The LU factorization implements a Markowitz-type search for a pivot that locally minimizes the fill-in subject to a threshhold pivoting stability criterion. The default option is to use threshhold partial pivoting. The options `LU rook pivoting` and `LU complete pivoting` are more expensive than partial pivoting but are more stable and better at revealing rank.

`LU complete pivoting` is the default for the `npOpt` interface.

`LU density tolerance` r_1 Default = 0.6
`LU singularity tolerance` r_2 Default = $\epsilon^{2/3} \approx$ `3.7e-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 $|U_{jj}| \le r_2$ or $|U_{jj}| < r_2 \max_i |U_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

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

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

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

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

$$\texttt{rowerr} = \max_i \ \texttt{viol}_i / \|x\| \ \le \ \epsilon_r, \tag{7.1}$$

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

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` ϵ_d Default = `1.0e-6`

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

$$\texttt{maxComp} = \max_j \ \texttt{Comp}_j / \|\pi\| \ \le \ \epsilon_d, \tag{7.2}$$

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

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

In the major iteration log, `maxComp` 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` ≥ 1).
- `b` BASIS statistics, i.e., information relating to the basis matrix whenever it is refactorized. (This output is always provided if `JFDXbs` ≥ 10).
- `X` x_k, the nonlinear variables involved in the objective function or the constraints.
- `D` π_k, the dual variables for the nonlinear constraints.
- `F` $F(x_k)$, the values of the nonlinear constraint functions.
- `J` $J(x_k)$, the Jacobian matrix.

To obtain output of any 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 line search. It applies to all nonlinear problems, once a "feasible solution" or "feasible subproblem" has been found.

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

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

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

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

`Minimize` Default
`Maximize`
`Feasible point`

The keywords `Minimize` and `Maximize` specify the required direction of optimization. It applies to both linear and nonlinear terms in the objective.

The keyword `feasible point` 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.

`Minor iterations limit` k Default = 500

If the number of minor iterations for the optimality phase of the QP subproblem exceeds k, then all nonbasic QP variables that have not yet moved are frozen at their current values and the reduced QP is solved to optimality.

Note that more than k minor iterations may be necessary to solve the reduced QP to optimality. These extra iterations are necessary to ensure that the terminated point gives a suitable direction for the line search.

In the major iteration log, a `t` at the end of a line indicates that the corresponding QP was artificially terminated using the limit k.

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

`Minor feasibility tolerance` t Default = `1.0e-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 $t =$ `1.0e-6`, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious. In general, keep x as far away from singularities as possible.)

- If `Scale option` ≥ 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 print level` k Default = 1

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.

≥ 1 A single line of output each minor iteration (controlled by `Print frequency` and `Summary frequency`).

≥ 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 kth 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.

`New superbasics limit` i Default = 99

This option causes early termination of the QP subproblems if the number of free variables has increased significantly since the first feasible point. If the number of new superbasics is greater than i the nonbasic variables that have not yet moved are frozen and the resulting smaller QP is solved to optimality.

In the major iteration log, a "`T`" at the end of a line indicates that the QP was terminated early in this way.

`Old basis file` f Default = 0

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

The file will usually have been output previously as a `New Basis file`. It will not be acceptable if the number of rows or columns in the problem has been altered.

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

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

- When $i = 1$, all columns of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$ are searched.

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

`Pivot tolerance` r Default $= \epsilon^{2/3} \approx$ `3.7e-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 $x+\alpha p$ for some search direction p, a "ratio test" is used to determine which component of x reaches an upper or lower bound first. The corresponding element of p is called the pivot element.
- 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 $= 100$

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

`Proximal point method` i Default $= 1$

$i = 1$ or 2 specifies minimization of $\|x - x_0\|_1$ or $\frac{1}{2}\|x - x_0\|_2^2$ when the starting point x_0 is changed to satisfy the linear constraints (where x_0 refers to nonlinear variables).

`Punch file` f Default $= 0$

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

`Save frequency` k Default $= 100$

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

`Scale option`	i	Default = 2 (LP) or 1 (NLP)
`Scale 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 $\begin{pmatrix} A & -I \end{pmatrix}$ 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:

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

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

`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`).

```
Start Objective  Check at Column          k          Default = 1
Start Constraint Check at Column          k          Default = 1
Stop  Objective  Check at Column          l          Default = n1'
Stop  Constraint Check at Column          l          Default = n1''
```

If `Verify level` > 0, these keywords may be used to abbreviate the verification of individual derivative elements computed by subroutine `usrfun`. 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 option would again be appropriate.

```
Summary file              f                          Default = 6
Summary frequency         k                          Default = 100
```

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

```
Superbasics limit         i          Default = min{500, n1 + 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 `cw`, `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`($\mathtt{maxrw} + 1$: `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.

`Timing level` i Default = 3

$i = 0$ suppresses output of cpu times. (Intended for installations with dysfunctional timing routines.)

`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_0 is evaluated at points of the form $x+\alpha p$, where x and p are fixed and α varies. if $|f_0|$ exceeds $f_{\max}$ or α exceeds $\alpha_{\max}$, iterations are terminated with the exit message `Problem is unbounded (or badly scaled)`.

If singularities are present, unboundedness in $f_0(x)$ may be manifested by a floating-point overflow (during the evaluation of $f_0(x + \alpha p)$), 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 derivatives computed by the user-provided routines. Derivatives 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 `usrfun` for `snOptA`, and 2 calls to `funcon` and 3 calls to `funobj` for `snOptB`.
1	Individual 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 problem Jacobian will be checked.
3	Options 2 and 1 will both occur (in that order).
-1	Derivative checking is disabled.

`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 derivatives are not checked, so they result in no overhead.

`Violation limit` τ 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

$$v_i(x_{k+1}) \leq \tau \max\{1, v_i(x_0)\}, \tag{7.3}$$

where x_0 is the point at which the nonlinear constraints are first evaluated and $v_i(x)$ is the ith 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 τ. 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 τ may be any large positive value.

8. 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).

8.1. 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 131 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.

8.2. The major iteration log

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

Label	*Description*
`Itns`	The cumulative number of minor iterations.
`Major`	The current major iteration number.
`Minors`	is the number of iterations required by both the feasibility and optimality phases of the QP subproblem. Generally, `Minors` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see §2).
`Step`	The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.
`nCon`	The number of times subroutines `usrfun` or `funcon` have been called to evaluate the nonlinear problem functions. Evaluations needed for the estimation of the derivatives by finite differences are not included. `nCon` is printed as a guide to the amount of work required for the line search.
`Feasible`	is the value of `rowerr`, the maximum component of the scaled nonlinear constraint residual (7.1). The solution is regarded as acceptably feasible if `Feasible` is less than the `Major feasibility tolerance`. In this case, the entry is contained in parenthesis. 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 (7.2). 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.

`MeritFunction` 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.

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

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

As columns of B are replaced during the minor iterations, `LU` may fluctuate up or down but in general 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.

`BSwap` 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 $B_S = (\, B \quad S\,)^T$ with stability being favored more than sparsity.

`nS` The current number of superbasic variables.

`CondHz` An estimate of the condition number of R^TR, an estimate of Z^THZ, 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^TR). `Cond Hz` gives a rough indication of whether or not the optimization procedure is having difficulty. If ϵ is the relative precision of the machine being used, the SQP algorithm will make slow progress if `Cond Hz` becomes as large as $\epsilon^{-1/2} \approx 10^8$, 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.

`Penalty` is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if there are no nonlinear constraints.

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

Code *Meaning*

`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 line search 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 line search it was necessary to decrease the step in order to obtain a maximum constraint violation conforming to the value of `Violation limit`.

`l` The norm-wise change in the variables was limited by the value of the `Major step limit`. If this output occurs repeatedly during later iterations, it may be worthwhile increasing the value of `Major step limit`.

`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(γ).

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.

`n` No positive-definite BFGS update could be found. The approximate Hessian is unchanged from the previous iteration.

`R` The approximate Hessian has been reset by discarding all but the diagonal elements. This reset will be forced periodically by the `Hessian frequency` and `Hessian updates` 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.

`t` The minor iterations were terminated because of the `Minor iteration limit`.

`T` The minor iterations were terminated because of the `New superbasics limit`.

`u` The QP subproblem was unbounded.

`w` A weak solution of the QP subproblem was found.

`z` The superbasic limit was reached.

8.3. The minor iteration log

If `Minor print level` > 0, one line of information is output to the PRINT file every kth minor iteration, where k is the specified `Minor print frequency` (default $k = 1$). A heading is printed before the first such line following a basis factorization. The heading contains the items described below. In this description, a PRICE operation is defined to be the process by which 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 $A_{\mathtt{pp}}$ or $I_{\mathtt{pp}}$, the `pp`th segments of the constraint matrix $(A \quad -I)$.

Label *Description*

`Itn` The current iteration number.

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

Note that `dj` is the 1-norm of the reduced-gradient vector at the start of the iteration, just after the PRICE operation.

`LPstep,QPstep` The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. If a variable is made superbasic during the current iteration (`+SBS` > 0), `Step` will be the step to the nearest bound. During Phase 2, the step can be greater than one only if the reduced Hessian is not positive definite.

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

`SumInf` 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, `SumInf` may occasionally increase.

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

`rgNorm` 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.

(The heading is not printed if the problem is linear.)

`LPobjective,QPobjective,Elastic QPobj` The QP objective function after the present iteration. In elastic mode, the heading is changed to `Elastic QPobj`. In either case, the value printed decreases monotonically.

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

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

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

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

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

`L+U` The number of nonzeros representing the basis factors L and U. Immediately after a basis factorization $B = LU$, this is `lenL+lenU`, the number of subdiagonal elements in the columns of a lower triangular matrix and the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix. Further nonzeros are added to `L` when various columns of B are later replaced. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `L` will steadily increase, whereas the value of `U` may fluctuate up or down. Thus, in general, the value of `L+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.

`nS` The current number of superbasic variables. (The heading is not printed if the problem is linear.)

`cond Hz` See the major iteration log. (The heading is not printed if the problem is linear.)

8.4. Basis factorization statistics

If `Major print level` ≥ 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 §7.6.

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

Label	*Description*
`Factorize`	The number of factorizations since the start of the run.
`Demand`	A code giving the reason for the present factorization.

Code	*Meaning*
0	First LU factorization.
1	The number of updates reached the `Factorization Frequency`.
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 `Check Frequency`).
11	Ill-conditioning has caused inconsistent results.

`Itn` The current minor iteration number.

`Nonlin` The number of nonlinear variables in the current basis B.

`Linear` The number of linear variables in B.

`Slacks` The number of slack variables in B.

`B BR BS` or `BT factorize` The type of LU factorization.

- `B` Periodic factorization of the basis B.
- `BR` More careful rank-revealing factorization of B using threshold rook pivoting. This occurs mainly at the start, if the first basis factors seem singular or ill-conditioned. Followed by a normal `B` factorize.
- `BS` B_S is factorized to choose a well-conditioned B from the current $(\, B \quad S \,)$. Followed by a normal `B` factorize.
- `BT` Same as `BS` except the current B is tried first and accepted if it appears to be not much more ill-conditioned than after the previous `BS` factorize.

`m` The number of rows in B or B_S.

`n` The number of columns in B or B_S. Preceded by "`=`" or "`>`" respectively.

`Elems` The number of nonzero elements in B or B_S.

`Amax` The largest nonzero in B or B_S.

`Density` The percentage nonzero density of B or B_S.

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

`lenL` The number of nonzeros in L.

`Cmpressns` 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.

`Incres` The percentage increase in the number of nonzeros in L and U relative to the number of nonzeros in B or B_S.

`Utri` is the number of triangular rows of B or B_S at the top of U.

`lenU` The number of nonzeros in U.

`Ltol` The maximum subdiagonal element allowed in L. This is the specified `LU factor tolerance` or a smaller value that is currently being used for greater stability.

`Umax` The maximum nonzero element in U.

`Ugrwth` The ratio `Umax/Amax`, which ideally should not be substantially larger than 10.0 or 100.0. If it is orders of magnitude larger, it may be advisable to reduce the `LU factor tolerance` to 5.0, 4.0, 3.0 or 2.0, say (but bigger than 1.0).

As long as `Lmax` is not large (say 10.0 or less), max{`Amax`, `Umax`} / `DUmin` gives an estimate of the condition number of B. If this is extremely large, the basis is nearly singular. Slacks are used to replace suspect columns of B and the modified basis is refactored.

`Ltri` is the number of triangular columns of B or B_S at the left of L.

`dense1` is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.

`Lmax` The actual maximum subdiagonal element in L (bounded by `Ltol`).

`Akmax` The largest nonzero generated at any stage of the LU factorization. (Values much larger than `Amax` indicate instability.)

`growth` The ratio `Akmax/Amax`. Values much larger than 100 (say) indicate instability.

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

`dense2` is the number of columns remaining when the density of the basis matrix being factorized reached 0.6. (The Markowitz pivot strategy searches fewer columns at that stage.)

`DUmax` The largest diagonal of PUQ.

`DUmin` The smallest diagonal of PUQ.

`condU` The ratio `DUmax/DUmin`, which estimates the condition number of U (and of B if `Ltol` is less than 100, say).

8.5. Crash statistics

If `Major print level` ≥ 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.

Label	*Description*
`Slacks`	is the number of slacks selected initially.
`Free cols`	is the number of free columns in the basis, including those whose bounds are rather far apart.
`Preferred`	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.
`Unit`	is the number of unit columns in the basis.
`Double`	is the number of columns in the basis containing 2 nonzeros.
`Triangle`	is the number of triangular columns in the basis with 3 or more nonzeros.
`Pad`	is the number of slacks used to pad the basis (to make it a nonsingular triangle).

8.6. 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`
`EXIT -- feasible point found` (from option `Feasible point only`)

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

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

then the objective function would probably change in the dth 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 linear constraints are infeasible`

`EXIT -- infeasible problem, nonlinear infeasibilities minimized`

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 §7.6), 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 -- termination requested in objective routine
   EXIT -- termination requested in constraint routine
   EXIT -- termination requested in function  routine
```

These exits occur if a value `Status` < -1 is set during some call to the user-defined routines. SNOPT assumes that you want the problem to be abandoned forthwith.

```
7  EXIT -- some objective derivatives appear to be incorrect
```

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 (`G(`k`)` or `gObj(`j`)`) is being set to a value that disagrees markedly with its associated forward-difference estimate $\partial f_0/\partial x_j$. (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 `usrfun` or `funobj`. A simple omission (such as forgetting to divide f_0 by 2) could explain everything. If f_0 or a component $\partial f_0/\partial x_j$ 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 -- some constraint derivatives appear to be incorrect
```

This is analogous to the preceding exit. At least one of the computed constraint derivatives is significantly different from an estimate obtained by forward-differencing the vector $F(x)$. Follow the advice given above, trying to ensure that the arrays `F` and `G` are being set correctly in `usrfun` or `funcon`.

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

Several circumstances could lead to this exit.

1. Subroutines `usrfun`, `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

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

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

```
11  EXIT -- undefined functions with no recovery possible
    EXIT -- undefined functions at the first point
    EXIT -- undefined functions at the first feasible point
    EXIT -- undefined functions when checking derivatives
```

The user has indicated that the problem functions are undefined by assigning the value `Status` $= -1$ on exit from `usrfun`, `funobj` or `funcon`. SNOPT attempts to evaluate the problem functions closer to a point at which the functions are already known to be defined. This exit occurs if SNOPT is unable to find a point at which the functions are defined.

```
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. If there are some linear constraints and variables, try `Scale option 1` if scaling has not yet been used.

For certain highly structured basis matrices (notably those with band structure), a systematic growth may occur in the factor U. Consult the description of `Umax`, `Umin` and `Growth` in §8.4, and set the `LU factor tolerance` to 2.0 (or possibly even smaller, but not less than 1.0).

> If the following exits occur during the *first* basis factorization, the 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, i.e., the arrays `A(*)`, `iAfun(*)`, `jAvar(*)`, `iGfun(*)`, and `jGvar(*)` for `snOptA`, or `indA`, `Acol` and `locA` for `snOptB`.

`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 are using `snOptA` and you have added elements to `A(*)`, `iAfun(*)`, and `jAvar(*)` or `iGfun(*)`, and `jGvar(*)`, 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 "$j \quad x_j$" 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, and that all "real" variables and arrays are declared consistently. (They should be `double precision` on most machines.)

`40 EXIT -- Input arguments out of range`
At least one input argument for the interface is invalid. The printed output provides more detail of which argument(s) must be modified.

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

`41 EXIT -- char, int and real work arrays must have at least 500 elements`
SNOPT cannot start to solve a problem until these arrays are correctly defined.

`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.

8.7. 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 §8. 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 ± 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 $f_0(x)$ and the other maximizes $-f_0(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j will be reversed.

The ROWS section

General linear constraints take the form $l \le Ax \le u$. The ith constraint is therefore of the form

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

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 $l \le s \le u$. For the ith "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 $\alpha \le f_i(x) + a^T x \le \beta$ 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.

Label	*Description*
`Number`	The value $n + i$. This is the internal number used to refer to the ith slack in the iteration log.
`Row`	The name of the ith row.
`State`	The state of the ith row relative to the bounds α and β. The various states possible are as follows.

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

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

`A` *Alternative optimum possible.* The slack is nonbasic, but its reduced gradient is essentially zero. This means that if the slack were allowed to start moving 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.* The slack is basic, but it is equal to (or very close to) one of its bounds.

`I` *Infeasible.* The slack is basic and is currently violating one of its bounds by more than the `Feasibility tolerance`.

`N` *Not precisely optimal.* The slack 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 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` α, the lower bound on the row.

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

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

`I` The constraint number, i.

The COLUMNS section

Here we talk about the "column variables" x_j, $j = 1\colon n$. We assume that a typical variable has bounds $\alpha \le x_j \le \beta$.

Label *Description*

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

`Column` The name of x_j.

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

`LL` x_j is nonbasic at its lower limit, α.

`UL` x_j is nonbasic at its upper limit, β.

`EQ` x_j is nonbasic and fixed at the value $\alpha = \beta$.

`FR` x_j is nonbasic 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 jth 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` α, the lower bound on x_j.

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

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

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

8.8. The SOLUTION file

The information in a printed solution (§8.7) 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.)

8.9. 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 §3.2, using `Major print level 1` and `Minor print level 0`.

```
==============================
S N O P T  6.2-1(1) (Jan 2003)
==============================

Begin  Toy NLP problem

===> snOptA wrapper


Scale option  0,    Partial price   1


Nonlinear constraints       2     Linear constraints       0
Nonlinear variables         2     Linear variables         0
Jacobian  variables         2     Objective variables      2
Total constraints           2     Total variables          2




This is problem  Toy1
The user has defined       6   out of       6   first  derivatives


Major Minors     Step   nCon Feasible  Optimal  MeritFunction    nS Penalty
    0      2               1  4.1E-01  5.0E-01  1.0000000E+00     2           r
    1      2  1.0E+00      2 (0.0E+00) 3.1E-01 -4.1421356E-01     1         n rl
    2      1  1.0E+00      3  1.4E+00  1.5E-01 -9.3802987E-01     1         s
    3      1  1.0E+00      4  1.8E-01  3.3E-02 -9.6547072E-01     1 2.8E-03
    4      1  1.0E+00      5  3.4E-02  8.9E-03 -9.9129962E-01       2.8E-03
    5      0  1.0E+00      6  4.2E-02  4.8E-03 -1.0000531E+00       2.8E-03
    6      0  1.0E+00      7  1.9E-04  2.0E-05 -9.9999997E-01       3.3E-02
    7      0  1.0E+00      8 (3.7E-09)(4.0E-10)-1.0000000E+00       3.3E-02

EXIT -- optimal solution found

Problem name                 Toy1
No. of iterations                   7   Objective value     -1.0000000008E+00
No. of major iterations             7   Linear objective     0.0000000000E+00
Penalty parameter           3.253E-02   Nonlinear objective -1.0000000008E+00
No. of calls to funobj              9   No. of calls to funcon              9
No. of degenerate steps             0   Percentage                       0.00
Norm of x                     7.1E-01   Norm of pi                    1.0E+00
Max Primal infeas           0 0.0E+00   Max Dual infeas             2 8.0E-10
Nonlinear constraint violn    6.4E-09



Solution printed on file  15

Finished problem Toy1

Time for MPS input                          0.00 seconds
Time for solving problem                    0.01 seconds
Time for solution output                    0.00 seconds
Time for constraint functions               0.00 seconds
Time for objective function                 0.00 seconds

snOptA finished.
inform = 0
nInf   = 0
sInf   =  0.
Obj    = -1.
```

9. BASIS files

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

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

```
New Basis file     10
Backup    file     11
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 kth 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 3–5 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.

9.1. NEW and OLD BASIS files

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

As illustrated in Figure 3, 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)`.

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

If variable j is 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

$$j \qquad x_j$$

written with `format(i8, 1p, e24.14)` and terminated by an entry with $j = 0$, where j denotes the jth variable and x_j is a real value. The jth variable is either the jth column or the $(j-n)$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) $\neq$ 3, variable j will be initialized at the value x_j and its state will be reset to 2 (superbasic). If the number of superbasic variables has already reached the `Superbasics limit`, then variable j will be made nonbasic at 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 3: Format of NEW and OLD BASIS files

9.2. PUNCH and INSERT files

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

The standard MPS format has been slightly generalized to allow the saving and reloading of nonbasic solutions. It is illustrated in Figure 4. Apart from the first and last line, each entry has the following form:

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

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

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

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

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

Notes on PUNCH data

1. Variables are output in natural order. For example, on the first `XL` or `XU` line, *Name1* will be the first basic column and *Name2* will be the first row whose slack is not basic.

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 `ENDATA` line, to specify additional superbasic columns or slacks.

5. An `SB` line will not alter the status of *Name1* if the `Superbasics limit` has been reached. However, the associated *Value* will be retained.

9.3. DUMP and LOAD files

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

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

as illustrated in Figure 5. 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
```

Figure 4: Format of PUNCH/INSERT files

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

Figure 5: 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 ℓ_1 penalty method for nonlinear constraints*, in Numerical Optimization 1984, P. T. Boggs, R. H. Byrd, and R. B. Schnabel, eds., Philadelphia, 1985, pp. 26–40.

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

[6] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM J. Optim., 12 (2002), pp. 979–1006.

[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. Program., 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] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, London and New York, 1981.

[12] B. A. Murtagh and M. A. Saunders, *Large-scale linearly constrained optimization*, Math. Program., 14 (1978), pp. 41–72.

[13] ———, *A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints*, Math. Program., 16 (1982), pp. 84–117.

[14] ———, *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 Tom Aird, Arne Drud, David Gay, Rocky Nelson and Ulf Ringertz for their feedback while running SNOPT on numerous examples.