User's Guide for SNOPT Version 7.7:
Software for
Large-Scale Nonlinear Programming*

Philip E. GILL and Elizabeth WONG

Department of Mathematics
University of California, San Diego, La Jolla, CA 92093-0112, USA

Walter MURRAY and Michael A. SAUNDERS

Systems Optimization Laboratory
Department of Management Science and Engineering
Stanford University, Stanford, CA 94305-4121, USA

Mar 2021

Abstract

SNOPT is a general-purpose system for constrained optimization. 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. Search directions are obtained from QP subproblems that minimize a quadratic model of the Lagrangian function subject to linearized constraints. An augmented Lagrangian merit function is reduced along each search direction to ensure convergence from any starting point.

On large problems, SNOPT is most efficient if only some of the variables enter nonlinearly, or there are relatively few degrees of freedom at a solution (i.e., many constraints are active). 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 is re-entrant and suitable for any machine with a Fortran compiler. SNOPT may be called from a driver program in Fortran, MATLAB, or C/C++.

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

```
pgill@ucsd.edu          http://www.CCoM.ucsd.edu/~peg
walter@stanford.edu     http://www.stanford.edu/~walter
saunders@stanford.edu   http://www.stanford.edu/~saunders
elwong@ucsd.edu         http://www.CCoM.ucsd.edu/~elwong
```

Contents

1.   Introduction

SNOPT is a general-purpose system for constrained optimization. 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 of the form

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

where x is an n-vector of variables, l and u are constant lower and upper bounds, $f_0(x)$ is a smooth scalar objective function, A_L is a sparse matrix, and $f(x)$ is a vector of smooth nonlinear constraint functions $\{f_i(x)\}$. An optional parameter `Maximize` may specify that $f_0(x)$ should be maximized instead of minimized.

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

Upper and lower bounds are specified for all variables and constraints. The jth constraint may be defined as an *equality* by setting $l_j = u_j$. If certain bounds are not present, the associated elements of l or u may be set to special values that are treated as $-\infty$ or $+\infty$. Free variables and free constraints ("free rows") have both bounds infinite.

1.1.   Problem types

If $f_0(x)$ is linear and $f(x)$ is absent, NP is a *linear program* (LP) and SNOPT applies the primal simplex method [2]. Sparse basis factors are maintained by LUSOL [11] as in MINOS [17].

If only $f_0(x)$ is nonlinear, the problem is *linearly constrained* (LC) and tends to solve more easily than the general case with nonlinear constraints (NC). For both nonlinear cases, SNOPT applies a sparse sequential quadratic programming (SQP) method [7], using limited-memory quasi-Newton approximations to the Hessian of the Lagrangian. The merit function for step length control is an augmented Lagrangian, as in the dense SQP solver NPSOL [10, 13].

If $f_0(x)$ is absent, SNOPT attempts to find the point satisfying the constraints that is the closest to the initial point specified by the user.

In general, SNOPT requires less matrix computation than NPSOL and fewer evaluations of the functions than the nonlinear algorithms in MINOS [15, 16]. It is suitable for nonlinear problems with thousands of constraints and variables, and is most efficient if only some of the variables enter nonlinearly, or there are relatively few degrees of freedom at a solution (i.e., many constraints are active). However, unlike previous versions of SNOPT, there is no limit on the number of degrees of freedom.

1.2.   Implementation

SNOPT is implemented as a library of Fortran 77 subroutines. The source code is compatible with all known Fortran compilers.

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

1.3.   The SNOPT interfaces

SNOPT contains several *interfaces* between the user and the underlying solver, allowing problems to be specified in various formats.

New users are encouraged to use the `snOptA` interface. This allows linear and nonlinear constraints and variables to be entered in *arbitrary order*, and allows all nonlinear functions to be defined in one user routine. It may also be used with SnadiOpt [6], which provides gradients by *automatic differentiation* of the problem functions.

For efficiency reasons, the solver routines require nonlinear variables and constraints to come before linear variables and constraints, and they treat nonlinear objective functions separately from nonlinear constraints. `snOptB` (the *basic* interface) imposes these distinctions and was used by all versions of SNOPT prior to Version 6.

In some applications, the objective and constraint functions share data and computation. The `snOptC` interface allows the functions to be *combined* in one user routine.

Finally, `npOpt` is an interface that accepts problem data written in the same format as the dense SQP code NPSOL. It permits NPSOL users to upgrade with minimum effort.

A summary of the SNOPT interfaces follows:

`snOptA` (Section 3) is recommended for new users of SNOPT. The variables and constraints may be coded in any order. Nonlinear objective and constraint functions are defined by one user routine. May use SnadiOpt to compute gradients.

`snOptB` (Section 4) is the basic interface to the underlying solver. Nonlinear constraints and variables must appear first. A nonlinear objective is defined separately from any nonlinear constraints.

`snOptC` (Section 5) is the same as `snOptB` except the user combines the nonlinear objective and constraints into one routine.

`npOpt` (Section 6) accepts the same problem format as NPSOL. It is intended for moderate-sized dense problems (as is NPSOL!).

1.4.   Files

Every SNOPT interface reads or creates some of the following files:

Print file (Section 8) is a detailed iteration log with error messages and perhaps listings of the options and the final solution.

Summary file (Section 8.9) is a brief iteration log with error messages and the final solution status. Intended for screen output in an interactive environment.

Specs file (Section 7) is a set of run-time options, input by `snSpec` or `snSpecF`.

Solution file (Sections 8.7–8.8) keeps a separate copy of the final solution listing.

Basis files (Section 9) allow restarts.

Unit numbers for the Specs, Print, and Summary files are defined by inputs to subroutines `snInit` and `snSpec`. Alternatively, these files may be specified as character strings by using subroutines `snInitF` and `snSpecF`. Other SNOPT files are described in Sections 8 and 9.

1.5.   Overview of the package

SNOPT is normally accessed via a sequence of subroutine calls. For example, the interface
`snOptA` is invoked by the statements

```
call snInit( iPrint, iSumm, ... )
call snSpec( iSpecs, ...         )
call snoptA( Start, nF, n, ...  )
```

where `snSpec` reads a file of run-time options (if any). Also, individual run-time options
may be "hard-wired" by calls to `snSet`, `snSeti` and `snSetr`.

1.6.   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
&     cw(lencw)*8
double precision
&     rw(lenrw)
```

On entry:

iPrint   defines a unit number for the Print file. Typically $\texttt{iPrint} = 9$. On some systems,
         the file may need to be opened before `snInit` is called.
         If $\texttt{iPrint} \leq 0$, there will be no Print file output.

iSumm   defines a unit number for the Summary file. Typically $\texttt{iSumm} = 6$.
         (In an interactive environment, this usually denotes the screen.)

         On some systems, the file may need to be opened before `snInit` is called.
         If $\texttt{iSumm} \leq 0$, there will be no Summary file output.

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

On exit:

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

1.7.   Subroutine `snInitF`

Subroutine `snInitF` provides the same capabilities as `snInit`. The difference is that file-
names are accepted as input rather than Fortran unit numbers. If `snInitF` is called, the
user should call `snEndF` (Section 1.8) to close out files (or alternatively, manually close the
files with the given Fortran unit numbers returned by `snInitF`).

```
    subroutine snInitF
&   ( printfile, summaryfile, iPrint, iSumm,
&      cw, lencw, iw, leniw, rw, lenrw )

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

On entry:

`printfile` is a character string containing the name of the Print file. If `printfile` is empty, then there will be no Print file output (analogous to `iPrint` ≤ 0 in `snInit`).

`summaryfile` is a character string containing the name of the Summary file. If `summaryfile` is empty, then there will be no Summary file output (analogous to `iSumm` ≤ 0). To route the output to standard out, set `summaryfile = 'screen'`.

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

On exit:

`iPrint` is the unit number associated with the Print file.

`iSumm` is the unit number associated with the Summary file.

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` Some elements of `cw`, `iw`, `rw` are given values to indicate that most optional parameters are undefined.

1.8. Subroutine `snEndF`

Subroutine `snEndF` is called at the end of a run to clean up and close out any output files. This subroutine is only necessary if `snInitF` is called. Alternatively, the user can manually close the files using the returned Fortran unit numbers in `snInitF`.

```
    subroutine snEndF
&   ( cw, lencw, iw, leniw, rw, lenrw )

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

On entry:

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

1.9. What's New in SNOPT

SNOPT 7.7.7

- A minor change to the termination condition for badly-scaled problems now includes a test for near-feasibility of the final iterate.

SNOPT 7.7.6

- The default value for the violation limit is changed from 10 to 1.0e+6.

SNOPT 7.7.5

- Add option for the maximum number of iterations for the proximal-point problem.

- Revert previous change to the snLog argument list.

SNOPT 7.7.4

- Several updates and fixes for HOT starts have been incorporated.

SNOPT 7.7.3

- A bug that incorrectly declared problems with infeasible elastic variables optimal has been fixed. The number and sum of infeasibilities has been updated to correctly reflect elastic infeasibilities, if they exist.

- Solution output has also been updated to print the constraint multipliers correctly, and to correctly label infeasible constraint slacks.

- Matlab mexfile bug with Fortran printfile unit that caused fort.# output has been fixed. Ctrl-C support in Matlab also updated to work with r2019a.

SNOPT 7.7.1

- C/C++ interface are compiled into the SNOPT library. C support is now enabled by default.

SNOPT 7.7.0

- SNOPT 7.7.0 introduces new subroutines `snInitF` and `snSpecF`. These subroutines are alternatives to the existing subroutines `snInit` and `snSpec`. The purpose of the new subroutines is to hide the assignment of Fortran file unit numbers and file handling from the user. Users pass the filename as a character string to `snInitF` and `snSpecF`, and the file is opened internally using an available file unit number. When using shared libraries, users may get unexpected results with Fortran unit numbers if the executable program is not linked to the appropriate Fortran runtime library. These subroutines will eliminate this issue by handling unit numbers internally.

- The `f2c`'d version of SNOPT is no longer included in the distribution.

2.   Description of the SQP Method

Here we summarize the main features of the SQP algorithm used in SNOPT and introduce some terminology used in the description of the library routines and their arguments. The SQP algorithm is fully described by Gill, Murray and Saunders [8].

2.1.   Constraints and slack variables

Problem NP contains n variables in x. Let m be the number of components of $f(x)$ and $A_L x$ combined. The upper and lower bounds on those terms define the *general constraints* of the problem. SNOPT converts the general constraints to equalities by introducing a set of *slack variables* $s = (s_1, s_2, \ldots, s_m)^T$. 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 $5 \le s_1 \le +\infty$. Problem NP can be written in the equivalent form

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

The general constraints become the equalities $f(x) - s_N = 0$ and $A_L x - s_L = 0$, where s_L and s_N are the *linear* and *nonlinear slacks*.

2.2.   Major iterations

The basic structure of the SQP algorithm 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 nonlinear constraints and the first-order conditions for optimality. At each x_k a QP subproblem is used to generate a search direction toward what will be 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 linearized constraint

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

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

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

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

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

then the QP subproblem can be written as

$$\boxed{\begin{aligned} \text{QP}_k \qquad &\underset{x,s}{\text{minimize}} \quad q(x, x_k) = g_k^T(x - x_k) + \tfrac{1}{2}(x - x_k)^T H_k (x - x_k) \\ &\text{subject to} \ \ Ax - s = b, \quad l \le \begin{pmatrix} x \\ s \end{pmatrix} \le u, \end{aligned}}$$

where $q(x, x_k)$ is a quadratic approximation to a modified Lagrangian function [7].   The matrix H_k is a quasi-Newton approximation to the Hessian of the Lagrangian. A BFGS update is applied after each major iteration. If some of the variables enter the Lagrangian linearly the Hessian will have some zero rows and columns. If the nonlinear variables appear first, then only the leading n_1 rows and columns of the Hessian need be approximated, where n_1 is the number of nonlinear variables.

2.3. Minor iterations

Solving the QP subproblem is itself an iterative procedure. Here, the iterations of the QP solver SQOPT [9] form the *minor* iterations of the SQP method.

SQOPT uses a reduced-Hessian active-set method implemented as a *reduced-gradient method* similar to that in MINOS [15].

At each minor iteration, the constraints $Ax - s = b$ are partitioned into the form

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

where the *basis matrix* B is square and nonsingular, and the matrices S, N are the remaining columns of $\begin{pmatrix} A & -I \end{pmatrix}$. The vectors x_B, x_S, x_N are the associated *basic, superbasic,* and *nonbasic variables* components of (x, s).

The term *active-set method* arises because the nonbasic variables x_N are temporarily frozen at their upper or lower bounds, and their bounds are considered to be *active*. Since the general constraints are satisfied also, the set of active constraints takes the form

$$\begin{pmatrix} B & S & N \\ & & I \end{pmatrix} \begin{pmatrix} x_B \\ x_S \\ x_N \end{pmatrix} = \begin{pmatrix} b \\ x_N \end{pmatrix},$$

where x_N represents the current values of the nonbasic variables. (In practice, nonbasic variables are sometimes frozen at values strictly *between* their bounds.) The reduced-gradient method chooses to move the superbasic variables in a direction that will improve the objective function. The basic variables "tag along" to keep $Ax - s = b$ satisfied, and the nonbasic variables remain unaltered until one of them is chosen to become superbasic.

At a nonoptimal feasible point (x, s) we seek a search direction p such that $(x, s) + p$ remains on the set of active constraints yet improves the QP objective. If the new point is to be feasible, we must have $Bp_B + Sp_S + Np_N = 0$ and $p_N = 0$. Once p_S is specified, p_B is uniquely determined from the system $Bp_B = -Sp_S$. It follows that the superbasic variables may be regarded as independent variables that are free to move in any desired direction. 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 need not be more than one for linear problems.

2.4. The reduced Hessian and reduced gradient

The dependence of p on p_S may be expressed compactly as $p = Zp_S$, where Z is a matrix that spans the null space of the active constraints:

$$Z = P \begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix} \tag{2.1}$$

where P permutes the columns of $\begin{pmatrix} A & -I \end{pmatrix}$ into the order $\begin{pmatrix} B & S & N \end{pmatrix}$. Minimizing $q(x, x_k)$ with respect to p_S now involves a quadratic function of p_S:

$$g^T Zp_S + \tfrac{1}{2} p_S^T Z^T HZp_S,$$

where g and H are expanded forms of g_k and H_k defined for all variables (x, s). This is a quadratic with Hessian $Z^T HZ$ (the *reduced Hessian*) and constant vector $Z^T g$ (the *reduced gradient*). If the reduced Hessian is nonsingular, p_S is computed from the system

$$Z^T HZp_S = -Z^T g. \tag{2.2}$$

The matrix Z is used only as an operator, i.e., it is not stored explicitly. Products of the form Zv or $Z^T g$ are obtained by solving with B or B^T. The package LUSOL [11] is used to maintain sparse LU factors of B as the BSN partition changes. From the definition of Z, we see that the reduced gradient can be computed from

$$B^T \pi = g_B, \qquad Z^T g = g_S - S^T \pi,$$

where π is an estimate of the *dual variables* associated with the m equality constraints $Ax - s = b$, and g_B is the basic part of g.

By analogy with the elements of $Z^T g$, we define a vector of *reduced gradients* (or *reduced costs*) for all variables in (x, s):

$$d = g - \begin{pmatrix} A^T \\ -I \end{pmatrix} \pi, \qquad \text{so that} \qquad d_S = Z^T g.$$

At a feasible point, the reduced gradients for the slacks s are the dual variables π.

The optimality conditions for subproblem QP_k may be written in terms of d. The current point 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 all superbasic variables ($d_S = 0$). In practice, SNOPT requests an *approximate* QP solution $(\widehat{x}_k, \widehat{s}_k, \widehat{\pi}_k)$ with slightly relaxed conditions on d_j.

If $d_S = 0$, no improvement can be made with the current BSN partition, and a nonbasic variable with non-optimal reduced gradient is selected to be added to S. The iteration is then repeated with n_S increased by one. At all stages, if the step $(x, s) + p$ would cause a basic or superbasic variable to violate one of its bounds, a shorter step $(x, s) + \alpha p$ is taken, one of the variables is made nonbasic, and n_S is decreased by one.

The process of computing and testing reduced gradients d_N is known as *pricing* (a term introduced in the context of the simplex method for linear programming). Pricing the jth variable means computing $d_j = g_j - a_j^T \pi$, where a_j is the jth column of $\begin{pmatrix} A & -I \end{pmatrix}$. If there are significantly more variables than general constraints (i.e., $n \gg m$), pricing can be computationally expensive. In this case, a strategy known as *partial pricing* can be used to compute and test only a subset of d_N.

Solving the reduced Hessian system (2.2) is sometimes expensive. With the option `QPSolver Cholesky`, an upper-triangular matrix R is maintained that satisfies $R^T R = Z^T H Z$. Normally, R is computed from $Z^T H Z$ at the start of phase 2 and is then updated as the BSN sets change. For efficiency the dimension of R should not be excessive (say, $n_S \leq 2000$). This is guaranteed if the number of nonlinear variables is "moderate". Other `QPSolver` options are available for problems with many superbasic variables.

2.5. The merit function

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

$$\mathcal{M}(x, s, \pi) = f_0(x) - \pi^T \big(f(x) - s_N\big) + \tfrac{1}{2}\big(f(x) - s_N\big)^T D \big(f(x) - s_N\big), \qquad (2.3)$$

where D is a diagonal matrix of penalty parameters ($D_{ii} \geq 0$), and π now refers to dual variables for the nonlinear constraints in NP. If (x_k, s_k, π_k) denotes the current solution estimate and $(\widehat{x}_k, \widehat{s}_k, \widehat{\pi}_k)$ denotes the QP solution, the linesearch determines a step α_k ($0 < \alpha_k \leq 1$) such that the new point

$$\begin{pmatrix} x_{k+1} \\ s_{k+1} \\ \pi_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ s_k \\ \pi_k \end{pmatrix} + \alpha_k \begin{pmatrix} \widehat{x}_k - x_k \\ \widehat{s}_k - s_k \\ \widehat{\pi}_k - \pi_k \end{pmatrix} \qquad (2.4)$$

gives a *sufficient decrease* in the merit function (2.3). When necessary, the penalties in D are increased by the minimum-norm perturbation that ensures descent for $\mathcal{M}$ [13]. 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 [10, 3].

2.6.   Treatment of constraint infeasibilities

SNOPT makes explicit allowance for infeasible constraints. First, infeasible *linear* constraints are detected by solving the linear program

$$
\begin{array}{ll}
\text{FP} \qquad\qquad & \underset{x,v,w}{\text{minimize}} \quad e^T(v+w) \\[2mm]
& \text{subject to } \; l \le \begin{pmatrix} x \\ A_L x - v + w \end{pmatrix} \le u, \;\; v \ge 0, \;\; w \ge 0,
\end{array}
$$

where e is a vector of ones, and the nonlinear constraint bounds are temporarily excluded from l and u. This is equivalent to minimizing the sum of the general linear constraint violations subject to the bounds on x. (The sum is the ℓ_1-norm of the linear constraint violations. In the linear programming literature, the approach is called *elastic programming*.)

The linear constraints are infeasible if the optimal solution of FP has $v \ne 0$ or $w \ne 0$. SNOPT then terminates without computing the nonlinear functions.

Otherwise, all subsequent iterates satisfy the linear constraints. (Such a strategy allows linear constraints to be used to define a region in which the functions can be safely evaluated.) SNOPT proceeds to solve NP as given, using search directions obtained from the sequence of subproblems QP_k.

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 thereafter solves the problem

$$
\begin{array}{ll}
\text{NP}(\gamma) \qquad\qquad & \underset{x,v,w}{\text{minimize}} \quad f_0(x) + \gamma e^T(v+w) \\[2mm]
& \text{subject to } \; l \le \begin{pmatrix} x \\ f(x) - v + w \\ A_L x \end{pmatrix} \le u, \;\; v \ge 0, \;\; w \ge 0,
\end{array}
$$

where γ is a nonnegative parameter (the *elastic weight*), and $f_0(x) + \gamma e^T(v+w)$ is called a *composite objective* (the ℓ_1 penalty function for the nonlinear constraints).

The value of γ may increase automatically by multiples of 10 if the optimal v and w continue to be nonzero. 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 NP is fundamental to the $\mathrm{S}\ell_1\mathrm{QP}$ algorithm of Fletcher [4]. See also Conn [1].

The initial value of γ is controlled by the optional parameter `Elastic weight` (p. 81).

3.   The snOptA Interface

The snOptA interface accepts a format that allows the constraints and variables to be defined in any order. The optimization problem is assumed to be in the form

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

where the upper and lower bounds are constant, $F(x)$ is a vector of smooth linear and nonlinear constraint functions $\{F_i(x)\}$, and $F_{obj}(x)$ is one of the components of F to be minimized, as specified by the input parameter ObjRow. (The option Maximize specifies that $F_{obj}(x)$ should be maximized instead of minimized.) snOptA reorders the variables and constraints so that the problem is in the form NP (Section 1).

Ideally, the first derivatives (gradients) of F_i should be known and coded by the user. If only some gradients are known, snOptA estimates the missing ones by 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") 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 of just some of the variables. This structure can be exploited by snOptA (see Section 3.3).

3.1.   Subroutines associated with snOptA

snOptA is accessed via the following routines:

snInit (Section 1.6) or snInitF (Section 1.7) must be called before any other snOptA routines.

snSpec (Section 7.4) or snSpecF (Section 7.5) may be called to input a Specs file (a list of run-time options).

snSet, snSeti, snSetr (Section 7.6) may be called to specify a single option.

snGet, snGetc, snGeti, snGetr (Section 7.7) may be called to obtain an option's current value.

snOptA (Section 3.4) is the main solver.

snJac (Section 3.5) may be called to find the sparsity structure of the Jacobian.

usrfun (Section 3.6) is supplied by the user and called by snOptA to define the functions $F_i(x)$ and ideally their gradients. (This routine has a fixed parameter list but may have any convenient name. It is passed to snOptA as a parameter.)

snMemA (Section 3.8) computes the size of the workspace arrays cw, iw, rw required for given problem dimensions. Intended for Fortran 90 and C drivers that reallocate workspace if necessary.

3.2. Getting Started

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

$$
\begin{aligned}
\underset{x_1, x_2}{\text{minimize}} \quad & x_2 \\
\text{subject to} \quad & x_1^2 + 4x_2^2 \leq 4, \\
& (x_1 - 2)^2 + x_2^2 \leq 5, \\
& x_1 \geq 0.
\end{aligned}
\tag{3.1}
$$

In the format of problem NPA, we have $l_x \leq x \leq u_x$ and $l_F \leq F(x) \leq u_F$ as follows:

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

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

Let $G(x)$ be the Jacobian matrix of partial derivatives, so that $G_{ij}(x) = \partial F_i(x)/\partial x_j$ gives the gradients of $F_i(x)$ as the ith row of G:

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

Now we must provide **snOptA** the following information:

1. *The index of the objective row.* Here, **ObjRow** $= 1$.

2. *The upper and lower bounds on x.* The vectors l_x and u_x are input as arrays:

```
infBnd   =   1.1d+20
xlow(1)  =   0.0
xlow(2)  =  -infBnd
xupp(1)  =   infBnd
xupp(2)  =   infBnd
```

   where **infBnd** represents "infinity". It must be at least as large as the **Infinite Bound** (default value 10^{20}).

3. *The upper and lower bounds on $F(x)$.* The vectors l_F and u_F are also in arrays:

```
Flow(1)  =  -infBnd
Flow(2)  =  -infBnd
Flow(3)  =  -infBnd
Fupp(1)  =   infBnd
Fupp(2)  =   4.0
Fupp(3)  =   5.0
```

   Note that the objective row must have infinite bounds to make it a "free" row.

4. *A subroutine* `usrfun` *that computes* $F(x)$. For this example, `usrfun` would 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
```

5. *As many elements of* $G(x)$ *as possible.* Ideally, `Derivative option` 1 should be specified and subroutine `usrfun` should compute all derivatives of $F(x)$. Elements that are identically zero may be omitted.

   Here, `usrfun` could include the following additional assignments. They compute all G_{ij} *row-wise*, excluding $G_{11} \equiv 0$:

```
G(1) =                     1.0           ! G(1,2)
G(2) =   2.0* x(1)                       ! G(2,1)
G(3) =                     8.0*x(2)      ! G(2,2)
G(4) =   2.0*(x(1) - 2.0)                ! G(3,1)
G(5) =                     2.0*x(2)      ! G(3,2)
```

   The elements of G may be stored in any order (row-wise, column-wise, or more randomly). The ordering implies a list of coordinates $\{(i,j)\}$, in this case

$$(1,2) \ (2,1) \ (2,2) \ (3,1) \ (3,2).$$

   This list *must match* the integer arrays `iGfun` and `jGvar` below.

   If `Derivative option` 0 is specified, some or all of the gradient elements need not be assigned values in `G`. `snOptA` will estimate them by finite differences.

6. *The pattern of nonzero elements of the Jacobian* G. This is a list of coordinates $\{(i,j)\}$ of the nonzero elements G_{ij}, held in two integer arrays. If $i = $ `iGfun(k)` and $j = $ `jGvar(k)`, then `G(k)` holds G_{ij}. In the example, the program calling `snOptA` would include the following:

```
iGfun(1) = 1   ! row coordinate of G(1) =  1.0
jGvar(1) = 2   ! col coordinate of G(1)

iGfun(2) = 2   ! row coordinate of G(2) =  2.0* x(1)
jGvar(2) = 1   ! col coordinate of G(2)

iGfun(3) = 2   ! row coordinate of G(3) =  8.0* x(2)
jGvar(3) = 2   ! col coordinate of G(3)

iGfun(4) = 3   ! row coordinate of G(4) =  2.0*(x(1) - 2.0)
jGvar(4) = 1   ! col coordinate of G(4)

iGfun(5) = 3   ! row coordinate of G(5) =  2.0* x(2)
jGvar(5) = 2   ! col coordinate of G(5)
```

   If `Derivative option` 0 is specified, `iGfun` and `jGvar` may be defined *automatically* by subroutine `snJac` (p. 24).

3.3. Exploiting problem structure

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

Suppose that each 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 `nF`$\times$`n` Jacobian of $F(x)$ is the sum of two sparse matrices of the same size: $F'(x) = G(x) + A$, where $G(x) = f'(x)$ and A is the matrix with elements $\{A_{ij}\}$. The two matrices must be *non-overlapping* in the sense that every element of the Jacobian $F'(x) = G(x) + A$ is an element of $G(x)$ or an element of A, but *not both*.

For example, the function

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

can be written as

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

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

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

The nonzero elements of A and G are provided to `snOptA` in coordinate form. The elements of A are entered as triples (i, j, A_{ij}) in the arrays `iAfun`, `jAvar`, `A`. The sparsity pattern of G is entered as pairs (i, j) in the arrays `iGfun`, `jGvar`. The corresponding entries G_{ij} (any that are known) are assigned to appropriate array elements `G(k)` in the user's subroutine `usrfun`.

The elements of A and G may be stored in any order. Duplicate entries are ignored. As mentioned, `iGfun` and `jGvar` may be defined automatically by subroutine `snJac` (p.) when `Derivative option` 0 is specified and `usrfun` does not provide any gradients.

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 function routine `usrfun` and all other routines.

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

 external
&      usrfun
 integer
&      INFO, 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
&      Prob*8, cu(lencu)*8, cw(lencw)*8,
&      Fnames(nFname)*8, xnames(nxname)*8
```

On entry:

`Start`   is an integer that specifies how a starting point is to be obtained.

> `Start = 0`   (Cold start) requests that the CRASH procedure be used, unless an
> `Old basis`, `Insert`, or `Load` file is specified.
>
> `Start = 1`   is the same as `0` but more meaningful when a basis file is given.
>
> `Start = 2`   (Warm start) means that `xstate` and `Fstate` define a valid starting
> point (perhaps from an earlier call, though not necessarily).

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

> `nF` is the dimension of the function vector `F`. *Since F includes the objective (if any)
> and all the general linear and nonlinear constraints, there must be 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. ($\mathtt{n} > 0$)

iAfun(lenA), jAvar(lenA), A(lenA) define the coordinates (i, j) and values A_{ij} of the nonzero elements of the linear part A of the function $F(x) = f(x) + Ax$. The first **neA** entries are valid. $(0 \leq \mathtt{neA} \leq \mathtt{lenA}$ and $\mathtt{lenA} \geq 1)$

The entries may define the elements of A in any order.

lenA   is the dimension of the arrays **iAfun**, **jAvar**, **A** that hold (i, j, A_{ij}). $(\mathtt{lenA} \geq 1)$

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

iGfun(lenG), jGvar(lenG) define the coordinates (i, j) of G_{ij}, the nonzero elements of the nonlinear part of the derivatives $G(x) + A$ of the function $F(x) = f(x) + Ax$. The first **neG** entries are valid. $(0 \leq \mathtt{neG} \leq \mathtt{lenG}$ and $\mathtt{lenG} \geq 1)$

The entries may define the elements of G in any order, but subroutine **usrfun** must define the values of **G** in exactly the same order.

lenG   is the dimension of the coordinate arrays **iGfun**, **jGvar**. $(\mathtt{lenG} \geq 1)$

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

nxname, nFname give the number of variable and constraint names provided in the character arrays **xname** and **Fname**. If $\mathtt{nxname} = 1$ and $\mathtt{nFname} = 1$, *no* names are provided. (Generic names will be used in the printed solution.) Otherwise, the values must be $\mathtt{nxname} = \mathtt{n}$ and $\mathtt{nFname} = \mathtt{nF}$ and all names must be provided in **xnames** and **Fnames** (below).

ObjAdd is a constant that will be added to the objective row **F(Objrow)** for printing purposes. Typically, $\mathtt{ObjAdd} = \mathtt{0.0d+0}$.

ObjRow says which row of $F(x)$ is to act as the objective function. If there is no such row, set $\mathtt{ObjRow} = 0$. Then **snOptA** will seek a *feasible point* such that $l_F \leq F(x) \leq u_F$ and $l_x \leq x \leq u_x$. $(0 \leq \mathtt{ObjRow} \leq \mathtt{nF})$

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 (Section 3.6) is the name of a subroutine that calculates the nonlinear portion $f(x)$ of the vector of problem functions $F(x) = f(x) + Ax$, and (optionally) its Jacobian $G(x)$ for a given vector x. **usrfun** 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 non-existent bounds, set $\mathtt{xlow}(j) \leq -\mathtt{infBnd}$ or $\mathtt{xupp}(j) \geq \mathtt{infBnd}$, where **infBnd** is the **Infinite Bound size** (default value 10^{20}).
To fix the jth variable at $x_j = \beta$, set $\mathtt{xlow}(j) = \mathtt{xupp}(j) = \beta$ (with $|\beta| < \mathtt{infBnd}$).

Flow(nF), Fupp(nF) contain the lower and upper bounds l_F and u_F on $F(x)$.

To specify non-existent bounds, set $\mathtt{Flow}(j) \leq -\mathtt{infBnd}$ or $\mathtt{Fupp}(j) \geq \mathtt{infBnd}$.
To make the ith constraint an *equality*, $F_i(x) = \beta$, set $\mathtt{Flow}(i) = \mathtt{Fupp}(i) = \beta$.

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

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

xstate(n) usually contains a set of initial states for each variable x.

1. For Cold starts (`Start` = 0 or 1 with no basis file provided), `xstate` and `x` must be defined.

   If there is no wish to provide special information, you may set $\texttt{xstate}(j) = 0$, $\texttt{x}(j) = 0.0$ for all $j = 1:\texttt{n}$. All variables will be eligible for the initial basis.

   Less trivially, to say that the optimal value of x_j will probably be one of its bounds, set $\texttt{xstate}(j) = 4$ and $\texttt{x}(j) = \texttt{xlow}(j)$ or $\texttt{xstate}(j) = 5$ and $\texttt{x}(j) = \texttt{xupp}(j)$ as appropriate.

   A CRASH procedure is used to select an initial basis. The initial basis matrix will be triangular (ignoring certain small entries in each column). The values $\texttt{xstate}(j) = 0,\ 1,\ 2,\ 3,\ 4,\ 5$ have the following meaning:

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

   After CRASH, variables for which $\texttt{xstate}(j) = 2$ are made superbasic. Other variables not selected for the basis are made nonbasic at the value $\texttt{x}(j)$ (or the closest value inside their bounds). See the description of `xstate` below (on exit).

2. If `Start` = 0 or 1 and a basis file *is* provided, `xstate` and `x` need not be set.

3. For Warm starts (`Start` = 2), `xstate` and `x` must be defined, and all $\texttt{xstate}(j)$ must be 0, 1, 2 or 3 (perhaps from a previous call).

`F(nF)` sometimes contains a set of initial values for the functions $F(x)$. See next.

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

1. For Cold starts (`Start` = 0 or 1 with no basis file provided), `Fstate` and `F` must be defined.

   If there is no wish to provide special information, you may set $\texttt{Fstate}(i) = 0$, $\texttt{F}(i) = 0.0$ for all $i = 1:\texttt{nF}$. All rows will be eligible for the initial basis.

   Less trivially, to say that the optimal value of row i will probably be one of its bounds, set $\texttt{Fstate}(i) = 4$ and $\texttt{F}(i) = \texttt{Flow}(i)$ or $\texttt{Fstate}(i) = 5$ and $\texttt{F}(i) = \texttt{Fupp}(i)$ as appropriate.

   During CRASH, the values $\texttt{Fstate}(i) = 0, 1, 2, 3, 4, 5$ have the following meaning:

$\texttt{Fstate}(i)$	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 $\texttt{Fstate}(i) = 2$ are made superbasic. Other rows not selected for the basis are made nonbasic at the value $\texttt{F}(i)$ (or the closest value inside their bounds). See the description of `Fstate` below (on exit).

2. If `Start` = 0 or 1 and a basis file *is* provided, `Fstate` and `F` need not be set.

3. For Warm starts (`Start` = 2), $\texttt{Fstate}(1:\texttt{nF})$ must be 0, 1, 2 or 3 (perhaps from a previous call).

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

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 `usrfun` doesn't reference these parameters, you may use any arrays of the appropriate type, such as `cw`, `iw`, `rw` (see next paragraph). Conversely, 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`. Their lengths `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 $500 + \text{n} + \text{nF}$; 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 factorization. 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 `snMemA` (Section 3.8), or from a preliminary run of `snOptA` with `lencw` $=$ `leniw` $=$ `lenrw` $= 500$. See `mincw`, `miniw`, `minrw` below (on exit).

On exit:

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

`xstate(n)` gives the final state of the variables as follows:

`xstate`(j)	State of variable j	Usual value of $\mathtt{x}(j)$
0	nonbasic	$\mathtt{xlow}(j)$
1	nonbasic	$\mathtt{xupp}(j)$
2	superbasic	Between $\mathtt{xlow}(j)$ and $\mathtt{xupp}(j)$
3	basic	Between $\mathtt{xlow}(j)$ and $\mathtt{xupp}(j)$

     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 $\mathtt{x}(j)$ 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).

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

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

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

INFO   reports the result of the call to `snOptA`. Here is a summary of possible values. Further details are in Section 8.6.

Finished successfully
1   optimality conditions satisfied
2   feasible point found
3   requested accuracy could not be achieved
5   elastic objective minimized
6   elastic infeasibilities minimized

The problem appears to be infeasible
11   infeasible linear constraints
12   infeasible linear equality constraints
13   nonlinear infeasibilities minimized
14   linear infeasibilities minimized
15   infeasible linear constraints in QP subproblem
16   infeasible nonelastic constraints

The problem appears to be unbounded
21   unbounded objective
22   constraint violation limit reached

Resource limit error
31   iteration limit reached
32   major iteration limit reached
33   the superbasics limit is too small
34   time limit reached

Terminated after numerical difficulties
41   current point cannot be improved
42   singular basis
43   cannot satisfy the general constraints
44   ill-conditioned null-space basis
45   unable to compute acceptable LU factors

Error in the user-supplied functions
51   incorrect objective derivatives
52   incorrect constraint derivatives
56   irregular or badly scaled problem functions

Undefined user-supplied functions
61   undefined function at the first feasible point
62   undefined function at the initial point
63   unable to proceed into undefined region

User requested termination
71   terminated during function evaluation
74   terminated from monitor routine

Insufficient storage allocated
81   work arrays must have at least 500 elements
82   not enough character storage
83   not enough integer storage
84   not enough real storage

> *Input arguments out of range*
> 91   invalid input argument
> 92   basis file dimensions do not match this problem
>
> *System error*
> 141   wrong number of basic variables
> 142   error in basis package

mincw, miniw, minrw say how much character, integer, and real storage is needed to solve the problem. If `Print level` > 0, these values are printed. If `snOptA` terminates because of insufficient storage (`INFO` $= 82$, 83 or 84), the values may be used to define better values of `lencw`, `leniw` or `lenrw`.

If `INFO` $= 82$, the work array `cw(lencw)` is too small. `snOptA` may be called again with `lencw = mincw`.

If `INFO` $= 83$ or 84, 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 because it is not certain how much storage the basis factorization needs.)

nS   is the final number of superbasic variables.

nInf, sInf give the number and the sum of the infeasibilities of constraints that lie outside one of their bounds by more than the `Minor feasibility tolerance` *before the solution is unscaled.*

If any *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 variables and linear constraints lying outside their 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 bounds by more than the `Minor feasibility tolerance`. Again this is *before the solution is unscaled.*

3.5. Subroutine `snJac`

If you set `Derivative option` 0 and your function routine `usrfun` provides *none* of the derivatives in `G`, you may call subroutine `snJac` to determine the input arrays `iAfun`, `jAvar`, `A`, `iGfun`, `jGvar` for `snOptA`. These define the pattern of nonzeros in the Jacobian matrix. A typical sequence of calls is

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

(The `Derivative option` could also be set in the Specs file.)

Subroutine `snJac` determines the sparsity pattern for the Jacobian and identifies the constant elements automatically. To do so, `snJac` approximates $F'(x)$ at three random perturbations of the given 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. Since the random points are not chosen close together, the heuristic will correctly classify the Jacobian elements in the vast majority of cases.

In general, `snJac` finds that the Jacobian can be permuted to the form $\begin{pmatrix} G(x) & A_3 \\ A_2 & A_4 \end{pmatrix}$, where A_2, A_3, A_4 are constant. Note that $G(x)$ might contain elements that are also constant, but `snJac` must classify them as nonlinear. (This is because `snOptA` "removes" linear variables from the calculation of F by setting them to zero before calling `usrfun`.) A knowledgeable user would be able to move such elements from `usrfun`'s $F(x)$ and enter them as part of `iAfun`, `jAvar`, `A` for `snOptA`.

```
 subroutine snJac
&    ( INFO, 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
&      INFO, 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
&      cu(lencu)*8, cw(lencw)*8
```

On entry:

Most arguments are identical to those of `snOptA`.

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

   `lenA` should be an *overestimate* of the number of elements in the linear part of the Jacobian. The value `lenA = nF × 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}). (`lenG` ≥ 1).

        `lenG` should be an overestimate of the number of elements in the nonlinear part of
        the Jacobian. The value `lenG` = `nF` $\times$ `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 first `neA` entries define the row and column indices i = `iAfun(k)` and j =
        `jAvar(k)` and elements A_{ij} = `A(k)` for `k` $= 1$:`neA`.

`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 G_{ij}, the nonzero elements of
        the nonlinear part of the derivatives $G(x) + A$ of the function $F(x) = f(x) + Ax$.
        The actual elements of G are assigned in subroutine `usrfun`.

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

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

> *Finished successfully*
> 102   satisfactory coordinates were found
>
> *Undefined user-supplied functions*
> 61   user indicates that the functions are undefined at the initial point
>
> *User requested termination*
> 71   user requested termination by returning `mode` < -1 from `usrfun`
>
> *Insufficient storage allocated*
> 82   not enough 8-character workspace to solve the problem
> 83   not enough integer workspace to solve the problem
> 84   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 `INFO` = 82, 83 or 84, these values may be used
        to define better values of `lencw`, `leniw` or `lenrw`.

        If `INFO` = 82, the work array `cw(lencw)` was too small. `snJac` may be called again
        with `lencw` = `mincw`.

        If `INFO` = 83 or 84, 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`

Your own version of subroutine `usrfun` is where you define the nonlinear portion $f(x)$ of the problem functions $F(x) = f(x) + Ax$, along with its gradient elements $G_{ij}(x) = \partial f_i(x)/\partial x_j$. This subroutine is passed to `snOptA` as the external parameter `usrfun`. (A dummy subroutine is needed even if $f(x) \equiv 0$ and all functions are linear.)

In general, `usrfun` 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 option` 1.

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 3.7).

In practice it is often convenient *not* to code gradients. `snOptA` is able to estimate them by finite differences, using a call to `usrfun` for each variable x_j for which some $\partial f_i(x)/\partial x_j$ needs 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 gradients as you like.* This option is implemented as follows. Just before the function routine is called, each element of the derivative array `G` is initialized to a specific value. On exit, any element retaining that value must be estimated by finite differences.

Some rules of thumb follow.

1. For maximum reliability, compute all gradients.

2. If the gradients 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 gradients if `needG` $= 0$ on entry to `usrfun`.)

3. If not all gradients are known, you must specify `Derivative option` 0. 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 `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 `usrfun` is being developed, use the `Verify` option to check the computation of gradients that are supposedly known.

7. `usrfun` is not called until the linear constraints and bounds on x are satisfied. This helps confine x to regions where the functions $f_i(x)$ 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 some of the functions are undefined. The linesearch will shorten the step and try again.

9. Set `Status` < -1 if you want `snOptA` to stop.

```
      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
   &      cu(lencu)*8
```

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 $= 2 + \text{INFO}/10$, where INFO is the integer returned by snOptA (see p.). 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;
>> if Status $= 5$, an iterations limit was reached.

> If the functions are expensive to evaluate, it may be desirable to do nothing on the last call. The first executable statement could be if (Status .ge. 2) return.

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, nF, f(nF)   concern the calculation of $f(x)$.
nF is the length of the full vector $F(x) = f(x) + Ax$ as defined in the call to snOptA.
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 $F(x)$ 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 simplify the code, you may ignore the value of needf and compute $f(x)$ on every entry to usrfun.

needf may also be ignored with Derivative linesearch and Derivative option 1 (both defaults). In this case, needf is always 1, and f must always be assigned.

needG, lenG, G(lenG)   concern the calculation of the derivatives of the function $f(x)$. lenG is the length of the coordinate arrays iGvar and jGfun in the call to snOptA. needG indicates if G must be assigned during this call of 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. For each $k = 1$:lenG, the value of G(k) should be $\partial f_i(x)/\partial x_j$, where $i = $ iGfun(k), $j = $ jGvar(k).

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 routine 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 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 to evaluate f or its gradients at the current x. (For example, the problem functions may not be defined there.)

During the linesearch, $f(x)$ is evaluated at points $x = x_k + \alpha p_k$ for various steplengths α, where $f(x_k)$ has already been evaluated satisfactorily. For any such x, if you set Status $= -1$, snOptA will reduce α and evaluate f again (closer to x_k, where it is more likely to be defined).

If for some reason you wish to terminate the current problem, set Status ≤ -2.

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.

3.7.   Example `usrfun`

Here we give subroutine `usrfun` for the problem

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

plus bounds on x. This problem has 4 variables, a nonlinear objective function, 2 nonlinear constraints, and 1 linear constraint. The vector of functions is

$$
F(x) = \begin{pmatrix} 3x_1 + 5x_2 + (x_1 + x_3 + x_4)^2 \\ x_1 + x_3^2 + x_4^2 \\ 2x_3 + 4x_4 \\ x_2 + x_4^4 \end{pmatrix}.
$$

The Jacobian of F is the matrix

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

We can write $F(x) = f(x) + Ax$ and $F'(x) = f'(x) + A = G(x) + A$, where

$$
f(x) = \begin{pmatrix} 3x_1 + (x_1 + x_3 + x_4)^2 \\ x_3^2 + x_4^2 \\ 0 \\ x_4^4 \end{pmatrix}, \qquad
A = \begin{pmatrix} 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 0 & 0 \end{pmatrix},
$$

and G is the non-overlapping matrix

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

The calling program must assign many parameters for input to `snOptA`, including

$$
\mathtt{nF} = 4, \quad \mathtt{n} = 4, \quad \mathtt{ObjRow} = 1, \quad \mathtt{neA} = 5, \quad \mathtt{neG} = 6 \quad \mathtt{lenG} = 6 \ \ (\geq \mathtt{neG}).
$$

Some of these parameters are passed to subroutine `usrfun` (next page). Note that `usrfun` works only with the *nonlinear variables* (x_1, x_3, x_4) that appear in f and G. The array `f` is set to $f(x)$, *excluding the term Ax* (which is evaluated separately by `snOptA`). The array `G` is set to all nonzero entries of $G(x)$, *excluding the matrix A*.

For illustration, we test `needF` and `needG` to economize on function and gradient evaluations (even though they are cheap here). Note that `Nonderivative 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
     &    cu(lencu)*8

*     ==================================================================
*     Computes the nonlinear objective and constraint terms for the toy
*     problem featured in the SNOPT user's guide.
*     ==================================================================
      integer
     &    neG, Obj, Out
      double precision
     &    sum, x1, x3, x4
*     ------------------------------------------------------------------
      Out = 15                  ! Output unit number
      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

      x1  = x(1)
      x3  = x(3)
      x4  = x(4)
      sum = x1 + x3 + x4

      if (needF .gt. 0) then
         f(Obj) =   3.0d0*x1 + sum**2
         f(2)   =   x3**2 + x4**2
*!!      f(3)   =                      ! Linear constraint omitted!
         f(4)   =   x4**4
      end if

      neG = 0
      if (needG .gt. 0) then
         neG        =  neG + 1
         G(neG)     =  2.0d0*sum + 3.0d0
*!!      iGfun(neG) =  Obj             ! Not used, but included for clarity!
*!!      jGvar(neG) =  1               ! Not used
         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*x3
*         iGfun(neG)  =   2
*         jGvar(neG)  =   3
          neG         =   neG + 1
          G(neG)      =   2.0d0*x4
*         iGfun(neG)  =   2
*         jGvar(neG)  =   4
          neG         =   neG + 1
          G(neG)      =   4.0d0*x4**3
*         iGfun(neG)  =   4
*         jGvar(neG)  =   4
       end if

       end ! subroutine usrfun
```

3.8. Subroutine `snMemA`

This routine estimates the size of the workspace arrays `cw`, `iw`, `rw` required to solve an optimization problem of given dimensions. `snMemA` is not strictly needed in f77 because all workspace must be defined explicitly in the driver program at compile time. It is available for users wishing to allocate storage dynamically in f90 or C.

The actual storage required also depends on the values of the following options:

>   `Hessian full memory`
> *or* `Hessian limited memory` and `Hessian updates`
>   `Reduced Hessian dimension`
>   `Superbasics limit`

If these options have not been set, default values are assumed. Ideally the correct values should be set *before* the call to `snMemA`.

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

  integer
&      INFO, lencw, leniw, lenrw, mincw, miniw, minrw, n, neA, neG,
&      nF, nFname, nxname, iw(leniw)
  double precision
&      rw(lenrw)
  character
&      cw(lencw)*8
```

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). For a sequence of problems, `snMemA` may be called once with overestimates of these quantities.

On entry:

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

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

On exit:

`INFO`   reports the result of the call to `snMemA`. Further details are in Section 8.6.

                  Insufficient storage allocated
      81   work arrays must have at least 500 elements

                  Finished successfully
    104   memory requirements estimated

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

To use `snMemA`, the first step is to allocate the work arrays. These may be temporary arrays `tmpcw`, `tmpiw`, `tmprw` (say) or the `snOptA` arrays `cw`, `iw`, `rw`, which will be reallocated

after the storage limits are known. Here we illustrate the use of `snMemA` using the same arrays for `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`, `rw` arrays in order to retain the options.

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

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

As with all SNOPT routines, `snInit` or `snInitF` *must* be called to set default options:

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

call snInitF
&   ( printfile, summaryfile, iPrint, iSumm,
&     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

This installs `ltmpcw`, `ltmpiw`, `ltmprw` as the default internal upper limits on the `snOptA` workspace (see `Total real workspace` p.). They are used to compute the boundaries of any user-defined workspace in `cw`, `iw`, or `rw`.

The next step is to call `snMemA` to obtain `mincw`, `miniw`, `minrw` as estimates of the storage needed by `snOptA`:

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

The output values of `mincw`, `miniw`, `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 or C to allocate the final work arrays for the problem.

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

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

An alternative way is to call `snInit` (or `snInitF`) again with arguments `lencw`, `leniw`, `lenrw`:

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

However, this has the twin effects of resetting all options to their default values and reprinting the SNOPT banner (unless `iPrint` $= 0$ and `iSumm` $= 0$ are set for the Print and Summary files).

4.   The snOptB Interface

snOptB is the "base" user interface with arguments identical to versions of SNOPT up to Version 5.3-4. The optimization problem is assumed to be in the form NP (p. 4) with the data ordered so that nonlinear constraints and variables come first. A typical invocation is

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

Subroutine snOpt is identical to snOptB, and the analogous call provides compatibility with previous versions of SNOPT:

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

4.1.   Subroutines used by snOptB

snOptB is accessed via the following routines:

---

snInit (Section 1.6) or snInitF (Section 1.7) must be called before any other snOptB routines.

snSpec (Section 7.4) or snSpecF (Section 7.5) may be called to input a Specs file (a list of run-time options).

snSet, snSeti, snSetr (Section 7.6) may be called to specify a single option.

snGet, snGetc, snGeti, snGetr (Section 7.7) may be called to obtain an option's current value.

snOptB, snOpt (Section 4.4) are the main solvers. They are identical.

funcon, funobj (Sections 4.6–4.7) are supplied by the user and called by snOptB. They define the constraint and objective functions and ideally their gradients. (They have a fixed parameter list but may have any convenient name. They are passed to snOptB and snOpt as parameters.)

snMemB (Section 4.10) computes the size of the workspace arrays cw, iw, rw required for given problem dimensions. Intended for Fortran 90 and C drivers that reallocate workspace if necessary.

4.2.   Identifying structure in the objective and constraints

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

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

with bounds $w \geq 0$, $z \geq 0$. This problem has several characteristics that can be exploited:

- The objective function is the sum of a *nonlinear* function of the three variables $x' = (u, v, w)$ 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 $y'' = (w, z)$.

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 $f_0(u, v, w) = (u + v + w)^2$ to include only the nonlinear terms. The variables $x' = (u, v, w)$ are known as *nonlinear objective variables*, and their dimension n_1' is specified by the `snOptB` input parameter `nnObj` $(= 3$ here$)$. The linear part $3w + 5z$ of the objective is treated as an additional linear constraint whose row index is specified by the input parameter `iObj`. 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. Choosing `iObj` $= 4$, we think of the problem as

$$\begin{array}{ll} \underset{u,v,w,z,s_4}{\text{minimize}} & (u + v + w)^2 + \ s_4 \\ \text{subject to} & u^2 + v^2 + \ w \qquad\quad = 2 \\ & \qquad\quad v^4 \qquad + \ z = 4 \\ & 2u + 4v \qquad\qquad \geq 0 \\ & \qquad\qquad 3w + 5z = s_4 \end{array}$$

with bounds $w \geq 0$, $z \geq 0$, $-\infty \leq s_4 \leq \infty$, where s_4 is treated *implicitly* as the 4th slack variable. (Internal to `snOptB`, slacks s_1, s_2 and s_3 are associated with the first three constraints, with s_1 and s_2 fixed at 0 and $0 \leq s_3 \leq \infty$.)

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 $f_2(u, v) = v^4$. The number of nonlinear constraints (the dimension of f) is specified by the input parameter `nnCon` $= 2$. The variables $x'' = (u, v)$ are known as *nonlinear Jacobian variables*, with dimension n_1'' specified by `nnJac` $= 2$. Thus, the constraint functions and the linear part of the objective have the form

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

where x'' is the first `nnJac` variables, $f(x'')$ is defined by subroutine `funcon`, and y'' contains the remaining variables, i.e., $y'' = (w, z)$ in the example. The full Jacobian is of the form

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

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

The inequalities $l_1 \leq f(x'') + A_3 y'' \leq u_1$ and $l_2 \leq A_2 x + A_4 y'' \leq u_2$ 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 $f'(x'')$ smaller.

Together the nonlinear objective and nonlinear Jacobian variables comprise the *nonlinear variables*. The number of nonlinear variables n_1 is therefore the *larger* of the dimensions of x' and x'', i.e., $n_1 = \max\{n_1', n_1''\}$ $(= \max(\text{nnObj}, \text{nnJac}))$.

4.3. Problem dimensions

The following picture illustrates the problem structure just described:
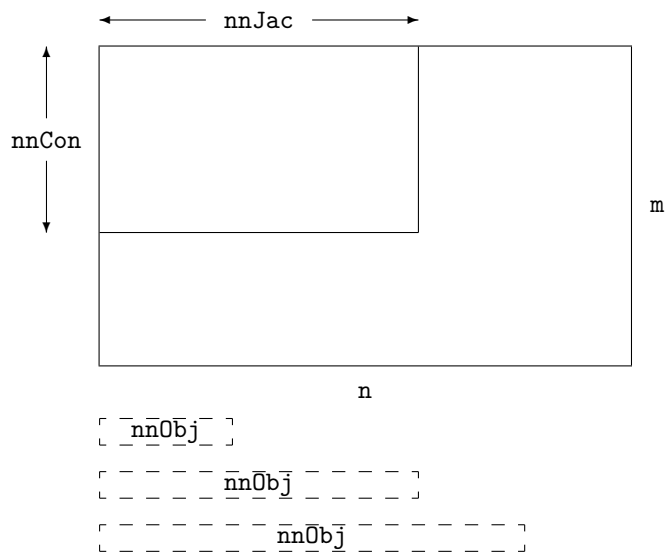
The dimensions are all input parameters to subroutine `snOptB` (see the next section). For linear programs, `nnCon`, `nnJac`, `nnObj` are all zero. If a linear objective term exists, `iObj` points to one of the bottom rows ($\texttt{nnCon} < \texttt{iObj} \leq \texttt{m}$).

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, $\texttt{nnObj} \leq \texttt{nnJac}$ or vice versa.

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

4.4. Subroutine snOptB

Problem NP 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 `funcon` and `funobj`.

```
      subroutine snOptB
&    ( Start, m, n, neA, nName,
&      nnCon, nnObj, nnJac,
&      iObj, ObjAdd, Prob,
&      funcon, funobj,
&      Acol, indA, locA, bl, bu, Names,
&      hs, x, pi, rc,
&      INFO, mincw, miniw, minrw,
&      nS, nInf, sInf, Obj,
&      cu, lencu, iu, leniu, ru, lenru,
&      cw, lencw, iw, leniw, rw, lenrw )

 external
&      funcon, funobj
 integer
&      INFO, iObj, lencu, lencw, leniu, leniw, lenru, lenrw, m,
&      mincw, miniw, minrw, n, neA, nInf, nName, nnCon, nnJac, nnObj,
&      nS, hs(n+m), indA(neA), iu(leniu), iw(leniw), locA(n+1)
 double precision
&      Obj, ObjAdd, sInf, Acol(neA), bl(n+m), bu(n+m), pi(m),
&      rc(n+m), ru(lenru), rw(lenrw), x(n+m)
 character*(*)
&      Start
 character
&      Prob*8, cu(lencu)*8, cw(lencw)*8, Names(nName)*8
```

On entry:

Start   is a character string that specifies how a starting point is to be obtained.

  'Cold'   (Cold start) requests that the CRASH procedure be used, unless an
                `Old basis`, `Insert`, or `Load` file is specified.

  'Basis file' is the same as `Start = 'Cold'` but more meaningful when a basis
                file is given.

  'Warm'   (Warm start) means that `hs` defines a valid starting point (probably
                from an earlier call, though not necessarily).

m   is m, the number of general constraints. This is the number of rows in the full
        matrix A in (4.2). ($m > 0$)

        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 `Acol`, `indA`, `locA` below.

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

neA   is the number of nonzero entries in A (including the Jacobian for any nonlinear constraints). $(\texttt{neA} > 0)$

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

nnCon   is m_1, the number of nonlinear constraints. $(\texttt{nnCon} \geq 0)$

nnObj   is n_1', the number of nonlinear objective variables. $(\texttt{nnObj} \geq 0)$

nnJac   is n_1'', the number of nonlinear Jacobian variables. If $\texttt{nnCon} = 0$, then $\texttt{nnJac} = 0$. If $\texttt{nnCon} > 0$, then $\texttt{nnJac} > 0$.

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

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

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 Section 4.6.

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 Section 4.7.

Acol(neA), indA(neA), 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 $\texttt{Acol}(k)$, $\texttt{indA}(k)$ contains a matrix element and its corresponding row index. The array locA(*) is a set of pointers to the beginning of each column of A within Acol(*) and indA(*). Thus for $j = 1\!:\!n$, the entries of column j are held in $\texttt{Acol}(k\!:\!l)$ and their corresponding row indices are in $\texttt{indA}(k\!:\!l)$, where $k = \texttt{locA}(j)$ and $l = \texttt{locA}(j+1) - 1$,

   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 Derivative level $= 2$ 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 $\texttt{locA}(1) = 1$ and $\texttt{locA}(n+1) = \texttt{neA} + 1$.

   2. The Jacobian $f'(x)$ forms the top left corner of Acol and indA (see Section 4.2). If a Jacobian column j $(1 \leq j \leq \texttt{nnJac})$ contains any entries $\texttt{Acol}(k)$, $\texttt{indA}(k)$ associated with nonlinear constraints $(1 \leq \texttt{indA}(k) \leq \texttt{nnCon})$, 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 `gCon(*)` in the same order (except `gCon` does not contain elements in linear constraints).

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 $j = 2 : n + 1$. This row is made "free" by setting its bounds to be `bl(n+1) = −infBnd` and `bu(n+1) = infBnd`, where `infBnd` is typically `1.0e+20` (see the next paragraph).

`bl(n+m)`, `bu(n+m)` contain the lower and upper bounds l and u on the variables and slacks (x, s). The first n entries of `bl`, `bu`, `hs` and `x` refer to the variables x. The last m entries refer to the slacks s.

To specify non-existent bounds, set $\mathtt{bl}(j) \leq -\mathtt{infBnd}$ or $\mathtt{bu}(j) \geq \mathtt{infBnd}$, where `infBnd` is the `Infinite Bound size` (default value 10^{20}).
To fix the jth variable at $x_j = \beta$, set $\mathtt{bl}(j) = \mathtt{bu}(j) = \beta$ (with $|\beta| < \mathtt{infBnd}$).
To make the ith constraint an *equality*, $s_i = \beta$, set $\mathtt{bl}(n + i) = \mathtt{bu}(n + i) = \beta$.

For the data to be meaningful, it is required that $\mathtt{bl}(j) \leq \mathtt{bu}(j)$ 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 $\mathtt{nName} = n + m$, $\mathtt{Names}(j)$ should contain the 8-character name of the jth variable ($j = 1 : n + m$). If $j = n + i$, the jth variable is the ith row.

`hs(n+m)` usually contains a set of initial states for each variable x, or for each variable and slack (x, s). See next.

`x(n+m)` usually contains a set of initial values for x or (x, s).

1. For Cold starts (`Start = 'Cold'` or `'Basis file'` with no basis file provided), the first n elements of `hs` and `x` must be defined.

   If there is no wish to provide special information, you may set $\mathtt{hs}(j) = 0$, $\mathtt{x}(j) = 0.0$ 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 $\mathtt{hs}(j) = 4$ and $\mathtt{x}(j) = \mathtt{bl}(j)$ or $\mathtt{hs}(j) = 5$ and $\mathtt{x}(j) = \mathtt{bu}(j)$ as appropriate.

   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 $\mathtt{hs}(j) = 0, 1, 2, 3, 4, 5$ have the following meaning:

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

   After CRASH, variables for which $\mathtt{hs}(j) = 2$ are made superbasic. Other entries not selected for the basis are made nonbasic at the value $\mathtt{x}(j)$ (or the closest value inside their bounds). See the description of `hs` below (on exit).

2. If `Start = 'Cold'` or `'Basis file'` and a basis file *is* provided, `hs` and `x` need not be set.

3. For Warm starts, all of $\mathtt{hs}(1 : n + m)$ must be 0, 1, 2 or 3 and all of $\mathtt{x}(1 : n + m)$ must have values (perhaps from some previous call).

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 $\mathtt{pi}(i) = 0.0$, $i = 1 : \mathtt{nnCon}$.

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 work-
           space. 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). Conversely,
           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**. Their lengths **lencw**, **leniw**, **lenrw** must all be at least 500.

           In general, $\mathtt{lencw} = 500$ 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
           factorization. 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 $\mathtt{lencw} = \mathtt{leniw} =
           \mathtt{lenrw} = 500$. See **mincw**, **miniw**, **minrw** below (on exit).

On exit:

hs(n+m) gives the final state of the variables and slacks as follows:

$\mathtt{hs}(j)$	State of variable j	Usual value of $\mathtt{x}(j)$
0	nonbasic	$\mathtt{bl}(j)$
1	nonbasic	$\mathtt{bu}(j)$
2	superbasic	Between $\mathtt{bl}(j)$ and $\mathtt{bu}(j)$
3	basic	Between $\mathtt{bl}(j)$ and $\mathtt{bu}(j)$

           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
           $\mathtt{x}(j)$ lies strictly between its bounds.

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

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

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

rc(n+m) is a vector of reduced costs, $g - \begin{pmatrix} A & -I \end{pmatrix}^T \pi$, where g is the gradient of the
           objective if **x** is feasible (or the gradient of the phase-1 objective otherwise). The
           last m entries are π.

INFO   reports the result of the call to `snOptB`. Here is a summary of possible values. Further details are in Section 8.6.

         Finished successfully
    1   optimality conditions satisfied
    2   feasible point found
    3   requested accuracy could not be achieved
    5   elastic objective minimized
    6   elastic infeasibilities minimized

         The problem appears to be infeasible
    11   infeasible linear constraints
    12   infeasible linear equality constraints
    13   nonlinear infeasibilities minimized
    14   linear infeasibilities minimized
    15   infeasible linear constraints in QP subproblem
    16   infeasible nonelastic constraints

         The problem appears to be unbounded
    21   unbounded objective
    22   constraint violation limit reached

         Resource limit error
    31   iteration limit reached
    32   major iteration limit reached
    33   the superbasics limit is too small
    34   time limit reached

         Terminated after numerical difficulties
    41   current point cannot be improved
    42   singular basis
    43   cannot satisfy the general constraints
    44   ill-conditioned null-space basis
    45   unable to compute acceptable LU factors

         Error in the user-supplied functions
    51   incorrect objective derivatives
    52   incorrect constraint derivatives
    56   irregular or badly scaled problem functions

         Undefined user-supplied functions
    61   undefined function at the first feasible point
    62   undefined function at the initial point
    63   unable to proceed into undefined region

         User requested termination
    72   terminated during constraint evaluation
    73   terminated during objective evaluation
    74   terminated from monitor routine

         Insufficient storage allocated
    81   work arrays must have at least 500 elements
    82   not enough character storage
    83   not enough integer storage
    84   not enough real storage

> *Input arguments out of range*
> 91   invalid input argument
> 92   basis file dimensions do not match this problem
>
> *System error*
> 141   wrong number of basic variables
> 142   error in basis package

mincw, miniw, minrw say how much character, integer, and real storage is needed to solve the problem. If `Print level` > 0, these values are printed. If snOptB terminates because of insufficient storage (`INFO = 82`, 83 or 84), the values may be used to define better values of `lencw`, `leniw` or `lenrw`.

If `INFO = 82`, the work array `cw(lencw)` was too small. snOptB may be called again with `lencw = mincw`.

If `INFO = 83` or 84, 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 because it is not certain how much storage the basis factorization needs.)

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 `Minor feasibility tolerance` *before the solution is unscaled.*

If any *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 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 bounds by more than the `Minor feasibility tolerance`. Again this is *before the solution is unscaled.*

Obj   is the final value of the nonlinear part of the objective function. If $nInf = 0$, Obj is the nonlinear objective, if any. If $nInf > 0$ but the linear constraints are feasible, then Obj is the nonlinear objective. If $nInf > 0$ and the linear constraints are infeasible, Obj is zero.

Note that Obj does not include contributions from the constant term `ObjAdd` or the objective row, if there is one. The final value of the objective being optimized is `ObjAdd + x(n+iObj) + Obj`, where `iObj` is the index of the objective row in A.

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 `funcon` or `funobj` 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` ≤ 2. 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 near bounds.

8. Set `mode = −1` if some of the functions are undefined. The linesearch will shorten the step and try again.

9. Set `mode` $\leq −2$ if you want `snOptB` to stop.

4.6. 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 $\partial f/\partial x_j$.

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(*,*)` (stored column-wise in Fortran). It is then simple to compute the Jacobian by rows or by columns. For sparse Jacobians, it is essential to use a one-dimensional array `gCon(*)` to conserve storage.

```
      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
     &     cu(lencu)*8
!!!   double precision   gCon(nnCon,nnJac)  ! Dense  ! Choose ONE of these
      double precision   gCon(neJac)        ! Sparse
```

On entry:

`mode`   indicates whether `fCon` or `gCon` or both must be assigned during the present call of `funcon` ($0 \leq$ `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 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` $\leq n$). 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` $\times$ `nnJac`.

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

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

>   If `nState` $= 0$, there is nothing special about the current call to `funcon`.
>
>   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 is a nonlinear objective, 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 the last call to `funcon` will occur *before* the last call to `funobj`.
>
>   In general, the last call is made with `nState` $= 2 + $ `INFO`$/10$, where `INFO` is the integer returned by `snOptB` (see p.). 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;
>>   if `nState` $= 5$, an iterations limit was reached.
>
>   If the functions are expensive to evaluate, it may be desirable to do nothing on the last call. The first executable statement could be   `if (nState .ge. 2) return`.

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 `funcon` and `funobj` 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:

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` consistently with the arrays `Acol`, `indA`, `locA` that define the sparsity pattern of $f(x'')$ and A_2 in (4.2), *excluding the elements of A_2*. There is no internal check for consistency (except indirectly via the `Verify` option), so great care is essential.

mode may be used to indicate that you are unable to evaluate f or its gradients at the current x. (For example, the problem functions may not be defined there).

>   During the linesearch, $f(x)$ is evaluated at points $x = x_k + \alpha p_k$ for various steplengths α, where $f(x_k)$ has already been evaluated satisfactorily. For any such x, if you set `mode` $= -1$, `snOptB` will reduce α and evaluate f again (closer to x_k, where it is more likely to be defined).
>
>   If for some reason you wish to terminate the current problem, set `mode` ≤ -2.

4.7. Subroutine funobj

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

```
 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
&      cu(lencu)*8
```

On entry:

mode   may be set as in funcon.

nnObj   is the number of variables involved in $f_0(x)$ ($0 < \text{nnObj} \le n$). 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 is used as in funcon.

cu(lencu), iu(leniu), ru(lenru) are the same as in funcon.

On exit:

mode   may be set as in funcon to indicate that you are unable to evaluate f_0 at x.

       If you wish to terminate the solution of the current problem, set mode ≤ -2.

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 $= 0$).

4.8. Example

Here we give the subroutines funobj and funcon for the example of Section 4.2, repeated here for convenience with generic variables x_j:

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

and $x_3 \ge 0$, $x_4 \ge 0$. This problem has 4 variables, 3 nonlinear objective variables, 2 nonlinear Jacobian variables, 2 nonlinear constraints, 1 linear constraint, and two bounded variables. The objective has some linear terms that we include as an extra "free row" (with infinite bounds). The calling program must assign the following 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 Jacobian variables (x_1, x_2).

For interest, we test `mode` to economize on gradient evaluations (even though they are cheap here). Note that `Nonderivative 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
     &     cu(lencu)*8

*     ===================================================================
*     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). First we treat the Jacobian as a *dense matrix*. In Fortran it is preferable to access two-dimensional arrays column-wise, as shown. Each column of `gCon` contains two elements, even though one of them is zero. The `Acol`, `indA`, `locA` data structure *must include these elements*, as well as other entries belonging to the linear constraints.

Since `funcon` is called before `funobj`, we test `nState` for the first and last entries.

```fortran
      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
&        cu(lencu)*8

*     ===================================================================
*     Toy NLP problem with dense Jacobian.
*     ===================================================================
      integer
&        nout
      double precision
&        x1, x2

      nout   = 9
      x1     = x(1)
      x2     = x(2)

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

      if (mode .eq. 0  .or.  mode .eq. 2) then
         fCon(1)    = x1**2 + x2**2
         fCon(2)    =          x2**4
      end if

      if (mode .ge. 1) then
         gCon(1,1) = 2.0d+0*x1      ! Jacobian elements for column 1
         gCon(2,1) = 0.0d+0         ! Can't be omitted

         gCon(1,2) = 2.0d+0*x2      ! Jacobian elements for column 2
         gCon(2,2) = 4.0d+0*x2**3
      end if

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

      end ! subroutine funcon
```

Now we treat the Jacobian as a sparse matrix, stored column-wise. Note that `gCon` has only 3 entries because we intentionally omit the zero entry in column 1. The first column of the `Acol`, `indA`, `locA` data structure must *also* have that entry deleted.

```
      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(neJac), ru(lenru), x(nnJac)
      character
     &      cu(lencu)*8

*     ==================================================================
*     Toy NLP problem with dense Jacobian.
*     ==================================================================
      integer
     &      nout
      double precision
     &      x1, x2

      nout   = 9
      x1     = x(1)
      x2     = x(2)

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

      if (mode .eq. 0  .or.  mode .eq. 2) then
         fCon(1)   = x1**2 +  x2**2
         fCon(2)   =          x2**4
      end if

      if (mode .ge. 1) then
         gCon(1) = 2.0d+0*x1        ! Jacobian elements for column 1

         gCon(2) = 2.0d+0*x2        ! Jacobian elements for column 2
         gCon(3) = 4.0d+0*x2**3
      end if

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

      end ! subroutine funcon
```

4.9. 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 is 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 Acol(*) 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.

4.10. Subroutine snMemB

This routine estimates the size of the workspace arrays cw, iw, rw required to solve an optimization problem of given dimensions. snMemB is not strictly needed in f77 because all workspace must be defined explicitly in the driver program at compile time. It is available for users wishing to allocate storage dynamically in f90 or C.

The actual storage required also depends on the values of the following options:

>   Hessian full memory
> *or* Hessian limited memory and Hessian updates
>   Reduced Hessian dimension
>   Superbasics limit

If these options have not been set, default values are assumed. Ideally the correct values should be set *before* the call to snMemB.

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

 integer
&      INFO, lencw, leniw, lenrw, m, mincw, miniw, minrw, n, neA,
&      negCon, nnCon, nnJac, nnObj, iw(leniw)
 double precision
&      rw(lenrw)
 character
&      cw(lencw)*8
```

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

On entry:

negCon is the number of nonzeros in the Jacobian gCon (negCon ≤ nnCon ∗ nnJac).

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

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

On exit:

INFO   reports the result of the call to snMemB. Further details are in Section 8.6.

          Insufficient storage allocated
      81   work arrays must have at least 500 elements

          Finished successfully
      104   memory requirements estimated

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

To use snMemB, the first step is to allocate the work arrays. These may be temporary
arrays tmpcw, tmpiw, tmprw (say) or the snOptB arrays cw, iw, rw, which will be reallocated
after the storage limits are known. Here we illustrate the use of snMemB using the same
arrays for 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, rw arrays in
order to retain the options.

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

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

As with all SNOPT routines, snInit (or snInitF) *must* be called to set default options:

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

call snInitF
&   ( printfile, summaryfile, iPrint, iSumm,
&     cw, ltmpcw, iw, ltmpiw, rw, ltmprw )
```

This installs ltmpcw, ltmpiw, ltmprw as the default internal upper limits on the snOptB
workspace (see Total real workspace p. 93). They are used to compute the boundaries
of any user-defined workspace in cw, iw or rw.

The next step is to call snMemB to obtain mincw, miniw, minrw as estimates of the storage
needed by snOptB:

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

The output values of mincw, miniw, 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 or C to allocate the final work arrays for the problem.

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

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

An alternative way is to call `snInit` (or `snInitF`) again with arguments `lencw`, `leniw`, `lenrw`:

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

However, this has the twin effects of resetting all options to their default values, and reprinting the SNOPT banner (unless `iPrint = 0` and `iSumm = 0` are set for the 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 NP 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 routine usrfun.

```
      subroutine snOptC
     &   ( Start, m, n, neA, nName,
     &     nnCon, nnObj, nnJac,
     &     iObj, ObjAdd, Prob,
     &     usrfun,
     &     Acol, indA, locA, bl, bu, Names,
     &     hs, x, pi, rc,
     &     INFO, mincw, miniw, minrw,
     &     nS, nInf, sInf, Obj,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )

      external
     &     usrfun
      integer
     &     INFO, iObj, lencu, leniu, lenru, lencw, leniw, lenrw,
     &     mincw, miniw, minrw, m, n, neA, nName, nS, nInf, nnCon,
     &     nnObj, nnJac, indA(neA), hs(n+m), locA(n+1), iu(leniu),
     &     iw(leniw)
      double precision
     &     Obj, ObjAdd, sInf, Acol(neA), bl(n+m), bu(n+m), pi(m),
     &     rc(n+m), ru(lenru), rw(lenrw), x(n+m)
      character*(*)
     &     Start
      character
     &     Prob*8, Names(nName)*8, cu(lencu)*8, cw(lencw)*8
```

All arguments except usrfun are the same as those for snOptB on p. . A 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 Sections [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
     &     cu(lencu)*8
!!!   double precision    gCon(nnCon,nnJac)  ! Dense  ! Choose ONE of these
      double precision    gCon(neJac)            ! Sparse
```

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 $f_0(x)$ ($0 < \text{nnObj} \le n$). 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 $f(x)$ ($0 < \text{nnJac} \le n$). These must be the first `nnJac` variables in the problem.

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

neJac   is the number of nonzero elements in `gCon`. If `gCon` is stored as a two-dimensional array, then $\text{neJac} = \text{nnCon} \times \text{nnJac}$.

x(nnL)   contains the nonlinear 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.

> If nState ≥ 2, snOptC 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 nState $= 2 + $ INFO, where INFO 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;
>> if nState $= 5$, an iterations limit was reached.

> If the functions are expensive to evaluate, it may be desirable to do nothing on the last call. The first executable statement could be if (nState .ge. 2) return.

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 routine 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. For any such x, if you set mode to -1, snOptC will reduce α and evaluate the functions again (closer to x_k, where they are more likely to be defined).

> If for some reason you wish to terminate the current problem, set mode ≤ -2.

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. [10]). For the case of npOpt it is convenient to restate problem NP with the constraints reordered as follows:

$$
\begin{array}{ll}
\text{DenseNP} & \underset{x}{\text{minimize}} \qquad f_0(x) \\[2ex]
& \text{subject to} \ \ l \le \begin{pmatrix} x \\ A_L x \\ f(x) \end{pmatrix} \le u,
\end{array}
$$

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 is

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

where npSpec reads a set of optional parameter definitions from the file with unit number iSpecs.

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. 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.
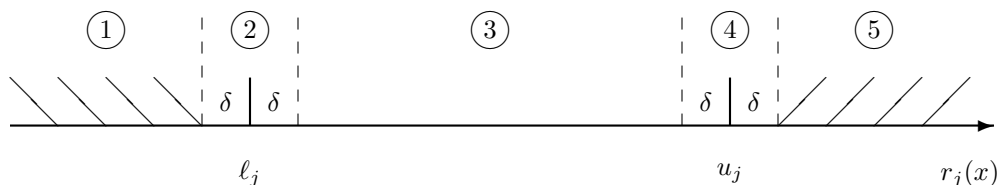

Figure 1:   Illustration of the constraints $\ell_j \le r_j(x) \le u_j$. The bounds ℓ_j and u_j are considered "satisfied" if $r_j(x)$ lies in Regions 2, 3 or 4, where δ is the feasibility tolerance. The constraints $r_j(x) \ge \ell_j$ and $r_j(x) \le u_j$ are both considered "inactive" if $r_j(x)$ lies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

6.1. Subroutines used by `npOpt`

`npOpt` is accessed via the following routines:

`npInit` (Section 6.7) or `npInitF` must be called before any other `npOpt` routines.

`npSpec` (Section 7.8) or `npSpecF` may be called to input a Specs file (a list of run-time options).

`npSet`, `npSeti`, `npSetr` (Section 7.8) may be called to specify a single option.

`npGet`, `npGeti`, `npGetr` (Section 7.8) may be called to obtain an option's current value.

`npOpt` (Section 6) is the main solver.

`funcon`, `funobj` (Section 6.3) are supplied by the user and called by `npOpt`. They define the constraint functions $f(x)$ and objective function $f_0(x)$ and ideally their gradients. (They have a fixed parameter list but may have any convenient name. They are passed to `npOpt` as parameters.)

`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.

6.2. Subroutine `npOpt`

In the following specification of `npOpt`, we define $r(x)$ as the vector of combined constraint functions $r(x) = \begin{pmatrix} x & A_L x & f(x) \end{pmatrix}$, and use `nctotl` to denote a variable that holds its dimension: `nctotl = n + nclin + 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,
     &     INFO, majIts, iState,
     &     fCon, gCon, cMul, fObj, gObj, Hess, x,
     &     iw, leniw, rw, lenrw )

      external
     &     funcon, funobj
      integer
     &     INFO, 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 ($n > 0$).

`nclin`   is m_L, the number of general linear constraints ($nclin > 0$).

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

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

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

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

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

`bl(nctotl), bu(nctotl)` contain the lower and upper bounds for $r(x)$ in problem DenseNP.

> To specify non-existent bounds, set $\texttt{bl}(j) \leq -\texttt{infBnd}$ or $\texttt{bu}(j) \geq \texttt{infBnd}$, where `infBnd` is the `Infinite Bound size` (default value 10^{20}).
> To specify an *equality* constraint (say $r_j(x) = \beta$), set $\texttt{bl}(j) = \texttt{bu}(j) = \beta$, where $|\beta| < \texttt{infBnd}$.

> For the data to be meaningful, it is required that $\texttt{bl}(j) \leq \texttt{bu}(j)$ 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`. See Sections 6.3–6.5.

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

> 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, $\texttt{istate}(j)$ 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 $\texttt{istate}(j)$ 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 $\texttt{bl}(j) = \texttt{bu}(j)$. 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 $k \geq \texttt{n}$. If $\texttt{ncnln} = 0$, `gCon` is not referenced. (In that case, `gCon` may be dimensioned (`ldg,1`) with $\texttt{ldg} = 1$.)

> 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 Section 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 $\text{istate}(j) = 0$, $j = \text{n} + \text{nclin} + i$, then $\text{cMul}(j)$ should be zero. If the constraint $r_j(x) \geq \ell_j$ is active ($\text{istate}(j) = 1$), $\text{cMul}(j)$ should be non-negative, and if $r_j(x) \leq u_j$ is active ($\text{istate}(j) = 2$), $\text{cMul}(j)$ 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., $\text{H}(i,j) \approx \partial^2\mathcal{L}(x,\lambda)/\partial x_i\partial x_j$, 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 `INFO` = 43 or 44.

On exit:

`INFO`   reports the result of the call to `npOpt`. Here is a summary of possible values (further details are in Section 8.6).

	Finished successfully
1	optimality conditions satisfied
2	feasible point found
3	requested accuracy could not be achieved
5	elastic objective minimized
6	elastic infeasibilities minimized
	The problem appears to be infeasible
11	infeasible linear constraints
12	infeasible linear equality constraints
13	nonlinear infeasibilities minimized
14	linear infeasibilities minimized
15	infeasible linear constraints in QP subproblem
16	infeasible nonelastic constraints
	The problem appears to be unbounded
21	unbounded objective
22	constraint violation limit reached

Resource limit error

31   iteration limit reached
32   major iteration limit reached
33   the superbasics limit is too small
34   time limit reached

Terminated after numerical difficulties

41   current point cannot be improved
42   singular basis
43   cannot satisfy the general constraints
44   ill-conditioned null-space basis
45   unable to compute acceptable LU factors

Error in the user-supplied functions

51   incorrect objective derivatives
52   incorrect constraint derivatives
56   irregular or badly scaled problem functions

Undefined user-supplied functions

61   undefined function at the first feasible point
62   undefined function at the initial point
63   unable to proceed into undefined region

User requested termination

71   terminated during function evaluation
74   terminated from monitor routine

Insufficient storage allocated

81   work arrays must have at least 500 elements
82   not enough character storage
83   not enough integer storage
84   not enough real storage

Input arguments out of range

91   invalid input argument
92   basis file dimensions do not match this problem

System error

141   wrong number of basic variables
142   error in basis package

iter   is the number of major iterations performed.

istate   describes the status of the constraints $\ell \leq r(x) \leq u$ in problem DenseNP. For the
         jth lower or upper bound, $j = 1$ to nctotl, the possible values of istate(j) are as
         follows, where δ is the specified 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 as follows:

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

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 Section 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 an estimate of H, 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 con-
straints. The objective function is defined by subroutine funobj, and the nonlinear con-
straints 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 ap-
proximated by finite differences; see Section 7 for a discussion of the optional parameter
Derivative level. Just before either funobj or funcon is called, each element of the cur-
rent 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 [14] for a detailed discussion). If all gradients
cannot be provided, it is similarly advisable to provide as many as possible. During the
development of subroutines funobj and funcon, the Verify parameter (p. 94) 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. For any such x, if you set `mode` to -1, `npOpt` will reduce α and evaluate the functions again (closer to x_k, where they are more likely to be defined).

If for some reason you wish to terminate the current problem, set `mode` ≤ -2.

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 Section 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` ($\mathtt{ldg} \geq 1$ and $\mathtt{ldg} \geq \mathtt{ncnln}$).

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   may be set as in `funobj`.

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.

6.7.   Subroutine npInit

Subroutine `npInit` must be called before any other NPOPT routine. It defines the Print and Summary files, prints a title on both files, and sets all user options to be undefined. (Each NPOPT interface will later check the options and set undefined ones to default values.)

```
      subroutine npInit
   &    ( iPrint, iSumm, iw, leniw, rw, lenrw )

      integer
   &      iPrint, iSumm, leniw, lenrw, iw(leniw)
      double precision
   &      rw(lenrw)
```

On entry:

iPrint defines a unit number for the Print file. Typically `iPrint` = 9. On some systems, the file may need to be opened before `npInit` 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 `npInit` is called.
       If `iSumm` ≤ 0, there will be no Summary file output.

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

On exit:

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

6.8.   Subroutine `npInitF`

Subroutine `npInitF` provides the same capabilities as `npInit`. The difference is that filenames are accepted as input rather than Fortran unit numbers.

```
 subroutine npInitF
&   ( printfile, summaryfile, iPrint, iSumm, iw, leniw, rw, lenrw )

 character*(*)
&     printfile, summaryfile
 integer
&     iPrint, iSumm, leniw, lenrw, iw(leniw)
 double precision
&     rw(lenrw)
```

On entry:

`printfile` is a character string containing the name of the Print file. If `printfile` is empty, then there will be no Print file output (analogous to `iPrint` ≤ 0 in `npInit`).

`summaryfile` is a character string containing the name of the Summary file. If `summaryfile` is empty, then there will be no Summary file output (analogous to `iSumm` ≤ 0). To route the output to standard out, set `summaryfile = 'screen'`.

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

On exit:

`iPrint` is the unit number associated with the Print file.

`iSumm` is the unit number associated with the Summary file.

`iw(leniw)`, `rw(lenrw)` Some elements of `iw` and `rw` are given values to indicate that most optional parameters are undefined.

6.9.   Subroutine `npMem`

This routine estimates the size of the workspace arrays `cw`, `iw`, `rw` required to solve an optimization problem of given dimensions. `npMem` is not strictly needed in f77 because all workspace must be defined explicitly in the driver program at compile time. It is available for users wishing to allocate storage dynamically in f90 or C.

The actual storage required also depends on the value of the option `Superbasics limit`. If this option has not been set, the default value is assumed. Ideally the correct value should be set *before* the call to `npMem`.

```
 subroutine npMem
&   ( iExit, n, nclin, ncnln,
&     miniw, minrw,
&     iw, leniw, rw, lenrw )

 implicit
&     none
 integer
&     iExit, n, nclin, ncnln, miniw, minrw, leniw, lenrw,
&     iw(leniw)
 double precision
&     rw(lenrw)
```

The arguments **n**, `nclin` and `ncnln` define the problem being solved and are identical to the arguments used in the call to `npOpt` (see Section 6.2). For a sequence of problems, `npMem` may be called once with overestimates of these quantities.

On entry:

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

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

On exit:

`INFO`   reports the result of the call to `npMem`. Further details are in Section 8.6.

> > *Insufficient storage allocated*
> 81   work arrays must have at least 500 elements
>
> > *Finished successfully*
> 104   memory requirements estimated

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

To use `npMem`, the first step is to allocate the work arrays. These may be temporary arrays `tmpiw, tmprw` (say) or the `npOpt` arrays `iw, rw`, which will be reallocated after the storage limits are known. Here we illustrate the use of `npMem` using the same arrays for `npMem` and `npOpt`. Note that the `npMem` arrays are used to store the optional parameters, and so any temporary arrays must be copied into the final `iw, rw` arrays in order to retain the options.

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

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

As with all NPOPT routines, `npInit` (or `npInitF`) *must* be called to set default options:

```
      call npInit
     &    ( iPrint, iSumm, iw, ltmpiw, rw, ltmprw )
! or, alternatively,
      subroutine npInitF
     &    ( printfile, summaryfile, iPrint, iSumm, iw, ltmpiw, rw, ltmprw )
```

These calls install `ltmpiw` and `ltmprw` as the default internal upper limits on the `npOpt` workspace (see `Total real workspace` p. 93). They are used to compute the boundaries of any user-defined workspace in `iw` or `rw`.

   The next step is to call `npMem` to obtain `miniw` and `minrw` as estimates of the storage needed by `npOpt`:

```
      subroutine npMem
     &    ( iExit, n, nclin, ncnln,
     &      miniw, minrw,
     &      iw, ltmpiw, rw, ltmprw )
```

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

```
      leniw = miniw
      lenrw = minrw
```

These values may be used in f90 or C to allocate the final work arrays for the problem.

   One last step is needed before `npOpt` is called. The current upper limits `ltmpiw` and `ltmprw` must be replaced by the estimates `miniw` and `minrw`. This can be done using the option setting routine `npSeti` as follows:

```
      Errors = 0                    ! Counts the number of errors
      iPrt   = 0                    ! Suppress print   output
      iSum   = 0                    ! Suppress summary output
      call npSeti
     &    ( 'Total integer workspace',  0, iPrt, iSum, Errors,
     &      iw, ltmpiw, rw, ltmprw )

      call npSeti
     &    ( 'Total real    workspace',  0, iPrt, iSum, Errors,
     &      iw, ltmpiw, rw, ltmprw )
```

An alternative way is to call `npInit` (or `npInitF`) again with arguments `leniw`, `lenrw`:

```
      call npInit
     &    ( iPrint, iSumm, iw, leniw, rw, lenrw )
```

However, this has the twin effects of resetting all options to their default values, and reprinting the NPOPT banner (unless $iPrint = 0$ and $iSumm = 0$ are set for the Print and Summary files).

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. Other values may be specified in two ways:

- By calling subroutine `snSpec` or `snSpecF` to read a Specs file (Section 7.1).

- By calling the option-setting routines `snSet`, `snSeti`, `snSetr` (Section 7.6).

The current value of an optional parameter may be examined by calling one of the routines `snGet`, `snGetc`, `snGeti`, `snGetr` (Section 7.7).

7.1.   The SPECS file

The Specs file contains a list of options and values 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`. The file is in free format. Each line specifies a single option, using one or more items as follows:

1. A *keyword* (required for all options).

2. A *phrase* (one or more words) that qualifies the keyword (only for some options).

3. A *number* that specifies an integer or real value (only for some options). Such numbers may be up to 16 contiguous characters in Fortran 77's `I`, `F`, `E` or `D` formats, terminated by a space or new line.

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 (`*`) anywhere on a line. All subsequent characters on the line are ignored.

The `Begin` line is echoed to the Summary file.

7.2.   Multiple sets of options in the Specs file

The keyword `Skip` allows you to collect several sets of options within a single Specs file. In the following example, only the second set of options will be input.

```
Skip Begin options
   Scale all variables
End options

Begin options 2
   Scale linear variables
End options 2
```

The keyword `Endrun` prevents subroutine `snSpec` from reading past that point in the Specs file while looking for `Begin`.

7.3.   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 ($\epsilon \approx 2.22 \times 10^{-16}$).

```
BEGIN checklist of SPECS file parameters and their default values
* Printing
  Major print level                  1        * 1-line major iteration log
  Minor print level                  1        * 1-line minor iteration log
  Print    file                      ?        * specified by subroutines snInit or snInitF
  Summary file                       ?        * specified by subroutines snInit or snInitF
  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                    * options are normally listed
  System information                No        * Yes prints more system information


* Problem specification
  Minimize                                    * (opposite of Maximize)
* Feasible point                              * (alternative to Max or Min)
  Objective row                      1        * has precedence over ObjRow (snOptA)
  Infinite bound                1.0e+20 *


* 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        * NOT ALLOWED IN snOptA
  Stop  objective  check at col  n'_1     * NOT ALLOWED IN snOptA
  Start constraint check at col  1        * NOT ALLOWED IN snOptA
  Stop  constraint check at col  n''_1    * NOT ALLOWED IN snOptA


* Scaling
  Scale option                       0        * No scaling
  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 * e^{2/3}


* QP subproblems
  QPSolver                    Cholesky  * default
  Crash option                       3        * first basis is essentially triangular
  Elastic weight                1.0e+4  * used only during elastic mode
  Iterations limit               10000        * or 20m if that is more
  Partial price                      1        * 10 for large LPs
```

```
* SQP method
* Cold start                              * has precedence over argument start
* Warm start                              * (alternative to a cold start)
  Time limit                    0         * no time limit
  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         n_1 + 1       * n_1 = number of nonlinear variables
  Derivative level            3           * NOT ALLOWED IN snOptA
  Derivative option           1           * ONLY FOR snOptA
  Derivative linesearch                   *
* Nonderivative 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 x_0
  Reduced Hessian dimension  2000         * or Superbasics limit if that is less
  Violation limit           1.0e+6        * unscaled constraint violation limit
  Unbounded step size       1.0e+18       *
  Unbounded objective       1.0e+15       *

* Hessian approximation
  Hessian              full memory        * default if n_1 ≤ 75
  Hessian           limited memory        * default if n_1 > 75
  Hessian frequency        999999         * for full Hessian (never reset)
  Hessian updates              10         * 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

* LUSOL options
  LU factor tolerance        3.99         * for NP  (100.0 for LP)
  LU update tolerance        3.99         * for NP  ( 10.0 for LP)
  LU singularity tolerance  3.2e-11       *
  LU partial   pivoting                   * default threshold pivoting strategy
* LU rook      pivoting                   * threshold rook pivoting
* LU complete pivoting                    * threshold complete pivoting

* Basis files
  Old basis file               0          * input basis map
  New basis file               0          * output basis map
  Backup basis file            0          * output extra 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
  Sticky parameters              No       * Yes makes parameter values persist
  Timing level                   3        * print cpu times
End of SPECS file checklist
```

7.4. Subroutine snSpec

Subroutine **snSpec** may be called to input a Specs file (to specify options for a subsequent call of SNOPT).

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

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.

INFO   reports the result of calling **snSpec**. Here is a summary of possible values.

> *Finished successfully*

101   Specs file read.

> *Errors while reading Specs file*

131   No Specs file specified (iSpecs ≤ 0 or iSpecs > 99).

132   End-of-file encountered while looking for Specs file. **snSpec** encountered end-of-file or **Endrun** before finding **Begin** (see Section 7.2). The Specs file may not be properly assigned.

133   End-of-file encountered before finding **End**. Lines containing **Skip** or **Endrun** may imply that all options should be ignored.

134   **Endrun** found before any valid sets of options.

> 134   There were $i = \texttt{INFO} - 134$ errors while reading the Specs file.

7.5. Subroutine snSpecF

Subroutine **snSpecF** may be called to input a Specs file (to specify options for a subsequent call of SNOPT). **snSpecF** provides the same abilities as **snSpec**, accepting a filename in lieu of a Fortran unit number.

```
 subroutine snSpecF
&   ( specsfile, INFO, cw, lencw, iw, leniw, rw, lenrw )
 character*(*)
&     specsfile
 integer
&     INFO, lencw, leniw, lenrw, iw(leniw)
 double precision
&     rw(lenrw)
 character
&     cw(lencw)*8
```

On entry:

specsfile is character string containing the name of the Specs file.

On exit:

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

INFO reports the result of calling **snSpecF**. The possible value are identical to those of **snSpec**.

7.6.   Subroutines `snSet, snSeti, snSetr`

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

```
      subroutine snSet
     &   ( buffer,          iPrint, iSumm, Errors,
     &                      cw, lencw, iw, leniw, rw, lenrw )
      subroutine snSeti
     &   ( buffer, ivalue, iPrint, iSumm, Errors,
     &                      cw, lencw, iw, leniw, rw, lenrw )
      subroutine snSetr
     &   ( buffer, rvalue, iPrint, iSumm, Errors,
     &                      cw, lencw, iw, leniw, rw, lenrw )

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

On entry:

`buffer` is a string to be decoded. Restriction: `len(buffer)` ≤ 72 (`snSet`) or ≤ 55 (`snSeti`, `snSetr`). Use `snSet` if the string contains all relevant data. For example,

```
      call snSet ( 'Iterations 1000',    iPrint, iSumm, Errors, ... )
```

`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,

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

`rvalue` is a real value associated with the keyword in `buffer`. For example,

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

`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.

`Errors` is the cumulative number of errors, so it should be 0 before the first call in a group of calls to the option-setting routines.

On exit:

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

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

7.7.   Subroutines `snGet`, `snGetc`, `snGeti`, `snGetr`

These routines obtain the current value of a single option or indicate if an option has been set.

```
 integer function snGet
&   ( buffer,         Errors, cw, lencw, iw, leniw, rw, lenrw )
 subroutine snGetc
&   ( buffer, cvalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
 subroutine snGeti
&   ( buffer, ivalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
 subroutine snGetr
&   ( buffer, rvalue, Errors, cw, lencw, iw, leniw, rw, lenrw )

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

On entry:

`buffer` is a string to be decoded. Restriction: `len(buffer)` ≤ 72.

`Errors` is the cumulative number of errors, so it should be 0 before the first call in a group of calls to option-getting routines.

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

On exit:

`snGet` is 1 if the option contained in buffer has been set, otherwise 0. Use `snGet` to find if a particular optional parameter has been set. For example: if

```
    i = snGet( 'Hessian limited memory', Errors, ... )
```

then i will be 1 if SNOPT is using a limited-memory approximate Hessian.

`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, Errors, ... )
```

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

```
    call snGeti( 'Iterations limit', itnlim, Errors, ... )
```

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

```
    call snGetr( 'LU factor tol', factol, Errors, ... )
```

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

7.8.   Subroutines npSpec, npSpecF, npGet, npGeti, npGetr

These routines associated with the npOpt interface are analogous to snSpec, snSpecF, snGet, snGeti and snGetr.

```
 subroutine npSpec
&   ( iSpecs,    INFO, iw, leniw, rw, lenrw )

 subroutine npSpecF
&   ( specsfile, INFO, iw, leniw, rw, lenrw )

 integer function npGet
&   ( buffer,         Errors, iw, leniw, rw, lenrw )

 subroutine npGeti
&   ( buffer, ivalue, Errors, iw, leniw, rw, lenrw )

 subroutine npGetr
&   ( buffer, rvalue, Errors, iw, leniw, rw, lenrw )

 character*(*)
&     buffer,  specsfile
 integer
&     Errors, INFO, iSpecs, ivalue, leniw, lenrw, iw(leniw)
 double precision
&     rvalue, rw(lenrw)
```

7.9.   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 NP to refer to the objective and constraint functions.

`Backoff factor` τ_b Default = 0.1

This controls the reduction in the steplength when a function cannot be computed or is undefined at the current point during the linesearch. The step is reduced by a factor τ_b and functions are requested at the new point.

`Backup basis file` f 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.

   The following example eliminates this risk:

```
Old basis file      11      (or 0)
Backup basis file   11
New basis file      12
Save frequency      100
```

used with basis backupThe 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 = $\epsilon^{1/3} \approx$ `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` k Default = 60

Every kth 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.

Cold Start   Default = value of input argument `start`

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.

    This parameter has the same effect as the input arguments `start = 0` for `snOptA`, and
`start = 'Cold'` for `snOptB` and `snOptC`. If specified as an optional parameter, this value
has precedence over the value of the input argument `start`. This allows the `start` parameter
to be changed at run-time using the Specs file.

Crash option i Default = 3
Crash tolerance t 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 $A\,m\,I$. 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 A.

   2   CRASH is called twice (if there are nonlinear constraints). The first call looks for a
       triangular basis in linear rows, and the iteration proceeds with simplex iterations until
       the linear constraints are satisfied. The Jacobian is then evaluated for the first major
       iteration and CRASH is called again to find a triangular basis in the nonlinear rows
       (retaining the current basis for linear rows).

   3   CRASH is called up to three times (if there are nonlinear constraints). The first two
       calls treat *linear equalities* and *linear inequalities* separately. As before, the last call
       treats nonlinear rows before the first major iteration.

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

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

    When $t > 0.0$, 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 form the matrix shown under `LU factor`
`tolerance`; i.e., a tridiagonal matrix with entries -1, 2, -1. To help CRASH choose all m

columns for the initial basis, we would specify `Crash tolerance` t for some value of $t > 0.5$.

`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` Default
`Nonderivative linesearch`

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 line search is completed, the problem functions are called again with `mode` = 2. If the potential savings provided by a nonderivative line search 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 runtime 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 (see the discussion of the option `Derivative level`).

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 $\qquad\qquad\qquad\qquad f \qquad\qquad\qquad\qquad$ Default $= 0$

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

Elastic weight $\qquad\qquad\qquad\qquad \omega \qquad\qquad\qquad\qquad$ Default $= 10^4$

This keyword determines the initial weight γ associated with problem NP(γ) on p. 13.

At major iteration k, if elastic mode has not yet started, a scale factor $\sigma_k = 1 + \|g(x_k)\|_\infty$ is defined from the current objective gradient. Elastic mode is then started if the QP subproblem is infeasible, or the QP dual variables are larger in magnitude than $\sigma_k \omega$. The QP is re-solved in elastic mode with $\gamma = \sigma_k \omega$.

Thereafter, major iterations continue in elastic mode until they converge to a point that is optimal for problem NP(γ). If the point is feasible for NP ($v = w = 0$), it is declared locally optimal. Otherwise, γ is increased by a factor of 10 and major iterations continue. If γ has already reached a maximum allowable value, NP is declared locally infeasible.

Expand frequency $\qquad\qquad\qquad\qquad k \qquad\qquad\qquad\qquad$ Default $= 10000$

This option is part of the EXPAND anti-cycling procedure [12] 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 k iterations, the tolerance actually used by SNOPT increases from $\frac{1}{2}\delta$ to δ (in steps of $\frac{1}{2}\delta/k$).

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 k helps reduce the number of slightly infeasible nonbasic 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 $\qquad\qquad\qquad\qquad k \qquad\qquad\qquad\qquad$ Default $= 100$ (LP) or 50 (NP)

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.

```
Feasible point
see Minimize
```

```
Function precision                    $\epsilon_R$                    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 linesearch procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

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

```
Hessian dimension                     $i$              Default $= \min\{2000, n_1 + 1\}$
see Reduced Hessian dimension
```

```
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.)

The default value depends on n_1, the number of nonlinear variables (see p. 10). For `npOpt`, n_1 is the number of variables. For `snOptB` and `snOptC`, n_1 is the larger of `nnObj` and `nnJac`, the number of nonlinear objective and Jacobian variables. For `snOptA`, n_1 is determined by the implicit values of `nnObj` and `nnJac` determined after the constraints and variables are reordered.

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 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 = 10` have a similar effect to `Hessian Limited memory` and `Hessian updates = 10` (except that the latter retains the current diagonal during resets).

Hessian updates i Default = 10

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.

Infinite bound r Default = 1.0e+20

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

Iterations limit i 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. (`Itns` is an alternative keyword.)

Linesearch tolerance t Default = 0.9

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

- t must be a real value in the range $0.0 \leq t \leq 1.0$.

- The default value $t = 0.9$ requests just moderate accuracy in the linesearch.

- If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try $t = 0.1$, 0.01 or 0.001. The number of major iterations might decrease.

- If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. *If all gradients are known*, try $t = 0.99$. (The number of major iterations might increase, but the total number of function evaluations may decrease enough to compensate.)

- If not all gradients are known, a moderately accurate search remains appropriate. Each search will require only 1–5 function values (typically), but many function calls will then be needed to estimate missing gradients for the next iteration.

`Load file`	f	Default $= 0$

If $f > 0$, this references a file containing basis information in the format of Section 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`	t_1	Default $= 100.0$ (LP) or 3.99 (NP)
`LU update tolerance`	t_2	Default $=$ 10.0 (LP) or 3.99 (NP)

These tolerances affect the stability and sparsity of LUSOL's basis factors $B = LU$ [11] during refactorization and updating, respectively. They must satisfy t_1, $t_2 \geq 1.0$. The matrix L is a product of matrices of the form

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

where the multipliers μ satisfy $|\mu| \leq t_i$. Smaller values of t_i favor stability, while larger values favor sparsity.

For certain very regular structures (e.g., band matrices) it may be necessary to reduce t_1 and/or t_2 in order to achieve stability. For example, if the columns of A include a submatrix of the form

$$\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix},$$

one should set both t_1 and t_2 to values in the range $1.0 \leq t_i < 2.0$.

`LU partial  pivoting`	Default
`LU rook     pivoting`	
`LU complete pivoting`	

The LUSOL factorization implements a Markowitz-type search for pivots that locally minimize the fill-in subject to a threshold pivoting stability criterion. The `rook` and `complete pivoting` options are more expensive than `partial pivoting` but are more stable and better at revealing rank, as long as the `LU factor tolerance` is not too large (say $t_1 < 2.0$).

When numerical difficulties are encountered, SNOPT automatically reduces the LU tolerances toward 1.0 and switches (if necessary) to `rook pivoting` before reverting to the default or specified options at the next refactorization. (With `System information Yes`, relevant messages are output to the Print file.)

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

| `LU density     tolerance` | t_1 | Default $= 0.6$ |
| `LU singularity tolerance` | t_2 | Default $= \epsilon^{2/3} \approx$ `3.2e-11` |

The density tolerance t_1 is used during LUSOL's basis factorization $B = LU$. 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 t_1, the Markowitz strategy for choosing pivots is terminated and the remaining matrix is factored by a dense LU procedure. Raising t_1 towards 1.0 may give slightly sparser factors, with a slight increase in factorization time.

The singularity tolerance t_2 helps guard against ill-conditioned basis matrices. After B is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \leq t_2$ or $|U_{jj}| < t_2 \max_i |U_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart.)

| `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

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

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 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. If $k = 0$, both feasibility and optimality are checked.

| `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

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

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 $\begin{pmatrix} A & -I \end{pmatrix}$, and π is the set of QP dual variables):

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

In the major iteration log, `maxComp` appears as the quantity labeled "`Optimal`".

`Major 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

<div align="center">

`Major print level JFDXbs`

</div>

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 objective and constraint functions.

- **J** $J(x_k)$, the Jacobian matrix.

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

   If `J=1`, the Jacobian matrix will be output column-wise at the start of each major iteration. Column j will be preceded by the value of the corresponding variable x_j and a key to indicate whether the variable is basic, superbasic or nonbasic. (Hence if `J=1`, there is no reason to specify `X=1` unless the objective contains more nonlinear variables than the Jacobian.) A typical line of output is

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

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

   `Major print level 0` suppresses most output, except for error messages.

`Major step limit` r Default = 2.0

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

1. A linesearch determines a step 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 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 linesearch.

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 $t = \texttt{1.0e-6}$, 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. A basis map will also be saved at the end of a run, 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.

Objective Row   Default = value of `snOptA` argument `ObjRow`

This option is intended for `snOptA` only and may be used to specify which row of the input argument F is to act as the objective function. If specified as an optional parameter, this value takes precedence over the input argument `ObjRow`. This allows the `ObjRow` to be changed at run-time using the Specs file.

Old basis file f Default $= 0$

If $f > 0$, the starting point will be obtained from this file in the format of Section 9.1. The file will usually have been output previously as a New basis file. The file will not be acceptable if the number of rows or columns in the problem has been altered.

Partial price i Default $= 10$ (LP) or 1 (NP)

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 suit time-stage models with T time periods.

Pivot tolerance t 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" determines 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 t.

- It is common for two or more variables to reach a bound at essentially the same time. In such cases, the **Feasibility tolerance** provides some freedom to maximize the pivot element and thereby improve numerical stability. An excessively small **Feasibility tolerance** should therefore not be specified. effect on numerical stabilityinteraction with **feasibility tolerance**

- To a lesser extent, the **Expand frequency** also provides some freedom to maximize the pivot element. Hence, an excessively *large* **Expand frequency** should not be specified.

Print file f
Print frequency k Default $= 100$

If $f > 0$, the Print file is output to file number f. If **Minor print level** > 0, a line of the QP iteration log is output every kth iteration. The default f is obtained from subroutine **snInit**'s parameter **iPrint** or automatically assigned in **snInitF**. Set $f = 0$ to suppress output to the Print file.

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.

```
QPSolver   Cholesky                                                    Default
QPSolver   CG
QPSolver   QN
```

 This specifies the method used to solve system (2.2) for the search directions in phase 2 of the QP subproblem.

 `QPSolver Cholesky` holds the full Cholesky factor R of the reduced Hessian $Z^T H Z$. As the minor iterations proceed, the dimension of R changes with the number of superbasic variables. If the number of superbasic variables needs to increase beyond the value of `Reduced Hessian dimension`, the reduced Hessian cannot be stored and the solver switches to `QPSolver CG`. The Cholesky solver is reactivated if the number of superbasics stabilizes at a value less than `Reduced Hessian dimension`.

 `QPSolver QN` solves the QP using a quasi-Newton method similar to that of MINOS. In this case, R is the factor of a quasi-Newton approximate Hessian.

 `QPSolver CG` uses an active-set method similar to `QPSolver QN`, but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.

- The Cholesky QP solver is the most robust, but may require a significant amount of computation if the number of superbasics is large.

- The quasi-Newton QP solver does not require the computation of the R at the start of each QP subproblem. It may be appropriate when the number of superbasics is large but relatively few major iterations are needed to reach a solution (e.g., if SNOPT is called with a Warm start).

- The conjugate-gradient QP solver is appropriate for problems with large numbers of degrees of freedom (say, more than 2000 superbasics).

Reduced Hessian dimension i Default $= \min\{2000, n_1 + 1\}$
same as **Hessian dimension**

This specifies that an $i \times i$ triangular matrix R is to be available for use by the `QPSolver Cholesky` option (to define the reduced Hessian according to $R^T R = Z^T H Z$). The value of i affects when `QPSolver CG` is activated.

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 = 0
Scale tolerance                     t                        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 100).

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` t 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 t times its previous value, another scaling pass is performed to adjust the row and column scales. Raising t 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 three options determine whether the final solution obtained is to be output to the Print file. The `file` option operates independently; if $f > 0$, the final solution will be output to file f (whether optimal or not).

- For the `Yes` and `If Optimal` 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 `infBnd` (default value `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., 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 = n'_1
Stop  Constraint Check at Column        l                         Default = n''_1
```

These keywords are not allowed in the `snOptA` interface. The default values depend on n'_1 and n''_1, the numbers of nonlinear objective variables and Jacobian variables (see p.).

If `Verify level` > 0, these options may be used to abbreviate the verification of individual derivative elements computed by subroutines `funobj`, `funcon` and `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.

```
Sticky parameters                 No                                Default
Sticky parameters                 Yes
```

User-defined optional parameters may be modified so that they lie in a sensible range. For example, any tolerance specified as negative or zero will be changed to its positive default value. Specifying `Sticky parameters No` will result in the original user-defined parameters being reloaded into workspace after the run is completed. If a second run is made immediatly following a call with `Sticky parameters Yes` (e.g., with the `Hot start` option) then any modified parameter values will persist in workspace for the second run.

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

If $f > 0$, the Summary file is output to file f. If `Minor print level` > 0, a line of the QP iteration log is output every kth minor iteration. The default f is obtained from subroutine `snInit`'s parameter `iSumm` or assigned in `snInitF`. Set $f = 0$ to suppress the Summary file.

```
Superbasics limit                 i                          Default = n_1 + 1
```

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

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

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

Normally, i need not be greater than n_1+1, where n_1 is the number of nonlinear variables. For many problems, i may be considerably smaller than n_1. This will save storage if n_1 is very large.

```
Suppress parameters
```

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

```
System information    No                                    Default
System information    Yes
```

The `Yes` option provides additional information on the progress of the iterations, including Basis Repair details when ill-conditioned bases are encountered and the LU factorization parameters are strengthened.

```
Time limit                         i                        Default = 0
```

This places a limit of i cpu seconds on the time used for solving the problem. The default value $i = 0$ implies that no cpu limit is imposed.

```
Timing level                       ℓ                        Default = 3
```

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

```
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(maxrw + 1 : lenru)` for their own purpose. Similarly for the other work arrays.

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

```
Unbounded objective value    f_max              Default = 1.0e+15
Unbounded step size          α_max              Default = 1.0e+18
```

These parameters are intended to detect unboundedness in nonlinear problems. (They may not achieve that purpose!) During a linesearch, 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 $= 1.0e + 6$

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

$$v_i(x_{k+1}) \leq \tau \max\{10, v_i(x_0)\}, \tag{7.3}$$

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.

`Warm start`	Default $=$ value of input argument `start`

This parameter indicates that a basis is already specified via the input arrays for SNOPT. This option has the same effect as the input arguments `start = 2` for `snOptA`, and `start = 'Warm'` for `snOptB` and `snOptC`. If specified as an optional parameter, this value has precedence over the value of the input argument `start`. This allows the `start` parameter to be changed at run-time using the Specs file.

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 Section 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 linesearch.
`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 (2.3)). This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 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, $\texttt{LU} = \texttt{lenL} + \texttt{lenU}$, 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 $B_S = \begin{pmatrix} B & S \end{pmatrix}^T$ with stability being favored more than sparsity.

nS   The current number of superbasic variables.

CondHz   An estimate of the condition number of $R^T R$, an estimate of $Z^T H Z$, the reduced Hessian of the Lagrangian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R (which is a lower bound on the condition number of $R^T R$). `Cond Hz` gives a rough indication of whether or not the optimization procedure is having difficulty. If 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 major 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 linesearch gives a small step, or x is close to being optimal. In some cases, it may be necessary to re-solve the QP subproblem with the central-difference gradient and Jacobian.

d   During the linesearch it was necessary to decrease the step in order to obtain a maximum constraint violation conforming to the value of `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 $NP(\gamma)$.

         If SNOPT is already in elastic mode, an "`i`" indicates that the minimizer of the elastic subproblem does not satisfy the linearized constraints. (In this case, a feasible point for the usual QP subproblem may or may not exist.)

M   An extra evaluation of the problem functions was needed to define an acceptable positive-definite quasi-Newton update to the Lagrangian Hessian. This modification is only done when there are nonlinear constraints.

m   This is the same as "`M`" except that it was also necessary to modify the update to include an augmented Lagrangian term.

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 used when the Hessian approximation is diagonal, and hence always follows a Hessian reset.

t   The minor iterations were terminated because of the `Minor iterations 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 `Superbasics 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 the process by which a nonbasic variable (denoted by `jq`) is selected to become superbasic, in addition to those already in the superbasic set. If the problem is purely linear, variable `jq` usually becomes basic immediately (unless it should happen to reach its opposite bound and return to the nonbasic set). If `Partial price` is in effect, variable `jq` is selected from A_{pp} or I_{pp}, the `pp`-th segments of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$.

Label   *Description*

`Itn`   The current iteration number.

`LPmult,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 $\begin{pmatrix} A & -I \end{pmatrix}$.

         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 superbasic variable chosen to become nonbasic.

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

`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 LUSOL [11] factorizes the basis B or the rectangular matrix $B_S = \begin{pmatrix} B & S \end{pmatrix}^T$ before solution of the next QP subproblem. Gaussian elimination is used to compute sparse factors L and U, where PLP^T and PUQ are lower and upper triangular matrices for some permutation matrices P and Q. Stability is ensured as described under the `LU` options (page 84).

   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.

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 $\begin{pmatrix} B & S \end{pmatrix}$. 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, including its diagonals.

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

Umax   The largest 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 5.0 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   The number of triangular columns of B or B_S at the left of L.

dense1   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   The size of the block to be factorized nontrivially after the triangular rows and columns of B or B_S have been removed.

dense2   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 5.0, 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	The number of slacks selected initially.
Free cols	The number of free columns in the basis.
Preferred	The number of "preferred" columns in the basis (i.e., $\mathtt{hs}(j) = 3$ for some $j \leq n$).
Unit	The number of unit columns in the basis.
Double	The number of double columns in the basis.
Triangle	The number of triangular columns in the basis.
Pad	The number of slacks used to pad the basis.

8.6. EXIT conditions

When any solver or auxliliary routine in the SNOPT package terminates, a message of the following form is output to the Print and Summary files:

```
SOLVER EXIT  e -- exit condition
SOLVER INFO  i -- informational message
```

where e is an integer that labels a generic *exit condition*, and i labels one of several alternative *informational messages*. For example, SNOPT may output

```
SNOPTA EXIT 20 -- the problem appears to be unbounded
SNOPTA INFO 21 -- unbounded objective
```

where the exit condition gives a broad definition of what happened, while the informational message is more specific about the cause of the termination. The integer i is the value of the output argument `INFO`. The integer e may be recovered from `INFO` by changing the least significant digit to zero. Possible exit conditions for SNOPT follow:

0	Finished successfully
10	The problem appears to be infeasible
20	The problem appears to be unbounded
30	Resource limit error
40	Terminated after numerical difficulties
50	Error in the user-supplied functions
60	Undefined user-supplied functions
70	User requested termination
80	Insufficient storage allocated
90	Input arguments out of range
100	Finished successfully (associated with SNOPT auxiliary routines)
110	Errors while processing MPS data
120	Errors while estimating Jacobian structure
130	Errors while reading the Specs file
140	System error

Exit conditions 0–20 arise when a solution exists (though it may not be optimal). A basis file may be saved, and the solution is output to the Print or Solution files if requested.

If exit conditions 80–100 occur during the *first* basis factorization, the primal and dual variables `x` and `pi` will have their original input values. Basis files are saved if requested, but certain values in the printed solution will not be meaningful.

We describe each exit message from SNOPT and suggest possible courses of action.

```
EXIT --   0  finished successfully
INFO --   1  optimality conditions satisfied
INFO --   2  feasible point found     (from option Feasible point only)
INFO --   3  requested accuracy could not be achieved
```

These messages usually indicate a successful run. Basis files are saved, and the solution is printed and/or saved on the Solution file.

For `INFO 1` the final point seems to be a solution of NP. This means that `x` is *feasible* (it satisfies the constraints to the accuracy requested by the `Feasibility tolerance`), the reduced gradient is negligible, the reduced costs are optimal, and R is nonsingular.

In all cases, some caution should be exercised. For example, if the objective value is much better than expected, SNOPT 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 "`Optimality conditions satisfied`". Some of the variables or slacks may lie outside their bounds more than desired, especially if scaling was requested. Some information concerning the run can be obtained from the short summary given at the end of the print and summary files. Here is an example from the problem `Toy` discussed in Section 3.2.

```
SNOPTA EXIT    0 -- finished successfully
SNOPTA INFO    1 -- optimality conditions satisfied

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
Max x                     2 1.0E+00   Max pi                       1 1.2E-01
Max Primal infeas         0 0.0E+00   Max Dual infeas              2 8.0E-10
Nonlinear constraint violn    6.4E-09
```

`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

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

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 a smaller `Major optimality tolerance`.

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

If the 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.

```
EXIT -- 10   The problem appears to be infeasible
INFO -- 11   infeasible linear constraints
INFO -- 12   infeasible linear equality constraints
INFO -- 13   nonlinear infeasibilities minimized
INFO -- 14   linear infeasibilities minimized
INFO -- 15   infeasible linear constraints in QP subproblem
INFO -- 16   infeasible nonelastic constraints
```

This exit occurs if SNOPT is unable to find a point satisfying the constraints.

When the constraints are *linear*, the output messages are based on a relatively reliable indicator of infeasibility. 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.

```
EXIT -- 20   The problem appears to be unbounded
INFO -- 21   unbounded objective
INFO -- 22   constraint violation limit reached
```

For linear problems, unboundedness is detected by the simplex method when a nonbasic variable can 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 Section 7.9), 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 informational 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`.

```
EXIT -- 30  Resource limit error
INFO -- 31  iteration limit reached
INFO -- 32  major iteration limit reached
INFO -- 33  the superbasics limit is too small
INFO -- 34  time limit reached
```

Some 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.

If the superbasics limit is too small, then 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 `Superbasics limit` superbasics (and no room for any more).

In general, raise the `Superbasics limit` s by a reasonable amount, bearing in mind the storage needed for the reduced Hessian. (The `Reducd Hessian dimension` h will also increase to s unless specified otherwise, and the associated storage will be about $\frac{1}{2}s^2$ words.) In some cases you may have to set $h < s$ to conserve storage. The `QPSolver CG` option will be invoked and the rate of convergence will probably fall off severely.

```
EXIT -- 40  Terminated after numerical difficulties
INFO -- 41  current point cannot be improved
INFO -- 42  singular basis
INFO -- 43  cannot satisfy the general constraints
INFO -- 44  ill-conditioned null-space basis
```

These conditions arise only after the LU factorization options have been strengthened (automatically) as much as possible.

For `INFO 41`, SNOPT was unable to improve on a non-optimal point.

1. Subroutines `usrfun`, `funobj` or `funcon` may be returning accurate function values but inaccurate gradients (or vice versa). This is the most likely cause. Study the comments given for `INFO 51` and `52`, and check that the coding of the problem functions 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 `Major optimality tolerance` of 10^{-6} would need to be raised to about 10^{-3} for optimality to be declared (at a rather suboptimal point). Of course, it is better to revise the function coding to obtain as much precision as economically possible.

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

One should specify

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

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

For `INFO 42`, the first factorization attempt found the basis to be structurally or numerically singular. (Some diagonals of the triangular matrix U were deemed too small.) The associated variables were replaced by slacks and the modified basis refactorized, but singularity persisted.

For `INFO 43`, the basic variables x_B have been recomputed, 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, but a row check has revealed that the resulting solution does not satisfy the QP constraints $Ax - s = b$ sufficiently well.

For `INFO 44`, during computation of the reduced Hessian $Z^T H Z$, some column(s) of Z continued to contain very large values.

In all cases, the problem must be badly scaled (or the basis must be pathologically ill-conditioned without containing any large entries). Try `Scale option 2` if it has not yet been used.

```
EXIT -- 50   Error in the user-supplied functions
INFO -- 51   incorrect objective derivatives
INFO -- 52   incorrect constraint derivatives
INFO -- 56   irregular or badly scaled problem functions
```

For `INFO 51` and `INFO 52`, there may be errors in the subroutines that define the problem objective and constraints. If the objective derivatives appear to 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 because 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 `INFO 41` above.

For `INFO 51`, 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 the discrepancy. 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.

For `INFO 52`, at least one of the computed constraint derivatives is significantly different from an estimate obtained by forward-differencing the constraint vector $f(x)$ of problem NP. Follow the advice for `INFO 51`, trying to ensure that the arrays `F` and `G` are being set correctly in `usrfun` or `funcon`.

`INFO 56` is set when the search direction is essentially zero, but the optimality conditions are not satisfied numerically. The standard theory for SQP methods implies that if the first-order conditions for an solution hold, then the search direction computed by the QP subproblem is zero and no new solution estimate can be computed. Invariably, `INFO 56`

is an indication that the problem is sufficiently badly scaled that an optimal point is not recognized, i.e., the final point is usually optimal.

```
EXIT -- 60  Undefined user-supplied functions
INFO -- 61  undefined function at the first feasible point
INFO -- 62  undefined function at the initial point
INFO -- 63  unable to proceed into undefined region
```

The parameter `Status` (`snOptA`) or `mode` (`snOptB`, `snOptC`, `npOpt`) was assigned the value -1 in one of the user-defined routines `usrfun`, `funobj` or `funcon`. This value is used to indicate that the functions are undefined at the current point. SNOPT attempts to evaluate the problem functions closer to a point at which the functions have already been computed.

For `INFO` 61 and 62, SNOPT was unable to proceed because the functions are undefined at the initial point or the first feasible point.

For `INFO` 63, repeated attempts to move into a region where the functions are not defined resulted in the change in variables being unacceptably small. At the final point, it appears that the only way to decrease the merit function is to move into a region where the problem functions are not defined.

```
EXIT -- 70  User requested termination
INFO -- 71  terminated during function evaluation
INFO -- 72  terminated during constraint evaluation
INFO -- 73  terminated during objective evaluation
INFO -- 74  terminated from monitor routine
```

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

```
EXIT -- 80  Insufficient storage allocated
INFO -- 81  work arrays must have at least 500 elements
INFO -- 82  not enough character storage
INFO -- 83  not enough integer storage
INFO -- 84  not enough real storage
```

SNOPT cannot start to solve a problem unless the character, integer, and real work arrays are at least 500 elements.

If the storage arrays `cw(*)`, `iw(*)`, `rw(*)` are not large enough for the current problem, an estimate of the additional storage required is given in messages preceding the exit. The routine declaring `cw`, `iw`, `rw` should be recompiled with larger dimensions `lencw`, `leniw`, `lenrw`.

If `rw(*)` is not large enough, be sure that the `Reduced Hessian dimension` is not unreasonably large.

```
EXIT -- 90  Input arguments out of range
INFO -- 91  invalid input argument
INFO -- 92  basis file dimensions do not match this problem
```

These conditions occur if some data associated with the problem is out of range.

For `INFO` 91, at least one input argument for the interface is invalid. The Print and Summary files provide more detail about which arguments must be modified.

For `INFO` 92, an Old basis file could not be loaded properly. (In this situation, new basis files cannot be saved, and there is no solution to print.) On the first line of the Old basis

file, the dimensions 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.

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.

The basis file state vector will not match the current problem if, 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 x_j" entry following the map.

```
EXIT -- 140   System error
INFO -- 141   wrong number of basic variables
INFO -- 142   error in basis package
```

These conditions may arise while the basis is being factorized.

INFO 141 should not happen. The wrong SNOPT source files may have been compiled, or arguments of incorrect type may be used in the call to the SNOPT interface. 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.

For INFO 142, a preceding message describes 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 an error in the input parameters, i.e., the arrays A(*), iAfun(*), jAvar(*), iGfun(*), and jGvar(*) for snOptA, or indA, Acol and locA for snOptB and snOptC.

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 CONSTRAINTS section and a VARIABLES 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 Section 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 CONSTRAINTS section

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

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

and the value of $a^T x$ is called the *constraint value*. Internally, the linear constraints take the form $Ax - s = 0$, where the slack variables s should satisfy the bounds $l \leq s \leq u$. For the ith "constraint", it is the slack variable s_i that is directly available, and it is sometimes convenient to refer to its state.

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

`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 `Major optimality tolerance`.

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

`Value` The constraint value $a^T x$ (or $f_i(x) + a^T x$ for nonlinear constraints).

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

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

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

`Dual variable` 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 VARIABLES section

Here we talk about the "variables" x_j, $j = 1: n$. 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 `Major optimality tolerance`.

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

`Value` 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.

`Dual variable` The dual variable is 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 $m + j$.

   Note: If two problems are the same except that one minimizes $f_0(x)$ and the other maximizes $-f_0(x)$, their solutions will be the same but the signs of the constraint dual variables π_i and the reduced gradients d_j will be reversed.

8.8.   The SOLUTION file

The information in a printed solution (Section 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 CONSTRAINTS 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 VARIABLES 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 Section 3.2, using `Major print level 1` and `Minor print level 0`.

```
===============================
S N O P T  7.7.1    (Nov 2018)
===============================


Nonzero derivs  Jij        5
Non-constant    Jij's      4     Constant Jij's          1

SNJAC  EXIT 100 -- finished successfully
SNJAC  INFO 102 -- Jacobian structure estimated


Nonlinear constraints     2     Linear constraints      1
Nonlinear variables       2     Linear variables        0
Jacobian  variables       2     Objective variables     0
Total constraints         3     Total variables         2

The user has defined      0   out of      4   first  derivatives

Major Minors     Step  nCon Feasible  Optimal  MeritFunction     nS Penalty
    0      2            1   1.0E+00  1.0E+00  1.0000000E+00      2          r
    1      1  1.0E+00   2 (0.0E+00) 1.0E+00 -9.0016883E-13      2         rl
    2      2  1.0E+00   4  1.0E+00  2.5E-01 -1.0000000E+00              sm l
    3      0  1.0E+00   5  9.4E-01  1.8E-02 -9.7904716E-01         2.6E-02
    4      0  1.0E+00   6  5.5E-02  6.3E-03 -1.0005949E+00         2.6E-02
    5      0  1.0E+00   7  1.9E-04  2.4E-05 -1.0000000E+00         4.1E-01
    6      0  1.0E+00   8 (2.2E-09)(2.7E-10)-1.0000000E+00         8.7E-01

SNOPTA EXIT    0 -- finished successfully
SNOPTA INFO    1 -- optimality conditions satisfied
```

```
Problem name                   sntoya
No. of iterations                   5   Objective            -1.0000000003E+00
No. of major iterations             6   Linear    obj. term -1.0000000003E+00
Penalty parameter           8.708E-01   Nonlinear obj. term  0.0000000000E+00
User function calls (total)        36   Calls with modes 1,2 (known g)      8
Calls for forward differencing     16   Calls for central differencing      0
No. of degenerate steps             0   Percentage                       0.00
Max x                        2 1.0E+00   Max pi                       3 1.0E+00
Max Primal infeas            4 2.2E-09   Max Dual infeas              2 2.7E-10
Nonlinear constraint violn    2.2E-09
```

9. Basis Files

A basis file may be saved 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. They are invoked by options of the following form:

```
New basis file    10
Backup    file    11
Punch     file    20
Dump      file    30
```

The file numbers should be in the range 1–99, 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 using options of the following form:

```
Old basis file    10
Insert    file    20
Load      file    30
```

Only one such file will actually be loaded, with the order of precedence as shown. If no basis files are specified, one of the `Crash option`s takes effect.

Figures 2–4 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, which is problem `t4manne` in the SNOPT distribution. (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

These files may be called *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, or the bounds are different.)

As illustrated in Figure 2, 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. Any undefined names will be printed with a blank entry.

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)`:

```
Manne 10  ITN     30    Optimal Soln  NINF      0    OBJ  -2.670098627178E+00
OBJ=        RHS=         RNG=         BND=        M=    20 N=    30 SB=     7
0322222223033333333333333333310000000001111111111
       3    3.21442990995402E+00
       6    3.48787725533622E+00
       8    3.67642838933707E+00
       4    3.30400385655585E+00
       9    3.77158182629954E+00
       5    3.39521914186007E+00
       7    3.58172235348761E+00
       1    3.05000000000000E+00
       2    3.12665036215043E+00
      10    3.86666666666656E+00
       0
```

Figure 2:   Format of New and Old basis files for example `t4manne`

hs(j)	*State of the jth 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

$$j \qquad\qquad x_j$$

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). The list 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, the value x_j is recorded but the state is unaltered.

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 3. 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 *Value* is recorded in x. The key SB is an addition to the standard MPS format to allow for nonbasic solutions.

Notes on Punch Data

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

2. LL lines are *not* output for nonbasic variables whose lower bound is zero.

3. Superbasic slacks are output last.

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 4. The keys LL, UL, BS and SB mean Lower Limit, Upper Limit, Basic, and Superbasic respectively.

Notes on Dump data

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

2. Nonbasic free variables (strictly between their bounds) are output with key LL.

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 is retained.

7. (*Partial basis*) Let m be the number of rows in the problem. If fewer than m variables are specified to be basic, the first basis factorization will detect singularity and insert appropriate slacks.

8. (*Too many basics or superbasics*) If more than m variables are specified basic, or more than `Superbasics limit` are specified superbasic, the excess will be made nonbasic before iterations begin.

9.4. Restarting modified problems

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

Protection

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

Default Status

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

Restarting with different bounds

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

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

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

  In contrast, if X is nonbasic at its *upper* bound and if that bound is altered, the starting values for an arbitrary number of basic variables could be changed (since they will be recomputed from the nonbasic and superbasic variables). This may not be of great consequence, but sometimes it may be worthwhile to retain the old solution precisely. To do this, one can make X superbasic at the original bound value.

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

```
       j               5.0
```

near the end of an Old basis file, or the line

```
       SB X                5.0
```

near the end of an Insert or Load file. The `Superbasics limit` must be at least as large as the number of variables involved, even for purely linear problems.

  The same effect can be obtained when calling `sqOpt` with Warm or Hot Starts. Simply set $hs(j) = 2$ for the appropriate j.

Sequences of problems

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

Acknowledgments

```
NAME           Manne 10  PUNCH/INSERT
 LL x    1                3.05000E+00
 XL x    2  r     1       3.12665E+00
 SB x    3                3.21443E+00
 SB x    4                3.30400E+00
 SB x    5                3.39521E+00
 SB x    6                3.48787E+00
 SB x    7                3.58172E+00
 SB x    8                3.67642E+00
 SB x    9                3.77156E+00
 XL x   10  r     2       3.86667E+00
 LL x   11                9.50000E-01
 XL x   12  r     3       9.68423E-01
 XL x   13  r     4       9.97801E-01
 XL x   14  r     5       1.02820E+00
 XL x   15  r     6       1.05967E+00
 XL x   16  r     7       1.09227E+00
 XL x   17  r     8       1.12608E+00
 XL x   18  r     9       1.16117E+00
 XL x   19  r    10       1.19761E+00
 XU x   20  r    11       1.21394E+00
 XU x   21  r    12       7.66504E-02
 XU x   22  r    13       8.77750E-02
 XU x   23  r    14       8.95733E-02
 XU x   24  r    15       9.12127E-02
 XU x   25  r    16       9.26610E-02
 XU x   26  r    17       9.38453E-02
 XU x   27  r    18       9.47013E-02
 XU x   28  r    19       9.51453E-02
 XU x   29  r    20       9.51023E-02
 UL x   30                1.16000E-01
ENDATA
```

Figure 3:   Format of PUNCH/INSERT files

```
NAME           Manne 10    DUMP/LOAD
 LL x    1                3.05000E+00
 BS x    2                3.12665E+00
 SB x    3                3.21443E+00
 SB x    4                3.30400E+00
 SB x    5                3.39521E+00
 SB x    6                3.48787E+00
 SB x    7                3.58172E+00
 SB x    8                3.67642E+00
 SB x    9                3.77156E+00
 BS x   10                3.86667E+00
 LL x   11                9.50000E-01
 BS x   12                9.68423E-01
 BS x   13                9.97801E-01
 BS x   14                1.02820E+00
 BS x   15                1.05967E+00
 BS x   16                1.09227E+00
 BS x   17                1.12608E+00
 BS x   18                1.16117E+00
 BS x   19                1.19761E+00
 BS x   20                1.21394E+00
 BS x   21                7.66504E-02
 BS x   22                8.77750E-02
 BS x   23                8.95733E-02
 BS x   24                9.12127E-02
 BS x   25                9.26610E-02
 BS x   26                9.38453E-02
 BS x   27                9.47013E-02
 BS x   28                9.51453E-02
 BS x   29                9.51023E-02
 UL x   30                1.16000E-01
 LL r    1                0.00000E+00
 LL r    2                0.00000E+00
 LL r    3                0.00000E+00
 LL r    4                0.00000E+00
 LL r    5                0.00000E+00
 LL r    6                0.00000E+00
 LL r    7                0.00000E+00
 LL r    8                0.00000E+00
 LL r    9                0.00000E+00
 LL r   10                0.00000E+00
 UL r   11                0.00000E+00
 UL r   12                0.00000E+00
 UL r   13                0.00000E+00
 UL r   14                0.00000E+00
 UL r   15                0.00000E+00
 UL r   16                0.00000E+00
 UL r   17                0.00000E+00
 UL r   18                0.00000E+00
 UL r   19                0.00000E+00
 UL r   20                0.00000E+00
ENDATA
```

Figure 4: 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. 13

[2] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963. 4

[3] S. K. Eldersveld, *Large-Scale Sequential Quadratic Programming Algorithms*, PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1991. 13

[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, SIAM, pp. 26–40. 13

[5] R. Fourer, *Solving staircase linear programs by the simplex method. 1: Inversion*, Math. Program., 23 (1982), pp. 274–313. 91

[6] E. M. Gertz, P. E. Gill, and J. Muetherig, *User's guide for SnadiOpt: A package adding automatic differentiation to Snopt*, Report NA 01–1, Department of Mathematics, University of California, San Diego, 2001. 5

[7] 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. 4, 10

[8] ———, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM Rev., 47 (2005), pp. 99–131. 10

[9] P. E. Gill, W. Murray, M. A. Saunders, and E. Wong, *User's guide for SQOPT Version 7.7: Software for large-scale linear and quadratic programming*, Center for Computational Mathematics Report CCoM 19-01, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2019. 11

[10] 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. 4, 13, 57

[11] ———, *Maintaining LU factors of a general sparse matrix*, Linear Algebra Appl., 88/89 (1987), pp. 239–270. 4, 12, 84, 99

[12] ———, *A practical anti-cycling procedure for linearly constrained optimization*, Math. Program., 45 (1989), pp. 437–474. 81

[13] ———, *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. 4, 13

[14] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1981. 62

[15] B. A. Murtagh and M. A. Saunders, *Large-scale linearly constrained optimization*, Math. Program., 14 (1978), pp. 41–72. 4, 11

[16] ———, *A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints*, Math. Programming Stud., 16 (1982), pp. 84–117. 4

[17] ———, *MINOS 5.5 User's Guide*, Report SOL 83-20R, Department of Operations Research, Stanford University, Stanford, CA, Revised 1998. 4