User's Guide for DNOPT Version 4: Software for Nonlinear Programming*

Philip E. GILL and Elizabeth WONG
Department of Mathematics
University of California, San Diego, La Jolla, CA 92093-0112, USA

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

Aug 2020

Abstract

DNOPT is a general-purpose system for constrained optimization. It minimizes a linear or nonlinear function subject to bounds on the variables and general linear or nonlinear constraints. It is suitable for linear and quadratic programming and for linearly constrained optimization, as well as for general nonlinear programs.

DNOPT 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.

DNOPT 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, DNOPT 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). DNOPT 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 (or the f2c translator and a C compiler). DNOPT may be called from a driver program in Fortran, C, or MATLAB.

Keywords: optimization, nonlinear programming, nonlinear constraints, SQP methods, quasi-Newton updates, Fortran software, C software.

pgill@ucsd.edu	https://www.CCoM.ucsd.edu/~peg
saunders@stanford.edu	https://web.stanford.edu/~saunders
elwong@ucsd.edu	https://www.CCoM.ucsd.edu/~elwong

*Research supported in part by funding from Northrop Grumman Aerospace Systems.

Contents

1. Introduction **4**
- 1.1 Problem types . . . 4
- 1.2 Implementation . . . 4
- 1.3 Files . . . 5
- 1.4 Overview of the package . . . 5
- 1.5 Subroutines `dnBEGIN`, `dnEND` . . . 5

2. Description of the SQP method **6**
- 2.1 Major iterations . . . 7
- 2.2 Minor iterations . . . 7
- 2.3 The merit function . . . 9
- 2.4 Treatment of constraint infeasibilities . . . 10

3. The `dnOpt` interface **10**
- 3.1 Subroutines used by `dnOpt` . . . 11
- 3.2 Identifying structure in the objective and constraints . . . 11
- 3.3 Problem dimensions . . . 13
- 3.4 Subroutine `dnOpt` . . . 14
- 3.5 User-supplied subroutines for `dnOpt` . . . 19
- 3.6 Subroutine `funcon` . . . 21
- 3.7 Subroutine `funobj` . . . 22
- 3.8 Constant Jacobian elements . . . 23
- 3.9 Example . . . 24

4. The `dnOptH` interface **25**
- 4.1 Subroutines used by `dnOptH` . . . 26
- 4.2 Subroutine `dnOptH` . . . 27
- 4.3 User-supplied subroutines for `dnOptH` . . . 32
- 4.4 Subroutine `funhes` . . . 33
- 4.5 Example . . . 34

5. The `dnNPSOL` interface **36**
- 5.1 Subroutines used by `dnNPSOL` . . . 36
- 5.2 Subroutine `dnNPSOL` . . . 37
- 5.3 User-supplied subroutines for `dnNPSOL` . . . 41
- 5.4 Subroutine `funobj` . . . 42
- 5.5 Subroutine `funcon` . . . 43
- 5.6 Constant Jacobian elements . . . 44

6. Optional parameters **44**
- 6.1 The SPECS file . . . 44
- 6.2 Multiple sets of options in the Specs file . . . 45
- 6.3 SPECS file checklist and defaults . . . 45
- 6.4 Subroutine `dnSpec` . . . 47
- 6.5 Subroutines `dnSet`, `dnSetInt`, `dnSetReal` . . . 48
- 6.6 Subroutines `dnGet`, `dnGetChar`, `dnGetInt`, `dnGetReal` . . . 49
- 6.7 Description of the optional parameters . . . 50

7. Output **61**

7.1 The PRINT file 61
7.2 The major iteration log 62
7.3 The minor iteration log 64
7.4 EXIT conditions 65
7.5 Solution output 70
7.6 The SOLUTION file 78
7.7 The SUMMARY file 78

References **80**

Index **80**

1. Introduction

DNOPT 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 linear and quadratic programming and for linearly constrained optimization, as well as for general nonlinear programs of the form

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

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 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, DNOPT 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 $\ell_j = u_j$. 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 DNOPT applies the primal simplex method [2]. In this case, the dense orthogonal factors of a nonsingular working-set matrix are maintained.

If only the objective is nonlinear, the problem is *linearly constrained* (LC) and tends to solve more easily than the general case with nonlinear constraints (NC). For both nonlinear cases, DNOPT applies a sequential quadratic programming (SQP) method [6], using quasi-Newton approximations to the Hessian of the Lagrangian. The merit function for steplength control is an augmented Lagrangian, as in the sparse SQP solver SNOPT [7, 8].

The DNOPT package is suitable for nonlinear problems with up to several hundred 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).

1.2. Implementation

DNOPT is implemented as a library of Fortran 77 subroutines. The source code is compatible with all known Fortran 77, 90, and 95 compilers, and can be converted to C code by the `f2c` translator [3] included with the distribution.

All routines in DNOPT 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. Files

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

Print file (Section 7) is a detailed iteration log with error messages and perhaps listings of the options and the final solution.

Summary file (Section 7.7) is a brief iteration log with error messages and the final solution status. Intended for screen output in an interactive environment.

Specs file (Section 6) is a set of run-time options, input by `dnSpec`.

Solution file (Sections 7.5–7.6) keeps a separate copy of the final solution listing.

Unit numbers for the Specs, Print, and Summary files are defined by inputs to subroutines `dnBEGIN` and `dnSpec`. The other DNOPT files are described in Sections 7.

1.4. Overview of the package

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

```
call dnBEGIN( iPrint, iSumm, ...   )
call dnSpec ( iSpecs, ...          )
call dnOpt  ( Start, n, mLCon, ...)
call dnEND  ( iPrint, iSumm, ...   )
```

where `dnSpec` reads a file of run-time options (if any). Also, individual run-time options may be "hard-wired" by calls to `dnSet`, `dnSetInt` and `dnSetReal`.

1.5. Subroutines dnBEGIN, dnEND

Calls to subroutines `dnBEGIN` and `dnEND` start and finish a run (or set of runs) for a given problem. Subroutine `dnBEGIN` must be called before any other DNOPT routine. It defines the Print and Summary files, prints a title on both files, and sets all user options to be undefined. (Each DNOPT interface will later check the options and set undefined ones to default values.)

```
      subroutine dnBEGIN
     &   ( 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 `iPrint` $= 9$.

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

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

The calling sequence for `dnEND` is identical to that of `dnBEGIN`.

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

---

2. Description of the SQP method

Here we summarize the main features of the SQP algorithm used in DNOPT and introduce some terminology used in the description of the library routines and their arguments. The SQP algorithm is fully described by Gill, Saunders and Wong [14].

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.

The method of DNOPT starts by attempting to find a feasible point for the linear constraints and bounds. This is done by solving the linear program

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

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 $v \neq 0$ or $w \neq 0$. DNOPT then terminates without computing the nonlinear functions. Otherwise, all subsequent iterates satisfy the linear constraints. This strategy allows linear constraints to be used to define a region in which the functions can be safely evaluated.

2.1. 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 $\ell_L \le A_L x \le u_L$ and the linearized constraints

$$\ell_N \le f(x_k) + f'(x_k)(x - x_k) \le u_N,$$

where $f'(x_k)$ denotes the *Jacobian matrix*, whose elements are the first derivatives of $f(x)$ evaluated at x_k. The QP constraints then the $n+m$ linear constraints $\ell \le r(x) \le u$, where

$$r(x) = \begin{pmatrix} x \\ A(x - x_k) + b \end{pmatrix},$$

with A an $m \times n$ matrix and b an m-vector defined as

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

With these definitions, the QP subproblem can be written as

$$\text{QP}_k \qquad \underset{x,s}{\text{minimize}} \quad q_k(x) = g_k^T(x - x_k) + \tfrac{1}{2}(x - x_k)^T H_k (x - x_k)$$
$$\text{subject to} \quad \ell \le \begin{pmatrix} x \\ A(x - x_k) + b \end{pmatrix} \le u,$$

where $q_k(x)$ is a quadratic approximation to the Lagrangian function (see, e.g., [15]). 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.

2.2. Minor iterations

Solving the QP subproblem is itself an iterative procedure. Here, the iterations of the QP solver DQOPT form the *minor* iterations of the SQP method. DNOPT has two phases, a *feasibility phase* or *phase 1*, and an *optimality phase* or *phase 2*. In the feasibility phase, DNOPT finds a feasible point by minimizing the sum of the infeasibilities of the violated constraints. Once a feasible point has been found, the optimality phase involves minimizing the quadratic objective function within the feasible region. The same method is applied in both phases, but the objective function changes from the sum of infeasibilities to the objective function of QP_k.

DQOPT uses a reduced-Hessian active-set method implemented as a *reduced-gradient method*. Each iterate is associated with a subset of the constraints known as the *working set*. The term *active-set method* arises because the constraints with indices in the working set are *active* at each iterate, i.e., the constraints in the working set are satisfied with equality. An important property of the working set is that it consists of the indices of a *linearly independent* subset of the active-constraint gradients. 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" (or "infeasible") 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. It must be emphasized that even though a constraint can be active in the sense just described, it may not be included in the working set

At any given iterate x, the rows of the working-set matrix A_W consists of n_{FX} rows of the identity matrix and m_A rows of A. The components of x are partitioned into subvectors x_{FX} and x_{FR}, where the x_{FX} correspond to the components of x that are in the working set (i.e., temporarily fixed at their upper or lower bounds). If P is a permutation matrix that arranges the columns of A_W into the order $\begin{pmatrix} A_{FR} & A_{FX} \end{pmatrix}$ corresponding the elements of x_{FX} and x_{FR}, then the working-set matrix may be partitioned such that

$$A_W P = \begin{pmatrix} & I_{FX} \\ A_{FR} & A_{FX} \end{pmatrix}, \tag{2.1}$$

where A_{FR} has linearly independent rows. The vectors x_{FX} and x_{FR} are known as the associated *fixed*, and *free* components of x. (In practice, x_{FX} may include variables that are temporarily frozen at values strictly *between* their bounds.) At a nonoptimal feasible point x we seek a search direction p such that $x + p$ remains on the set of working-set constraints yet improves the objective function. If the new point is to be feasible, we must have $A_{FR}p_{FR} + A_{FX}p_{FX} = 0$ and $p_{FX} = 0$. These conditions may be expressed compactly as $p = Zp_Z$, where Z is a matrix with columns that span the null space of the working-set matrix A_W, i.e.,

$$Z = P\begin{pmatrix} Z_{FR} \\ 0 \end{pmatrix} \tag{2.2}$$

where P is the permutation matrix of (2.1) that permutes the columns of A_W into the order $\begin{pmatrix} A_{FR} & A_{FX} \end{pmatrix}$. Minimizing $q_k(x)$ with respect to p_Z now involves a quadratic function of p_Z:

$$g^T Z p_Z + \tfrac{1}{2} p_Z^T Z^T H Z p_Z,$$

where $g = \nabla q_k(x)$ and $H = H_k$. 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_Z is computed from the system

$$Z^T HZ p_Z = -Z^T g. \tag{2.3}$$

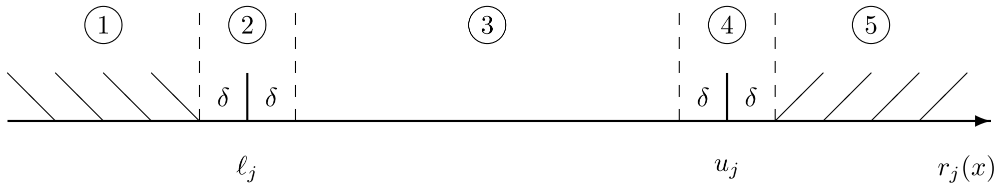


Figure 1: Illustration of the constraints $\ell_j \le r_j(x) \le u_j$. The bounds ℓ_j and u_j are considered "satisfied" if $r_j(x)$ lies in Regions 2, 3 or 4, where δ is the feasibility tolerance. The constraints $r_j(x) \ge \ell_j$ and $r_j(x) \le u_j$ are both considered "inactive" if $r_j(x)$ lies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

Once p_Z is specified, p is uniquely determined from the definition $p = Zp_Z$. The number of components of p_Z (n_Z say) therefore indicates the *number of degrees of freedom* remaining after the constraints have been satisfied. In broad terms, n_Z is a measure of *how nonlinear* the problem is. In particular, n_Z need not be more than one for linear problems.

An orthogonal factorization package is used to maintain TQ factors of A_{FR} as the working set changes. A *TQ factorization* of A_{FR} is given by:

$$A_{FR}Q_{FR} = \begin{pmatrix} 0 & T_{FR} \end{pmatrix}, \tag{2.4}$$

where T_{FR} is a nonsingular $m_W \times m_W$ upper-triangular matrix, and Q_{FR} is an $n_{FR} \times n_{FR}$ nonsingular matrix constructed from a product of orthogonal transformations (see [9]). If the columns of Q_{FR} are partitioned so that

$$Q_{FR} = \begin{pmatrix} Z_{FR} & Y_{FR} \end{pmatrix},$$

where Y_{FR} is $n_{FR} \times m_w$ and Z_{FR} is $n_{FR} \times n_Z$ (where $n_Z = n_{FR} - m_w$).

If $Z^Tg = 0$, then no further improvement can be made with the current working set. In this case, vectors π and z are computed from the equations

$$\begin{pmatrix} & A_{FR}^T \\ I_{FX} & A_{FX}^T \end{pmatrix} \begin{pmatrix} z \\ \pi \end{pmatrix} = \begin{pmatrix} g_{FR} \\ g_{FX} \end{pmatrix},$$

where g_{FR} and g_{FX} are the components of $\nabla q_k(x)$ corresponding to the free and fixed variables. The components of π and z are associated with the general and bound constraints in the working set respectively. These vectors form part of the $n + m$ vector y with components associated with the rows of the QP constraint matrix

$$\begin{pmatrix} I \\ A \end{pmatrix}.$$

In this case, $y_j = 0$ for a constraint that is not in the working set. The components of y are known as the *dual variables* or *Lagrange multipliers*.

The vector d of *dual infeasibilities* is given by $d_j = \max\{-y_j, 0\}$ if constraint j is in the working set at its lower bound, $d_j = \max\{y_j, 0\}$ if constraint j is in the working set at its lower bound, and $d_j = 0$ otherwise. If $d = 0$, then x is optimal for QP_k. Otherwise, a constraint with non-optimal dual variable is selected to be removed from the working set. The iteration is then repeated with n_Z increased by one. At all stages, if the step $x + p$ is not feasible, the largest step α is computed such that $x + \alpha p$ satisfies the constraints not in the working set. At $x + \alpha p$ at least one constraint not in the working set must be active, the number of working-set constraints is increased by one and n_Z is decreased by one. In practice, DNOPT requests an *approximate* QP solution $(\widehat{x}_k, \widehat{y}_k)$ with slightly relaxed conditions on y_j.

The reduced Hessian system (2.3) is solved an upper-triangular matrix R is maintained satisfying $R^TR = Z^THZ$. Normally, R is computed from Z^THZ at the start of phase 2 and is then updated as the working set changes.

2.3. The merit function

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

$$M(x, s, y) = f_0(x) - y^T\big(f(x) - s\big) + \tfrac{1}{2}\big(f(x) - s\big)^T D\big(f(x) - s\big), \tag{2.5}$$

where D is a diagonal matrix of penalty parameters ($D_{ii} \geq 0$), and y now refers to dual variables for the nonlinear constraints in (NP). The vector s is the vector of *nonlinear slacks*. At a solution of (NP), the nonlinear slacks satisfy $f(x) - s = 0$ and $\ell_N \leq s \leq u_N$. At a solution $\widehat{x}_k$ of QP_k a new estimate of the optimal nonlinear slacks is computed such that

$$f(x_k) - f'(x_k)(\widehat{x}_k - x_k) - \widehat{s}_k = 0.$$

If (x_k, s_k, y_k) denotes the current solution estimate and $(\widehat{x}_k, \widehat{s}_k, \widehat{y}_k)$ denotes the QP solution, the linesearch determines a step α_k $(0 < \alpha_k \leq 1)$ such that the new point

$$\begin{pmatrix} x_{k+1} \\ s_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ s_k \\ y_k \end{pmatrix} + \alpha_k \begin{pmatrix} \widehat{x}_k - x_k \\ \widehat{s}_k - s_k \\ \widehat{y}_k - y_k \end{pmatrix} \tag{2.6}$$

gives a *sufficient decrease* in the merit function (2.5). When necessary, the penalties in D are increased by the minimum-norm perturbation that ensures descent for M [12]. Prior to the line search, s_k is adjusted to minimize the merit function as a function of s.

2.4. Treatment of constraint infeasibilities

DNOPT makes explicit allowance for infeasible constraints. After finding a feasible point for the linear constraints and bounds, DNOPT 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 y for the nonlinear constraints become large), DNOPT enters "elastic" mode and thereafter solves the problem

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

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 Sℓ_1QP algorithm of Fletcher [4]. See also Conn [1].

The initial value of γ is controlled by the optional parameter `Elastic weight` (p. 52).

3. The `dnOpt` interface

`dnOpt` is the principal user interface in the DNOPT package. 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 of `dnOpt` is

```
call dnBEGIN( iPrint, iSumm, ...   )
call dnSpec ( iSpecs, ...          )
call dnOpt  ( start, n, mLCon, ... )
call dnEND  ( iPrint, iSumm, ...   )
```

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

3.1. Subroutines used by dnOpt

dnOpt is accessed via the following routines:

dnBEGIN (Section 1.5) must be called before any other dnOpt routines.

dnSpec (Section 6.4) may be called to input a Specs file (a list of run-time options).

dnSet, dnSetInt, dnSetReal (Section 6.5) may be called to specify a single option.

dnGet, dnGetChar, dnGetInt, dnGetReal (Section 6.6) may be called to obtain an option's current value.

dnOpt (Section 3.4) is the main solver.

funcon, funobj (Section 3.5) are supplied by the user and called by dnOpt. funcon and funobj 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 dnOpt as parameters.)

dnMem 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.

3.2. Identifying structure in the objective and constraints

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

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

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 dnOpt 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 iObjA. 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 iObjA of the linear constraint matrix. Choosing iObjA $= 2$, we think of the problem as

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

with bounds $w \geq 0$, $z \geq 0$, $-\infty \leq s_4 \leq \infty$, where s_4 is treated *implicitly* as the value of the 4th constraint.

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 `mNCon` $= 2$. The variables $x'' = (u, v)$ are known as *nonlinear Jacobian variables*, with dimension n_1'' specified by `nnJac` $= 2$. Thus, the combined vector of nonlinear and linear constraint functions has the form

$$\begin{pmatrix} f(x'') + J_2 y'' \\ A_1 x'' + A_2 y'' \end{pmatrix}, \tag{3.1}$$

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 nonlinear and linear Jacobian matrices are then $\begin{pmatrix} f'(x'') & J_2 \end{pmatrix}$ and $\begin{pmatrix} A_1 & A_2 \end{pmatrix}$, and the matrix seen by the QP subproblem has the form

$$\begin{pmatrix} f'(x'') & J_2 \\ A_1 & A_2 \end{pmatrix}, \tag{3.2}$$

with the Jacobian of f always appearing in the *top left corner*. The matrices $f'(x'')$ and J_2 are held in the array parameter `JCon`. The elements of `JCon` corresponding to $f'(x'')$ may be given any value (the correct values are computed internally). The arrays A_1 and A_2 are input via the array parameter `A`. (Elements that are identically zero must be included.)

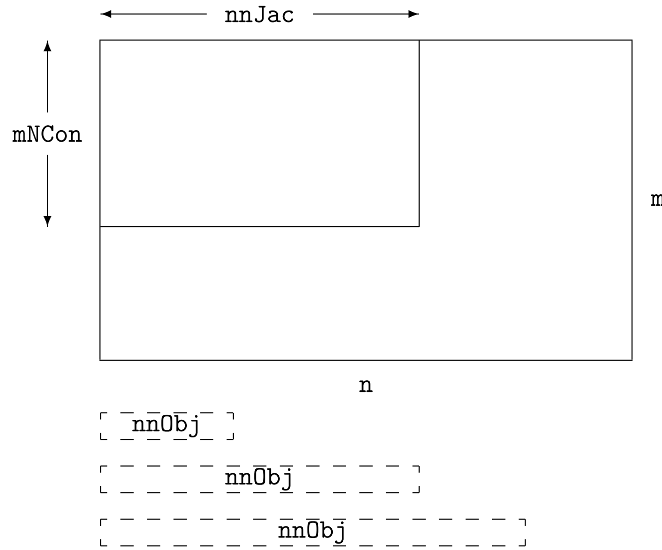
The inequalities $l_1 \leq f(x'') + J_2 y'' \leq u_1$ and $l_2 \leq A_1 x + A_2 y'' \leq u_2$ implied by the constraint functions (3.1) are known as the *nonlinear* and *linear* constraints respectively. Together, these two sets of inequalities constitute the *general constraints*.

In general, the vectors x' and x'' have different dimensions, but they *always overlap*, in the sense that the shorter vector is always the beginning of the other. In the example, the nonlinear Jacobian variables (u, v) are an ordered subset of the nonlinear objective variables (u, v, w). In other cases it could be the other way round—whichever is the most convenient—but the first way keeps $f'(x'')$ 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., $n_1 = \max\{n_1', n_1''\}$ ($= \max($`nnObj`$,$ `nnJac`$)$).

3.3. Problem dimensions

The following picture illustrates the problem structure just described:



The dimensions are all input parameters to subroutine `dnOpt` (see the next section). For linear programs, `mNCon`, `nnJac`, `nnObj` are all zero. If a linear objective term exists, `iObjA` points to one of the rows of A.

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

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

3.4. Subroutine dnOpt

In the following specification of `dnOpt`, we define $r(x)$ as the vector of combined constraint functions $r(x) = \begin{pmatrix} x & f(x) & A_L x \end{pmatrix}$, and use `nb` to denote a variable that holds its dimension: $\texttt{nb} = \texttt{n} + \texttt{mLCon} + \texttt{mNCon}$. 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 dnopt
     &   ( start, n, mLCon, mNCon, nnJac, nnObj,
     &     problemName, Names, nNames, iObjA, objAdd,
     &     funcon, funobj,
     &     state, A, ldA, bl, bu,
     &     fObj, gObj, fCon, JCon, ldJ, H, ldH,
     &     objNP, nInf, sInf, x, y,
     &     INFO, mincw, miniw, minrw,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )
      external
     &     funcon, funobj
      integer
     &     iObjA, INFO, ldA, ldJ, ldH, lencw, leniw, lenrw, lencu,
     &     leniu, lenru, n, mLCon, mNCon, nnJac, nnObj, nNames, nInf,
     &     mincw, miniw, minrw, start, state(n+mNCon+mLCon), iu(leniu),
     &     iw(leniw)
      double precision
     &     objAdd, objNP, fObj, sInf, A(ldA,*), bl(n+mNCon+mLCon),
     &     bu(n+mNCon+mLCon), gObj(n), fCon(ldJ), JCon(ldJ,*), H(ldH,*),
     &     x(n+mNCon+mLCon), y(n+mNCon+mLCon), ru(lenru), rw(lenrw)
      character
     &     problemName*8, Names(nNames)*8, cu(lencu)*8, cw(lencw)*8
```

On entry:

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

- `Start = 0` (Cold start) requests that a the crash procedure be used to define an initial working set.
- `Start = 1` (Warm start) means that `state` defines a valid starting point (perhaps from an earlier call, though not necessarily).
- `Start = 2` (Hot start) means that `state` defines a valid starting point and the argument `H(ldH,*)` defines a positive-definite approximate Hessian of the Lagrangian.

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

`mLCon` is m_L, the number of general linear constraints ($\texttt{mLCon} \geq 0$).

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

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

`ldJ` is the row dimension of the array `JCon` ($\texttt{ldJ} \geq 1$, $\texttt{ldJ} \geq \texttt{mNCon}$).

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

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

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

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

`iObjA` says which row of A is a free row containing a linear objective vector c. If there is no such row, `iObjA` $= 0$.

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

`ProblemName` is an 8-character name for the problem. `ProblemName` is used in the printed solution. A blank name may be used.

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

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

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

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

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

`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 `dnOpt`. For a detailed description of `funcon`, see Section 3.6.

`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 `dnOpt`. For a detailed description of `funobj`, see Section 3.7.

`state(nb)` is an integer array that need not be initialized if `dnOpt` is called with `start` $= 0$ (a cold start) or the `Cold start` option (the default).

For `start` $= 1$ (a warm start) every element of `state` must be set. If `dnOpt` has just been called on a problem with the same dimensions, `state` already contains valid values. Otherwise, `state`(j) should indicate whether either of the constraints $r_j(x) \geq l_j$ or $r_j(x) \leq u_j$ is expected to be active at a solution of (NP).

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

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

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

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

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

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

`JCon(ldJ,*)` is an array of dimension (`ldJ`,k) for some $k \geq \mathtt{n}$. If `mNCon` $= 0$, `JCon` is not referenced. (In that case, `JCon` may be dimensioned `(ldJ,1)` with `ldJ` $= 1$.)

In general, `JCon` need not be initialized before the call to `dnOpt`. However, if `Derivative level` $= 3$, any constant elements of `JCon` may be initialized. Such elements need not be reassigned on subsequent calls to `funcon` (see Section 3.8).

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

`x(nb)` usually contains a set of initial values for x.

1. For Cold starts (`Start` $= 0$), the first n elements of `state` and `x` must be defined.

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

   Less trivially, to say that the optimal value of variable j will probably be equal to one of its bounds, set $\mathtt{state}(j) = 1$ and $\mathtt{x}(j) = \mathtt{bl}(j)$ or $\mathtt{state}(j) = 2$ and $\mathtt{x}(j) = \mathtt{bu}(j)$ as appropriate.

   A CRASH procedure is used to select a working set.

2. For Warm or Hot starts (`Start` $= 1, 2$), all of $\mathtt{state}(1\!:\!\mathtt{nb})$ must be 0, 1 or 2 and $\mathtt{x}(1\!:\!\mathtt{n})$ must have values (perhaps from some previous call).

`y(nb)` is an array that need not be initialized if `dnOpt` is called with a `Cold start` (the default).

Otherwise, the ordering of `y` is the same as for `bl`, `bu` and `state`. For a `Warm start`, the components of `y` corresponding to nonlinear constraints must contain a multiplier estimate. The sign of each multiplier should match `state` as follows. If the ith nonlinear constraint is defined as "inactive" via the initial value $\mathtt{state}(j) = 0$, $j = \mathtt{n} + i$, then $\mathtt{y}(j)$ should be zero. If the nonlinear constraint $r_j(x) \geq l_j$ is active ($\mathtt{state}(j) = 1$), $\mathtt{y}(j)$ should be non-negative, and if $r_j(x) \leq u_j$ is active ($\mathtt{state}(j) = 2$), $\mathtt{y}(j)$ should be non-positive.

If necessary, `dnOpt` will change `y` to match these rules.

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

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 `dnOpt`'s workspace.

`cw(lencw), iw(leniw), rw(lenrw)` are 8-character, integer and real arrays of workspace for `dnOpt`. Their lengths `lencw`, `leniw`, `lenrw` must all be at least 500.

In general, `lencw` = 500 is appropriate. Appropriate values of `leniw` and `lenrw` may be obtained from a preliminary run with `lencw` = `leniw` = `lenrw` = 500. See `mincw`, `miniw`, `minrw` below (on exit).

On exit:

`INFO` reports the result of the call to `dnOpt`. Here is a summary of possible values. Further details are in Section 7.4.

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 at a feasible point
22 constraint violation limit reached

Resource limit error
31 iteration limit reached
32 major iteration limit reached

Terminated after numerical difficulties
41 current point cannot be improved
42 ill-conditioned working set
43 cannot satisfy the working-set constraints
44 Reduced gradient too large

Error in the user-supplied functions
51 incorrect objective derivatives
52 incorrect constraint derivatives
53 the QP Hessian is indefinite
57 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
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

`iter` is the number of major iterations performed.

`state` describes the status of the constraints $l \le r(x) \le u$ in problem (NP). For the jth lower or upper bound, $j = 1$ to `nb`, the possible values of `state`(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 `state` are labeled in the printed solution as follows:

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

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

`JCon` contains the Jacobian matrix of the nonlinear constraints at the final iterate, i.e., `JCon`(i, j) contains the partial derivative of the ith constraint function with respect to the jth variable, $i = 1$: `mNCon`, $j = 1$: `n`. (See the discussion of `JCon` under `funcon` in Section 3.6.)

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

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

`x` contains the final estimate of the solution.

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.

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 `dnOpt` terminates because of insufficient storage (`INFO = 82`, 83 or 84), `mincw`, `miniw` and `minrw` give the required values of `lencw`, `leniw` or `lenrw`.

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

If `INFO = 83` or 84, the work arrays `iw(leniw)` or `rw(lenrw)` are too small. `dnOpt` may be called again with `leniw` = `miniw` and `lenrw` = `minrw`.

3.5. User-supplied subroutines for dnOpt

The user must provide subroutines to define the nonlinear parts of the objective function and nonlinear constraints. They are passed to `dnOpt` 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 `dnOpt`: 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. `dnOpt` 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, `dnOpt` 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 `status` 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` ≤ -2 if you want `dnOpt` to stop.

3.6. 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 `JCon` 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, mNCon, nnJac, x, fCon, JCon, ldJ, status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     ldJ, mode, mNCon, nnJac, status, lencu, leniu, lenru,
     &     iu(leniu)
      double precision
     &     x(nnJac), fCon(mNCon), JCon(ldJ,*), ru(lenru)
      character*8
     &     cu(lencu)
```

On entry:

`mode` indicates whether `fCon` or `JCon` 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 `JCon` must be assigned (except perhaps constant elements of `JCon`).

Otherwise, `dnOpt` 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 `JCon`.
- If `mode` = 1, assign the known components of `JCon`. `fCon` is ignored.
- If `mode` = 0, only `fCon` need be assigned; `JCon` is ignored.

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

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

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

If `status` = 0, there is nothing special about the current call to `funcon`.

If `status` = 1, `dnOpt` 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 `status` ≥ 2, `dnOpt` 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 `status` $= 2 +$ `INFO`/10, where `INFO` is the integer returned by `dnOpt` (see p. 17). 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`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `dnOpt`. 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 `dnOpt`'s workspace arrays `cw`, `iw`, `rw`. For example, the 8-character problem name `ProblemName` is stored in `cw(51)`, and the dual variables for the general constraints are stored in `rw(lyCon)` onward, where `lyCon` = `iw(268)`. These will be accessible to both `funcon` and `funobj` if `dnOpt` 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(mNCon)` contains the computed constraint vector $f(x)$ (except perhaps if `mode` = 1).

`JCon(mNCon,nnJac)` contains the computed Jacobian $f'(x)$ (except perhaps if `mode` = 0). These gradient elements *must not include the elements of* J_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, `dnOpt` 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.

3.7. Subroutine `funobj`

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

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

On entry:

`mode` indicates whether `fObj` or `gObj` or both must be assigned during the present call of `funobj` ($0 \le$ `mode` ≤ 2).

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

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

- If `mode` $= 2$, assign `fObj` and the known components of `gObj`.
- If `mode` $= 1$, assign the known components of `gOb`. `fObj` is ignored.
- If `mode` $= 0$, only `fObj` need be assigned; `gObj` is ignored.

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

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

`status` 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` $= 0$).

3.8. 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 `JCon` one time only at the start of the optimization. An element of `JCon` that is not subsequently assigned in `funcon` will retain its initial value throughout. Constant elements may be loaded into `JCon` either *before* the call to `dnOpt` or during the the first call to `funcon` (signalled by the value `status` $= 1$). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case `JCon` 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 `JCon`(i,j) is set to a constant value, it need not be reset in subsequent calls to `funcon`, but the value `JCon`(i,j)`*x`(j) must nonetheless be added to `fCon`(i).

It must be emphasized that, if `Derivative level` < 2, unassigned elements of `JCon` are *not* treated as constant; they are estimated by finite differences, at non-trivial expense. indext`funcon`!assigning constant constraint derivatives indexttDerivative level

3.9. Example

Here we give the subroutines `funobj` and `funcon` for the example of Section 3.2, repeated here for convenience with generic variables x_j:

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

and $x_3 \geq 0$, $x_4 \geq 0$. 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:

```
n = 4   mLCon = 2   mNCon = 2    nnObj = 3    nnJac = 2    iObjA = 2
```

Subroutine `funobj` works with the nonlinear objective variables (x_1, x_2, x_3). As 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, Status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     mode, nnObj, Status, lencu, leniu,  lenru, iu(leniu)
      double precision
     &     fObj, x(nnObj), gObj(nnObj), ru(lenru)
      character*8
     &     cu(lencu)
!     ==================================================================
!     Simple toy Problem.
!     ==================================================================
      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). As `funcon` is called before `funobj`, we test `status` for the first and last entries.

```
      subroutine funcon
     &   ( mode, mNCon, nnJac, x, fCon, JCon, ldJ, status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     ldJ, mode, mNCon, nnJac, status, lencu, leniu, lenru,
     &     iu(leniu)
      double precision
     &     x(nnJac), fCon(mNCon), JCon(ldJ,*), ru(lenru)
      character*8
     &     cu(lencu)

!     ==================================================================
!     Simple toy Problem.
!     ==================================================================
      integer            Out
      parameter         (Out = 6)
!     ------------------------------------------------------------------
!     Print something on the first and last entry.

      if (status .eq. 1) then       ! First
         if (Out .gt. 0) write(Out, '(/a)') ' Starting  dntoy'
      else  if (status .ge. 2) then ! Last
         if (Out .gt. 0) write(Out, '(/a)') ' Finishing dntoy'
         return
      end if

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

      if (mode .ge. 1) then
!        -------------------------------------------
!        Nonlinear elements for row 1
!        -------------------------------------------
         JCon(1,1) = 2.0d+0*x(1)
         JCon(1,2) = 2.0d+0*x(2)

!        -------------------------------------------
!        Nonlinear elements for row 2
!        -------------------------------------------
         JCon(2,2) = 4.0d+0*x(2)**3
      end if

      end ! subroutine funcon
```

4. The dnOptH interface

dnOptH implements a second-derivative SQP method. 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 of dnOptH is

```
call dnBEGIN( iPrint, iSumm, ...   )
```

```
call dnSpec ( iSpecs, ...          )
call dnOptH ( start, n, mLCon, ... )
call dnEND  ( iPrint, iSumm, ...   )
```

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

4.1. Subroutines used by `dnOptH`

`dnOptH` is accessed via the following routines:

`dnBEGIN` (Section 1.5) must be called before any other `dnOpt` routines.

`dnSpec` (Section 6.4) may be called to input a Specs file (a list of run-time options).

`dnSet`, `dnSetInt`, `dnSetReal` (Section 6.5) may be called to specify a single option.

`dnGet`, `dnGetChar`, `dnGetInt`, `dnGetReal` (Section 6.6) may be called to obtain an option's current value.

`dnOptH` (Section 4.2) is the main solver.

`funcon`, `funobj`, `funhes` are supplied by the user and called by `dnOptH`. Subroutines `funcon` and `funobj` are identical to the routines called by `dnOpt` (Section 3.5). `funcon` and `funobj` define the constraint functions $f(x)$ and objective function $f_0(x)$ and ideally their gradients. `funhes` computes the Hessian of the Lagrangian function with respect to x. The routines `funcon`, `funobj` and `funhes` are passed to `dnOptH` as parameters. They have a fixed parameter list but may have any convenient name.

`dnMem` 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.

4.2. Subroutine dnOptH

The specification of dnOptH is identical to that of dnOpt with the exception of one external parameter funhes.

```
      subroutine dnOptH
     &   ( start, n, mLCon, mNCon, nnJac, nnObj,
     &     problemName, Names, nNames, iObjA, objAdd,
     &     funcon, funobj, funhes,
     &     state, A, ldA, bl, bu,
     &     fObj, gObj, fCon, JCon, ldJ, H, ldH,
     &     objNP, nInf, sInf, x, y,
     &     INFO, mincw, miniw, minrw,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )
      external
     &     funcon, funobj, funhes
      integer
     &     iObjA, INFO, ldA, ldJ, ldH, lencw, leniw, lenrw, lencu,
     &     leniu, lenru, n, mLCon, mNCon, nnJac, nnObj, nNames, nInf,
     &     mincw, miniw, minrw, start, state(n+mNCon+mLCon), iu(leniu),
     &     iw(leniw)
      double precision
     &     objAdd, objNP, fObj, sInf, A(ldA,*), bl(n+mNCon+mLCon),
     &     bu(n+mNCon+mLCon), gObj(n), fCon(ldJ), JCon(ldJ,*), H(ldH,*),
     &     x(n+mNCon+mLCon), y(n+mNCon+mLCon), ru(lenru), rw(lenrw)
      character
     &     problemName*8, Names(nNames)*8, cu(lencu)*8, cw(lencw)*8
```

On entry:

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

- Start = 0 (Cold start) requests that a the crash procedure be used to define an initial working set.
- Start = 1 (Warm start) means that state defines a valid starting point (perhaps from an earlier call, though not necessarily).
- Start = 2 (Hot start) means that state defines a valid starting point and the argument H(ldH,*) defines a positive-definite approximate Hessian of the Lagrangian.

n is n, the number of variables in the problem (n > 0).

mLCon is the number of general linear constraints (mLCon ≥ 0).

mNCon is the number of nonlinear constraints (mNCon ≥ 0).

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

ldJ is the row dimension of the array JCon (ldJ ≥ 1, ldJ $\geq$ mNCon).

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

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

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

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

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

`iObjA` says which row of A is a free row containing a linear objective vector c. If there is no such row, `iObjA` $= 0$.

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

`ProblemName` is an 8-character name for the problem. `ProblemName` is used in the printed solution. A blank name may be used.

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

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

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

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

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

`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 `dnOptH`. For a detailed description of `funcon`, see Section 3.6.

`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 `dnOptH`. For a detailed description of `funobj`, see Section 3.7.

`funhes` is the name of a subroutine that calculates the Hessian matrix of the function $f_0(x) - y^T f(x)$ for a given vector y. `funhes` must be declared `external` in the routine that calls `dnOptH`. For a detailed description of `funhes`, see Section 4.4.

`state(nb)` is an integer array that need not be initialized if `dnOptH` is called with `start` $= 0$ (a cold start) or the `Cold start` option (the default).

For `start` $= 1$ (a warm start) every element of `state` must be set. If `dnOptH` has just been called on a problem with the same dimensions, `state` already contains valid values. Otherwise, `state`(j) should indicate whether either of the constraints $r_j(x) \geq l_j$ or $r_j(x) \leq u_j$ is expected to be active at a solution of (NP).

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

0 Neither $r_j(x) \ge l_j$ nor $r_j(x) \le u_j$ is expected to be active.

1 $r_j(x) \ge l_j$ is expected to be active.

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

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

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

JCon(ldJ,*) is an array of dimension (ldJ,k) for some $k \ge$ n. If mNCon = 0, JCon is not referenced. (In that case, JCon may be dimensioned (ldJ,1) with ldJ = 1.)

In general, JCon need not be initialized before the call to dnOptH. However, if Derivative level = 3, any constant elements of JCon may be initialized. Such elements need not be reassigned on subsequent calls to funcon (see Section 3.8).

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

x(nb) usually contains a set of initial values for x.

1. For Cold starts (Start = 0), the first n elements of state and x must be defined.

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

   Less trivially, to say that the optimal value of variable j will probably be equal to one of its bounds, set state(j) = 1 and x(j) = bl(j) or state(j) = 2 and x(j) = bu(j) as appropriate.

   A CRASH procedure is used to select a working set.

2. For Warm or Hot starts (Start = 1, 2), all of state(1 : nb) must be 0, 1 or 2 and x(1 : n) must have values (perhaps from some previous call).

y(nb) is an array that need not be initialized if dnOptH is called with a Cold start (the default).

Otherwise, the ordering of y is the same as for bl, bu and state. For a Warm start, the components of y corresponding to nonlinear constraints must contain a multiplier estimate. The sign of each multiplier should match state as follows. If the ith nonlinear constraint is defined as "inactive" via the initial value state(j) = 0, j = n + i, then y(j) should be zero. If the nonlinear constraint $r_j(x) \ge l_j$ is active (state(j) = 1), y(j) should be non-negative, and if $r_j(x) \le u_j$ is active (state(j) = 2), y(j) should be non-positive.

If necessary, dnOptH will change y to match these rules.

cu(lencu), iu(leniu), ru(lenru) are 8-character, integer and real arrays of user workspace. They may be used to pass data or workspace to your function routines funcon and funobj (which have the same parameters). They are not touched by dnOptH.

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 `dnOptH`'s workspace.

`cw(lencw), iw(leniw), rw(lenrw)` are 8-character, integer and real arrays of workspace for `dnOptH`. Their lengths `lencw`, `leniw`, `lenrw` must all be at least 500.

In general, `lencw` = 500 is appropriate. Appropriate values of `leniw` and `lenrw` may be obtained from a preliminary run with `lencw` = `leniw` = `lenrw` = 500. See `mincw`, `miniw`, `minrw` below (on exit).

On exit:

`INFO` reports the result of the call to `dnOptH`. Here is a summary of possible values. Further details are in Section 7.4.

	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 at a feasible point
22	constraint violation limit reached
	Resource limit error
31	iteration limit reached
32	major iteration limit reached
	Terminated after numerical difficulties
41	current point cannot be improved
42	ill-conditioned working set
43	cannot satisfy the working-set constraints
44	Reduced gradient too large
	Error in the user-supplied functions
51	incorrect objective derivatives
52	incorrect constraint derivatives
53	the QP Hessian is indefinite
57	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
- 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

`iter` is the number of major iterations performed.

`state` describes the status of the constraints $l \le r(x) \le u$ in problem (NP). For the jth lower or upper bound, $j = 1$ to `nb`, the possible values of `state`(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 `state` are labeled in the printed solution as follows:

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

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

`JCon` contains the Jacobian matrix of the nonlinear constraints at the final iterate, i.e., `JCon`(i, j) contains the partial derivative of the ith constraint function with respect to the jth variable, $i = 1$: `mNCon`, $j = 1$: `n`. (See the discussion of `JCon` under `funcon` in Section 3.6.)

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

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

`x` contains the final estimate of the solution.

`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.

`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 `dnOptH` terminates because of insufficient storage (`INFO = 82`, 83 or 84), `mincw`, `miniw` and `minrw` give the required values of `lencw`, `leniw` or `lenrw`.

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

If `INFO = 83` or 84, the work arrays `iw(leniw)` or `rw(lenrw)` are too small. `dnOptH` may be called again with `leniw` = `miniw` and `lenrw` = `minrw`.

4.3. User-supplied subroutines for `dnOptH`

The user must provide subroutines to define the nonlinear parts of the objective function and nonlinear constraints. They are passed to `dnOptH` 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 `dnOptH`: 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. `dnOptH` 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, `dnOptH` 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 `status` 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` ≤ -2 if you want `dnOptH` to stop.

4.4. Subroutine funhes

This subroutine must calculate the Hessian with respect to x of the function $f_0(x) - y^T f(x)$.

```
      subroutine funhes
     &   ( mode, nnH, mNCon, x, y, H, ldH, status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     mode, nnH, mNCon, status, lencu, leniu, lenru, iu(leniu)
      double precision
     &     y(mNCon), H(ldH,*), x(nnH), ru(lenru)
      character*8
     &     cu(lencu)
```

On entry:

`mode` will call `funhes` with `mode` $= 0$. You may test `mode` to decide what to do:

- If `mode` $= 0$, `H` should be assigned the Hessian of the Lagrangian function $f_0(x) - y^T f(x)$.

`nnH` is the number of variables involved in the Lagrangian function (`nnH` $= \max($`nnObj`, `nnJac`$)$).

`mNCon` is m_N, the number of nonlinear constraints (`mNCon` > 0).

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

`y(mNCon)` contains the multipliers for the nonlinear constraints.

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

`status` 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 `funhes` to indicate that you are unable to evaluate H at x.

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

`H(ldH,*)` must contain the requested Hessian matrix.

4.5. Example

Here we give the subroutines `funobj` and `funcon` for the example of Section 3.2, repeated here for convenience with generic variables x_j:

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

and $x_3 \geq 0$, $x_4 \geq 0$. 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:

```
n = 4   mLCon = 2   mNCon = 2    nnObj = 3    nnJac = 2    iObjA = 2
```

Subroutine `funobj` works with the nonlinear objective variables (x_1, x_2, x_3). As 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, Status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     mode, nnObj, Status, lencu, leniu,  lenru, iu(leniu)
      double precision
     &     fObj, x(nnObj), gObj(nnObj), ru(lenru)
      character*8
     &     cu(lencu)
!     ==================================================================
!     Simple toy Problem.
!     ==================================================================
      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). As funcon is called before funobj, we test status for the first and last entries.

```
      subroutine funcon
     &   ( mode, mNCon, nnJac, x, fCon, JCon, ldJ, status,
     &     cu, lencu, iu, leniu, ru, lenru )
      integer
     &     ldJ, mode, mNCon, nnJac, status, lencu, leniu, lenru,
     &     iu(leniu)
      double precision
     &     x(nnJac), fCon(mNCon), JCon(ldJ,*), ru(lenru)
      character*8
     &     cu(lencu)

!     ==================================================================
!     Simple toy Problem.
!     ==================================================================
      integer            Out
      parameter         (Out = 6)
!     ------------------------------------------------------------------
!     Print something on the first and last entry.

      if (status .eq. 1) then       ! First
         if (Out .gt. 0) write(Out, '(/a)') ' Starting  dntoy'
      else  if (status .ge. 2) then ! Last
         if (Out .gt. 0) write(Out, '(/a)') ' Finishing dntoy'
         return
      end if

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

      if (mode .ge. 1) then
!        -------------------------------------------
!        Nonlinear elements for row 1
!        -------------------------------------------
         JCon(1,1) = 2.0d+0*x(1)
         JCon(1,2) = 2.0d+0*x(2)

!        -------------------------------------------
!        Nonlinear elements for row 2
!        -------------------------------------------
```

```
      JCon(2,2) = 4.0d+0*x(2)**3
   end if

   end ! subroutine funcon
```

5. The dnNPSOL interface

The `dnNPSOL` interface is designed for the solution of small dense problems. The calling sequences of `dnNPSOL` 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 `dnNPSOL` it is convenient to restate problem NP with the constraints reordered as follows:

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

where l and u are constant lower and upper bounds, f_0 is a smooth scalar objective function, A_L is a matrix, and $f(x)$ is a vector of smooth nonlinear constraint functions $\{f_i(x)\}$. The interface `dnNPSOL` 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 `dnNPSOL` is

```
call dnBEGIN( iPrint, iSumm, ...   )
call dnSpec ( iSpecs, ...          )
call dnNPSOL( n, nclin, ncnln, ... )
call dnEND  ( iPrint, iSumm, ...   )
```

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

5.1. Subroutines used by dnNPSOL

`dnNPSOL` is accessed via the following routines:

`dnBEGIN` (Section 1.5) must be called before any other `dnNPSOL` routines.

`dnSpec` (Section 6.4) may be called to input a Specs file (a list of run-time options).

`dnSet`, `dnSetInt`, `dnSetReal` (Section 6.5) may be called to specify a single option.

`dnGet`, `dnGetChar`, `dnGetInt`, `dnGetReal` (Section 6.6) may be called to obtain an option's current value.

`dnNPSOL` (Section 3) is the main solver.

`funcon`, `funobj` (Section 5.3) are supplied by the user and called by `dnNPSOL`. 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 `dnNPSOL` as parameters.)

`dnnpsolmem` 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.

5.2. Subroutine dnNPSOL

In the following specification of `dnNPSOL`, 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 dnNPSOL
     &   ( n, nclin, ncnln, ldA, ldJ, ldH,
     &     A, bl, bu, funcon, funobj,
     &     INFO, majIts, iState,
     &     fCon, JCon, cMul, fObj, gObj, Hess, x,
     &     iw, leniw, rw, lenrw )

      external
     &     funcon, funobj
      integer
     &     INFO, ldA, ldJ, 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(*), JCon(ldJ,*), 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 (`ncnln` > 0).

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

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

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

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

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

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

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

`funcon, funobj` are the names of subroutines that calculate the nonlinear constraint functions $f(x)$, the objective function $f_0(x)$ and (optionally) their gradients for a specified n-vector x. The arguments `funcon` and `funobj` must be declared as `external` in the routine that calls `dnNPSOL`. See Sections 5.3–5.5.

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

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

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

- 0 Neither $r_j(x) \ge \ell_j$ nor $r_j(x) \le u_j$ is expected to be active.
- 1 $r_j(x) \ge \ell_j$ is expected to be active.
- 2 $r_j(x) \le u_j$ is expected to be active.
- 3 This may be used if $\ell_j = u_j$. Normally an equality constraint $r_j(x) = \ell_j = u_j$ is active at a solution.

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

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

In general, `JCon` need not be initialized before the call to `dnNPSOL`. However, if `Derivative level` $= 3$, any constant elements of `JCon` may be initialized. Such elements need not be reassigned on subsequent calls to `funcon` (see Section 5.6).

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

Otherwise, the ordering of `cMul` is the same as for `bl`, `bu` and `istate`. For a `Warm start`, the components of `cMul` corresponding to nonlinear constraints must contain a multiplier estimate. The sign of each multiplier should match `istate` as follows. If the ith nonlinear constraint is defined as "inactive" via the initial value `istate`$(j) = 0$, $j =$ `n` $+$ `nclin` $+ i$, then `cMul`(j) should be zero. If the constraint $r_j(x) \ge \ell_j$ is active (`istate`$(j) = 1$), `cMul`(j) should be non-negative, and if $r_j(x) \le u_j$ is active (`istate`$(j) = 2$), `cMul`(j) should be non-positive.

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

`Hess(ldH,*)` is an array of dimension `(ldH,`k`)` for some $k \ge$ `n`. `Hess` need not be initialized if `dnNPSOL` 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., $\mathtt{H}(i,j) \approx \partial^2\mathcal{L}(x,\lambda)/\partial x_i \partial x_j$, 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 `dnNPSOL`.

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 dnNPSOL terminates with INFO = 43 or 44.

On exit:

`INFO` reports the result of the call to `dnNPSOL`. Here is a summary of possible values. Further details are in Section 7.4.

	Finished successfully
1	optimality conditions satisfied
2	feasible point found
3	requested accuracy could not be achieved
	The problem appears to be infeasible
11	infeasible linear constraints
12	infeasible linear equalities
13	nonlinear infeasibilities minimized
14	infeasibilities minimized
	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
	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
	Error in the user-supplied functions
51	incorrect objective derivatives
52	incorrect constraint derivatives
	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 $\ell \le r(x) \le u$ 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.

`JCon` contains the Jacobian matrix of the nonlinear constraints at the final iterate, i.e., `JCon`(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 `JCon` under `funcon` in Section 5.5.)

`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.

5.3. User-supplied subroutines for dnNPSOL

The user must provide subroutines that define the objective function and nonlinear constraints. The objective function is defined by subroutine `funobj`, and the nonlinear constraints are defined by subroutine `funcon`. *On every call*, these subroutines must return appropriate values of the objective and nonlinear constraints in `fObj` and `fCon`. The user should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see Section 6 for a discussion of the optional parameter `Derivative level`. Just before either `funobj` or `funcon` is called, each element of the current gradient array `g` or `JCon` 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 [13] 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. 60) should be used to check the calculation of any known gradients.

5.4. 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 `dnNPSOL` 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, `dnNPSOL` 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 `dnNPSOL`, 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, `dnNPSOL` 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, `dnNPSOL` 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).

5.5. 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 JCon 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, ldJ,
     &     needc, x, fCon, JCon, nState )

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

On entry:

mode is set by dnNPSOL 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 6). If some elements of JCon are unspecified, dnNPSOL 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 JCon).
- If mode = 1, the knowm components of the rows of JCon corresponding to positive values in needc must be set. Other rows of JCon 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 JCon 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 dnNPSOL, 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 dnNPSOL, and *must not be altered by funcon.*

ldJ is the leading dimension of the array JCon (ldJ $\geq$ 1 and ldJ $\geq$ ncnln).

needc is an array of dimension at least ncnln containing the indices of the elements of fCon or JCon 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.)

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

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

5.6. 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 `JCon` one time only at the start of the optimization. An element of `JCon` that is not subsequently assigned in `funcon` will retain its initial value throughout. Constant elements may be loaded into `JCon` either *before* the call to `dnNPSOL` 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 `JCon` 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 `JCon`(i,j) is set to a constant value, it need not be reset in subsequent calls to `funcon`, but the value `JCon`(i,j)`*x`(j) must nonetheless be added to `fCon`(i).

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

6. Optional parameters

The performance of each DNOPT 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 `dnSpec` to read a Specs file (Section 6.1).
- By calling the option-setting routines `dnSet`, `dnSetInt`, `dnSetReal` (Section 6.5).

The current value of an optional parameter may be examined by calling one of the routines `dnGet`, `dnGetChar`, `dnGetInt`, `dnGetReal` (Section 6.6).

6.1. 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.

6.2. 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 `dnSpec` from reading past that point in the Specs file while looking for `Begin`.

6.3. 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 subroutine dnBEGIN
  Summary file                  ?       * specified by subroutine dnBEGIN
  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)
  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       *
  Stop  objective  check at col  n1'     *
  Start constraint check at col  1       *
  Stop  constraint check at col  n1''    *

* Scaling
  Scale option                   1       * linear constraints and variables
  Scale tolerance                0.9     *
* Scale Print                            * default: scales are not printed

* Other Tolerances
  Crash tolerance                0.1     *
  Linesearch tolerance           0.9     * smaller for more accurate search
  Pivot tolerance                3.7e-11 * ε^(2/3)

* QP subproblems
  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     *
  Derivative level               3       *
  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)
  Penalty parameter              0.0     * initial penalty parameter
  Proximal point method          1       * satisfies linear constraints near x0
  Violation limit               10.0     * unscaled constraint violation limit
  Unbounded step size            1.0e+18 *
  Unbounded objective            1.0e+15 *

* Hessian approximation
  Hessian frequency         999999       * for full Hessian (never reset)
  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

* 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
End of SPECS file checklist
```

6.4. Subroutine dnSpec

Subroutine `dnSpec` may be called to input a Specs file (to specify options for a subsequent call of DNOPT).

```
      subroutine dnSpec
     &   ( 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 `dnSpec` is called.

On exit:

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

`INFO` reports the result of calling `dnSpec`. 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. `dnSpec` encountered end-of-file or `Endrun` before finding `Begin` (see Section 6.2). 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 $i =$ `INFO` $- 134$ errors while reading the Specs file.

6.5. Subroutines dnSet, dnSetInt, dnSetReal

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

```
      subroutine dnSet
     &   ( buffer,         iPrint, iSumm, Errors,
     &                     cw, lencw, iw, leniw, rw, lenrw )
      subroutine dnSetInt
     &   ( buffer, ivalue, iPrint, iSumm, Errors,
     &                     cw, lencw, iw, leniw, rw, lenrw )
      subroutine dnSetReal
     &   ( 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)` $\leq$ 72 (`dnSet`) or $\leq$ 55 (`dnSetInt`, `dnSetReal`). Use `dnSet` if the string contains all relevant data. For example,

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

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

```
itnlim = 1000
if (m .gt. 500) itnlim = 8000
call dnSetInt( '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 dnSetReal( '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.

6.6. Subroutines dnGet, dnGetChar, dnGetInt, dnGetReal

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

```
      integer function dnGet
     &   ( buffer,         Errors, cw, lencw, iw, leniw, rw, lenrw )
      subroutine dnGetChar
     &   ( buffer, cvalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
      subroutine dnGetInt
     &   ( buffer, ivalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
      subroutine dnGetReal
     &   ( 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:

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

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

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

`cvalue` is a string associated with the keyword in `buffer`. Use `dnGetChar` to obtain the names associated with an MPS file. For example, for the name of the bounds section use

```
call dnGetChar( 'Bounds', MyBounds, Errors, ... )
```

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

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

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

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

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

6.7. 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.

`Central difference interval` r Default = $\epsilon^{1/3} \approx$ `6.0e-6`

When `Derivative level` < 3) with `dnOpt`, 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 working-set 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 working set is refactorized and the 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 working set.

This parameter has the same effect as the input arguments `start = 0` for `dnOpt`. 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 tolerance` t Default = 0.1

This value is used in conjunction with the optional parameter `Cold start` (the default value). When making a cold start, the QP algorithm in `dnOpt` must select an initial working set. When $r \geq 0$, the initial working set will include (if possible) bounds or general inequality constraints that lie within r of their bounds. In particular, a constraint of the form $a_j^T x \geq l$ will be included in the initial working set if $|a_j^T x - l| \leq r(1 + |l|)$. If $r < 0$ or $r > 1$, the default value is used.

`Derivative level` i Default = 3

The keyword `Derivative level` specifies which nonlinear function gradients are known analytically and will be supplied to DNOPT 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, DNOPT 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, DNOPT 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 the Jacobian happens to be

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

where `*` indicates a known gradient and `?` indicates an unknown element, DNOPT 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 to estimate the elements in 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 linesearch is used to improve the merit function. A `Derivative linesearch` uses safeguarded cubic interpolation and requires both function and gradient values to compute estimates of the step α_k. If some analytic derivatives are not provided, or a `Nonderivative linesearch` is specified, DNOPT employs a linesearch based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

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

The selection of `Nonderivative linesearch` for `dnOpt` means that `funobj` and `funcon` are called with `mode` $= 0$ in the linesearch. Once the linesearch is completed, the problem functions are called again with `mode` $= 2$. If the potential savings provided by a nonderivative linesearch are to be realized, it is essential that `funobj` and `funcon` be coded so that the derivatives are not computed when `mode` $= 0$.

`Difference interval` h_1 Default $= \epsilon^{1/2} \approx$ `1.5e-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.

`Elastic weight` ω Default $= 10^4$

This keyword determines the initial weight γ associated with problem NP(γ) on p. 10.

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` k Default $= 10000$

This option is part of the EXPAND anti-cycling procedure [11] 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 DNOPT 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 degree of freedom. (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 degree of freedom, 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` k Default $= 100$ (LP) or 50 (NP)

At most k constraint changes will occur between factorizations of the working set matrix.

- With linear programs, the working-set 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 updates to the working set will occur as an optimum is approached. The number of iterations between working set 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 working set will be refactorized before the limit of k updates is reached.

`Feasible point`
see `Minimize`

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

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

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

- The default value of ϵ_R is appropriate for simple analytic functions.
- In some cases the function values will be the result of extensive computation, possibly involving an iterative procedure that can provide rather few digits of precision at reasonable cost. Specifying an appropriate `Function precision` may lead to savings, by allowing the 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`

`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 $r \le 0$, 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 $0.0 \le t \le 1.0$.
- 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.

`Log frequency` k Default = `100`

see `Print frequency`

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

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

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

$$\texttt{rowerr} = \max_i \ \texttt{viol}_i/\|x\| \ \le \ \epsilon_r, \tag{6.1}$$

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

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, DNOPT will have computed a solution (x, s, π) such that

$$\texttt{maxComp} = \max_j \ \texttt{Comp}_j/\|\pi\| \ \le \ \epsilon_d, \tag{6.2}$$

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):

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

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

`Major print level` p Default = 00001

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

In general, the value being specified may be thought of as a binary number of the form

`Major print level JFDXs`

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).
- `X` x_k, the nonlinear variables involved in the objective function or the constraints.
- `D` π_k, the dual variables for the nonlinear constraints.
- `F` $F(x_k)$, the values of the nonlinear constraint functions.
- `J` $J(x_k)$, the Jacobian matrix.

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

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

`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 `feasible point` means "Ignore the objective function" while finding a feasible point for the linear and nonlinear constraints. It can be used to check that the nonlinear constraints are feasible without altering the call to DNOPT.

`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`

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

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

- If the bounds and linear constraints cannot be satisfied to within t, the problem is declared *infeasible*. Let `sInf` be the corresponding sum of infeasibilities. If `sInf` is quite small, it may be appropriate to raise t by a factor of 10 or 100. Otherwise, some error in the data should be suspected.
- Nonlinear functions will be evaluated only at points that satisfy the bounds and linear constraints. If there are regions where a function is undefined, every attempt should be made to eliminate these regions from the problem.

  For example, if $f(x) = \sqrt{x_1} + \log x_2$, it is essential to place lower bounds on both variables. If $t =$ `1.0e-6`, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious. In general, keep x as far away from singularities as possible.)
- If `Scale option` ≥ 1, feasibility is defined in terms of the *scaled* problem (since it is then more likely to be meaningful).
- In reality, DNOPT 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. DNOPT 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 Factorization statistics generated during the periodic refactorization of the working set (see `Factorization frequency`). Statistics for the *first factorization* each major iteration are controlled by the `Major print level`.

`Pivot tolerance` t Default = $\epsilon^{2/3} \approx$ `3.7e-11`

During solution of QP subproblems, the pivot tolerance is used to prevent constraints from entering the working set if they would cause the working-set matrix to become almost singular.

- When x changes to $x + \alpha p$ for some search direction p, a "ratio test" determines which constraint reaches an upper or lower bound first. The corresponding element of p or $a_i^T p$ 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.
- 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 `dnBEGIN`'s parameter `iPrint`. 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).

`Scale option` i Default = 2 (LP) or 1 (NP)
`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 $\bar{\ell}_j = \ell_j / c(j)$, $\bar{u}_j = u_j / c(j)$, 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''

The default values depend on n_1' and n_1'', the numbers of nonlinear objective variables and Jacobian variables (see p. 11).

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 `dnBEGIN`'s parameter `iSumm`. Set $f = 0$ to suppress the Summary file.

`Suppress parameters`

Normally DNOPT 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 DNOPT not to print the full list.

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

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

The `Total ...` options place an *upper* limit on DNOPT'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 DNOPT to the lower end of `rw` in order to reduce paging activity slightly. (However, DNOPT 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 DNOPT'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 DNOPT's workspace (not counting the first 500 elements). Again, if DNOPT'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.

`System information No` Default
`System information Yes`

The `Yes` option provides additional information on the progress of the iterations.

`Time limit` ℓ Default = 0

This places a limit of ℓ cpu seconds on the time used for solving the problem. The default value $\ell = 0$ 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.)

`Unbounded objective value` $f_{\max}$ Default = `1.0e+15`
`Unbounded step size` $\alpha_{\max}$ Default = `1.0e+18`

These parameters are intended to detect unboundedness in nonlinear problems. (They may not achieve that purpose!) During a 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 `funcon` and 3 calls to `funobj` for `dnOpt`.
1	Individual objective gradients will be checked (with a more reliable test). A key of the form "`OK`" or "`Bad?`" indicates whether or not each component appears to be correct.
2	Individual columns of the problem Jacobian will be checked.
3	Options 2 and 1 will both occur (in that order).
−1	Derivative checking is disabled.

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

`Violation limit` τ Default = 10

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

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

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

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 working set is already specified via the input arrays for DNOPT. This option has the same effect as the input arguments `start = 2`. 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.

7. Output

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

7.1. The PRINT file

If `Print file` > 0, the following information is output to the Print file during the solution process. All printed lines are less than 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.
- A summary of the scaling procedure, if `Scale option` > 0.
- Notes about the initial working set resulting from a CRASH procedure.
- The major iteration log.
- The minor iteration log.
- The EXIT condition and some statistics about the solution obtained.
- The printed solution, if requested.

The last five items are described in the following sections.

7.2. The major iteration log

If `Major print level` > 0, one line of information is output to the Print file every 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 (6.1). 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 (6.2). It is an estimate of the degree of nonoptimality of the reduced costs. Both `Feasbl` and `Optimal` are small in the neighborhood of a solution.
`MeritFunction`	is the value of the augmented Lagrangian merit function (see (2.5)). 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.
`nZ`	The current number of degrees of freedom.
`CondHz`	An estimate of the condition number of R^TR, an estimate of Z^THZ, the reduced Hessian of the Lagrangian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R (which is a lower bound on the condition number of R^TR). `Cond Hz` gives a rough indication of whether or not the optimization procedure is having difficulty. If ϵ is the relative precision of the machine being used, the SQP algorithm will make slow progress if `Cond Hz` becomes as large as $\epsilon^{-1/2} \approx 10^8$, and will probably fail to find a better solution if `Cond Hz` reaches $\epsilon^{-3/4} \approx 10^{12}$. To guard against high values of `Cond Hz`, attention should be given to the scaling of the variables and the constraints. In some cases it may be necessary to add

upper or lower bounds to certain variables to keep them a reasonable distance from singularities in the nonlinear functions or their derivatives.

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

The summary line may include additional code characters that indicate what happened during the course of the 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 DNOPT is not in elastic mode, an "`i`" signifies that the QP subproblem is infeasible. This event triggers the start of nonlinear elastic mode, which remains in effect for all subsequent iterations. Once in elastic mode, the QP subproblems are associated with the elastic problem NP(γ). If DNOPT 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`.
`u`	The QP subproblem was unbounded.
`w`	A weak solution of the QP subproblem was found.

7.3. The minor iteration log

If `Minor print level` > 0, one line of information is output to the Print file every 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 matrix factorization.

The heading contains the items described below.

Label *Description*

`Itn` The current iteration number.

`Bnd` The number of bound constraints in the QP working set.

`Lin` The number of general constraints in the QP working set.

`Art` The number of artificial constraints in the QP working set.

`yMin` The value of the largest nonoptimal Lagrange multiplier.

`jDel` The index of the constraint selected for removal from the working set.

`jAdd` The index of the constraint added to the working set.

`Step` The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. If a constraint is added to the working set during the current iteration, `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.

`pInf,dInf` The number of primal and dual infeasibilities *after* the present iteration. The number of primal infeasibilities (`pInf`) will not increase unless the iterations are in elastic mode. The number of dual infeasibilities (`dInf`) is zero at a solution, but may increase or decrease between iterations.

`SumInf/Objective` If `nInf` > 0, this is `sInf`, the sum of infeasibilities after the present iteration. It usually decreases at each nonzero `Step`, but if `nInf` decreases by 2 or more, `SumInf` may occasionally increase.

If `nInf` $= 0$, then `Objective` is the objective value for the QP subproblem.

In elastic mode, the heading is changed to `nonElastic Inf` in elastic phase 1 and `Elastic Obj` in elastic phase 2. `nonElastic Inf` gives the number infeasible nonelastic constraints. `Elastic Obj` gives the objective function in phase 2 of elastic mode. `Elastic Obj` decreases monotonically.

`Cond T` The ratio of the diagonal elements of T with largest and smallest magnitude. `Cond T` estimates the condition number of the working-set matrix.

`nZ` The current number of degrees of freedom. (The heading is not printed if the problem is linear.)

`norm gZ` The norm of the reduced-gradient vector at the start of the iteration. During Phase 2 this norm will be approximately zero after a unit step.

(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.)

7.4. EXIT conditions

When any solver or auxliliary routine in the DNOPT 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, DNOPT may output

```
DNOPT EXIT 20 -- the problem appears to be unbounded
DNOPT 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 DNOPT 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 DNOPT auxiliary routines)
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). The solution is output to the Print or Solution files if requested.

We describe each exit message from `dnOpt` 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
INFO -- 5 elastic objective minimized
INFO -- 6 elastic infeasibilities minimized
```

These messages usually indicate a successful run. 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, DNOPT 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.

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

Problem name                 Toy
No. of iterations                  16   Objective            1.9001249992E+00
No. of major iterations            13   Linear    obj. term  9.9625002777E-02
Penalty parameter           4.264E+14   Nonlinear obj. term  1.8004999964E+00
No. of calls to funobj             22   No. of calls to funcon             22
Degrees of freedom                  1
No. of degenerate steps             0   Percentage                       0.00
Max x                       2 1.4E+00   Max pi                      1 1.9E+01
Max Primal inf              0 0.0E+00   Max Dual   inf              1 3.8E-08
Nonlinear constraint violn    5.9E-13
```

`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

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

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 DNOPT 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 nonelastic constraints
```

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

Unfortunately, even though DNOPT 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 at a feasible point
INFO -- 22  constraint violation limit reached
```

For linear problems, unboundedness is detected by the simplex method when an active variable or constraint can be increased or decreased by an arbitrary amount without causing a constraint to violate a bound. A message prior to the EXIT message will give the index of the active constraint. 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, DNOPT 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 6.7), 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
```

Some limit was exceeded before the required solution could be found. Check the iteration log to be sure that progress was being made.

```
EXIT -- 40  Terminated after numerical difficulties
INFO -- 41  current point cannot be improved
INFO -- 42  ill-conditioned working set
INFO -- 43  cannot satisfy the working-set constraints
INFO -- 44  Reduced gradient too large
```

For `INFO 41`, DNOPT 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

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

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

For `INFO 42`, the first factorization attempt found the working-set matrix to be numerically singular. (Some diagonals of the triangular matrix T were deemed too small.)

For `INFO 43`, the variables x have been recomputed, given the present the working set. A step of "iterative refinement" has also been applied to increase the accuracy of x, 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 QP iterations, the reduced gradient could not be reduced after several steps of iterative refinement.

In all cases, the problem must be badly scaled (or the working set 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
```

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 DNOPT 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`.

```
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 `mode` 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. DNOPT attempts to evaluate the problem functions closer to a point at which the functions have already been computed.

For `INFO 61` and 62, DNOPT 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. DNOPT 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
```

DNOPT 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`.

```
EXIT -- 90  Input arguments out of range
INFO -- 91  invalid input argument
```

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.

7.5. 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 variables section, a nonlinear constraint section and a linear constraint section then follow, giving one line of information for each variable and constraint.

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

To reduce clutter, a dot "`.`" is printed for any numerical value that is exactly zero. The values ± 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 VARIABLES section

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

Label	*Description*
`Number`	The column number, j. This is the internal number used to refer to x_j in the iteration log.
`Column`	The name of x_j.
`State`	The state of x_j relative to the bounds α and β. The various states possible are as follows.

- `LL` x_j is in the working set at its lower limit, α.
- `UL` x_j is in the working set at its upper limit, β.

`EQ` x_j is in the working set and fixed at the value $\alpha = \beta$.

`FX` x_j is temporarily fixed at some value strictly between its bounds: $\alpha < x_j < \beta$.

`FR` x_j is not in the working set. 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 in the working set, but its Lagrange multiplier 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 other variables and constraints not in the working set *might* change, giving a genuine alternative solution. The values of the dual variables *might* also change.

`D` *Degenerate.* x_j is not in the working set, 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 in the working set. Its dual infeasibility 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.

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

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

`Lagr multiplier` The dual variable is the value $z_j = g_j - \pi^T a_j$, 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).

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

The NONLINEAR ROWS and LINEAR ROWS sections

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

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

and the value of $a^T x$ is called the *constraint value.*

Nonlinear constraints $\alpha \le f_i(x) + a^T x \le \beta$ are treated similarly.

Label *Description*

`Number` The value $n + i$. This is the internal number used to refer to the ith row 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 in the working set at its lower limit, α.

UL The row is in the working set at its upper limit, β.

EQ The row is in the working set with $\alpha = \beta$.

FR The constraint is not in the working set.

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 constraint is active, but its Lagrange multipliers is essentially zero. This means that if the constraint value were allowed to start moving from its current value, there would be no change in the objective function. The values of the dual variables *might* also change.

D *Degenerate.* The constraint is free, but it is equal to (or very close to) one of its bounds.

I *Infeasible.* The constraint is free and is currently violating one of its bounds by more than the Feasibility tolerance.

N *Not precisely optimal.* The constraint is active. Its Lagrange multiplier 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).

Lower limit α, the lower bound on the row.

Upper limit β, the upper bound on the row.

Dual variable The value of the Lagrange multiplier π_i, often called the shadow price (or simplex multiplier) for the ith constraint.

Slack The amount by which the constraint value differs from its nearest bound. (For free rows, it is taken to be minus the Value.)

The following SOLUTION file is from the example of Section 3.2, using Solution Yes.

```
        ==============================
        D N O P T  2.2-1    (May 2016)
        ==============================

SPECS file
----------

     Begin  Options for dntoy  (example program for dnopt)
        Major Iterations Limit           50
        Major Print Level                 1 * One line or more every major
        Minor Print Level                 0 * One line         every minor s
        Print   Frequency                 1 * Print every itn
        Summary Frequency                 1 * Print every itn
        Solution                        Yes
        Elastic weight                  100.0
     End    Options for dntoy

DNSPEC   EXIT 100 -- finished successfully
DNSPEC   INFO 101 -- SPECS file read

Parameters
==========

Files
-----
Standard input.........         5       Solution file..........         0
(Printer)..............         9
(Specs file)...........         4
Standard output........         6

Frequencies
-----------
Print frequency........         1       Check frequency........        60       Expand frequency.......     10000
```

```
Summary frequency......          1      Factorization frequency         50      Hessian frequency......         30

QP subproblems
--------------
Scale tolerance........      0.000      Minor feasibility tol..   1.00E-06      Iteration limit........       1000
Scale option...........          0      Minor optimality tol...   1.00E-06      Minor print level......          0
Crash tolerance........      0.100      Pivot tolerance........   3.25E-11
Crash option...........          3

The SQP Method
--------------
Minimize...............                 Cold start.............                 Proximal point method..          1
Nonlinear objectiv vars          3      Major optimality tol...   2.00E-06      Function precision.....   3.00E-13
Unbounded step size....   1.00E+20      Max degrees of freedom.          4      Difference interval....   5.48E-07
Unbounded objective....   1.00E+10      Elastic weight.........   1.00E+02      Central difference int.   6.70E-05
Major step limit.......   2.00E+00      Derivative linesearch..                 Derivative level.......          3
Major iterations limit.         50      Linesearch tolerance...    0.90000      Verify level...........          0
Minor iterations limit.        100      Penalty parameter......   0.00E+00      Major Print Level......          1
Time limit (secs)...... 9999999.0

Hessian Approximation
---------------------
Hessian quasi-Newton...                 Hessian updates........         30

Nonlinear constraints
---------------------
Nonlinear constraints..          2      Major feasibility tol..   1.00E-06      Violation limit........   1.00E+06
Nonlinear Jacobian vars          2

Miscellaneous
-------------
Debug level............          0      Timing level...........          3
eps (machine precision)   2.22E-16      System information.....         No      Sticky parameters......         No
```

```
Total char*8  workspace       500       Total integer workspace      4000       Total real    workspace      3000
Total char*8  (minimum)       500       Total integer (minimum)       532       Total real    (minimum)       754

Matrix statistics
-----------------
              Total      Normal        Free       Fixed     Bounded
Rows              4           1           1           2           0
Columns           4           2           2           0           0
Biggest  constant element        4.0000E+00  (excluding fixed columns,
Smallest constant element        0.0000E+00   free rows, and RHS)

No. of objective coefficients             1
Biggest                          3.0000E+00  (excluding fixed columns)
Smallest                         3.0000E+00

Nonlinear constraints       2     Linear constraints       2
Nonlinear variables         3     Linear variables         1
Jacobian  variables         2     Objective variables      3
Total constraints           4     Total variables          4

The user has defined       4   out of       4   constraint gradients.
The user has defined       3   out of       3   objective  gradients.

Cheap test of user-supplied problem derivatives...

The constraint gradients seem to be OK.

-->  The largest discrepancy was    1.22E-07  in constraint     1

The objective  gradients seem to be OK.

Gradient projected in one direction   0.00000000000E+00
```

```
Difference approximation             6.08590116169E-08

  Itns Major minors    Step   nCon Feasible  Optimal  MeritFunction     nZ  condHz Penalty
     2     0      2              1  4.0E+00  4.0E-01  0.0000000E+00      1 1.0E+00         _  r
     3     1      1 3.3E-01      2  2.7E+00  3.1E-01  2.8666667E+01      1 1.0E+00 3.5E+00 _  rl
     4     2      1 4.3E-01      3  7.5E-01  6.7E-01  2.9315655E+01      1 1.0E+00 3.9E+00 _s  l
     7     3      3 1.0E+00      5  5.9E-02  1.2E+00  2.6836417E+01      1 1.0E+00 3.9E+00 _ m
     8     4      1 4.8E-01     10  5.0E-01  3.4E+00  1.8114971E+01      1 1.0E+00 3.9E+00 _ m
     9     5      1 1.0E+00     11  6.3E-03  2.8E+00  1.0001147E+01                3.9E+00 _
    10     6      1 1.0E+00     13  8.2E-01  9.0E-01  7.8937505E-01      1 1.0E+00 3.9E+00 _ m
    11     7      1 1.0E+00     15  1.3E-01  1.0E+00  4.1685822E+00                2.1E+01 _ M
    11     8      0 1.0E+00     16  4.1E-02  3.5E-01  2.0738198E+00                5.3E+00 _
    12     9      1 1.0E+00     17  8.5E-03  7.2E-02  1.9074941E+00      1 1.0E+00 5.6E+02 _
    13    10      1 1.0E+00     18  6.2E-04  1.0E-02  1.9021149E+00      1 1.0E+00 9.3E+03 _
    14    11      1 1.0E+00     19  9.4E-06  4.0E-04  1.9001641E+00      1 1.0E+00 8.7E+05 _
    15    12      1 1.0E+00     20 (1.4E-08) 2.6E-06  1.9001252E+00      1 1.0E+00 1.8E+09 _
    16    13      1 1.0E+00     21 (5.9E-13)(3.2E-09) 1.9001250E+00      1 1.0E+00 4.3E+14 _

DNOPT    EXIT   0 -- finished successfully
DNOPT    INFO   1 -- optimality conditions satisfied

Problem name                 Toy
No. of iterations                  16   Objective            1.9001249992E+00
No. of major iterations            13   Linear    obj. term  9.9625002777E-02
Penalty parameter           4.264E+14   Nonlinear obj. term  1.8004999964E+00
No. of calls to funobj             22   No. of calls to funcon             22
Degrees of freedom                  1
No. of degenerate steps             0   Percentage                       0.00
Max x                       2 1.4E+00   Max pi                      1 1.9E+01
Max Primal inf              0 0.0E+00   Max Dual   inf              1 3.8E-08
Nonlinear constraint violn    5.9E-13

Section 1 - Variables
```

```
Number .Column. State      Value       Lower bound     Upper bound   Lagr multiplier       Slack
     1  x1         FR  -0.7062201E-01      None            None
     2  x2         FR    1.412449          None            None
     3  x3         LL    0.000000           .              None          24.68378            .
     4  x4         FR   0.1992500E-01       .              None              .             0.1993E-01

Section 2 - Nonlinear Rows

Number ...Row.. State      Value       Lower bound     Upper bound   Lagr multiplier       Slack
     5  nc1        EQ    2.000000        2.000000        2.000000      -19.00012         -0.5884E-12
     6  nc2        EQ    4.000000        4.000000        4.000000       5.000000         -0.2132E-13

Section 3 - Linear Rows

Number ...Row.. State      Value       Lower bound     Upper bound   Lagr multiplier       Slack
     7  lc1        FR    5.508552           .              None              .              5.509
     8  lc2        FR   0.9962500E-01      None            None
```

7.6. The SOLUTION file

The information in a printed solution (Section 7.5) 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 by a self-contained program that extracts and saves certain values as required for possible further computation. Typically the first 14 records would be ignored. Each subsequent record may be read using

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

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

7.7. 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`.

```
==============================
D N O P T  2.2-1    (May 2016)
==============================

Begin  Options for dntoy  (example program for dnopt)

DNSPEC   EXIT 100 -- finished successfully
DNSPEC   INFO 101 -- SPECS file read



Nonlinear constraints       2     Linear constraints       2
Nonlinear variables         3     Linear variables         1
Jacobian  variables         2     Objective variables      3
Total constraints           4     Total variables          4



Starting  dntoy
The user has defined       4   out of       4   constraint gradients.
The user has defined       3   out of       3   objective  gradients.

Major minors     Step   nCon Feasible  Optimal  MeritFunction    nZ Penalty
    0      2       -       1  4.0E+00  4.0E-01  0.0000000E+00     1           r
    1      1   3.3E-01     2  2.7E+00  3.1E-01  2.8666667E+01     1 3.5E+00   rl
    2      1   4.3E-01     3  7.5E-01  6.7E-01  2.9315655E+01     1 3.9E+00 s  l
    3      3   1.0E+00     5  5.9E-02  1.2E+00  2.6836417E+01     1 3.9E+00  m
    4      1   4.8E-01    10  5.0E-01  3.4E+00  1.8114971E+01     1 3.9E+00  m
```

```
    5      1   1.0E+00    11  6.3E-03  2.8E+00  1.0001147E+01       3.9E+00
    6      1   1.0E+00    13  8.2E-01  9.0E-01  7.8937505E-01     1 3.9E+00  m
    7      1   1.0E+00    15  1.3E-01  1.0E+00  4.1685822E+00       2.1E+01  M
    8      0   1.0E+00    16  4.1E-02  3.5E-01  2.0738198E+00       5.3E+00
    9      1   1.0E+00    17  8.5E-03  7.2E-02  1.9074941E+00     1 5.6E+02

 Major minors     Step   nCon Feasible  Optimal  MeritFunction    nZ Penalty
   10      1   1.0E+00    18  6.2E-04  1.0E-02  1.9021149E+00     1 9.3E+03
   11      1   1.0E+00    19  9.4E-06  4.0E-04  1.9001641E+00     1 8.7E+05
   12      1   1.0E+00    20 (1.4E-08) 2.6E-06  1.9001252E+00     1 1.8E+09
   13      1   1.0E+00    21 (5.9E-13)(3.2E-09) 1.9001250E+00     1 4.3E+14

 DNOPT    EXIT   0 -- finished successfully
 DNOPT    INFO   1 -- optimality conditions satisfied

 Problem name                 Toy
 No. of iterations                  16   Objective            1.9001249992E+00
 No. of major iterations            13   Linear    obj. term  9.9625002777E-02
 Penalty parameter           4.264E+14   Nonlinear obj. term  1.8004999964E+00
 No. of calls to funobj             22   No. of calls to funcon             22
 Degrees of freedom                  1
 No. of degenerate steps             0   Percentage                       0.00
 Max x                       2 1.4E+00   Max pi                      1 1.9E+01
 Max Primal inf              0 0.0E+00   Max Dual   inf              1 3.8E-08
 Nonlinear constraint violn    5.9E-13

 Solution printed on file   9

 Finishing dntoy
```

Acknowledgements

We are grateful to Mark Milam and Mike Messina of Northrop Grumman Aerospace Systems for their helpful comments during the development of the DNOPT package. We would also like to thank Mark Milam for his invaluable support of the DNOPT project.

References

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

[2] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963. 4

[3] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, *A Fortran-to-C converter*, Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ, 1990. 4

[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. 10

[5] R. Fourer, *Solving staircase linear programs by the simplex method. 1: Inversion*, Math. Program., 23 (1982), pp. 274–313. 58

[6] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM J. Optim., 12 (2002), pp. 979–1006. 4

[7] ———, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM Rev., 47 (2005), pp. 99–131. 4

[8] ———, *User's guide for SNOPT Version 7: Software for large-scale nonlinear programming*, Numerical Analysis Report 06-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2006. 4

[9] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *Procedures for optimization problems with a mixture of bounds and general linear constraints*, ACM Trans. Math. Software, 10 (1984), pp. 282–298. 9

[10] ———, *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. 36

[11] ———, *A practical anti-cycling procedure for linearly constrained optimization*, Math. Program., 45 (1989), pp. 437–474. 52

[12] ———, *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. 10

[13] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1981. 41

[14] P. E. Gill, M. A. Saunders, and E. Wong, *An SQP method for medium-scale nonlinear programming*, Center for Computational Mathematics Report CCoM 16-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2016. 6

[15] P. E. Gill and E. Wong, *Sequential quadratic programming methods*, in Mixed Integer Nonlinear Programming, J. Lee and S. Leyffer, eds., vol. 154 of The IMA Volumes in Mathematics and its Applications, Springer New York, 2012, pp. 147–224. 7