NAG Library Function Document

nag_glopt_nlp_multistart_sqp (e05ucc)

1 Purpose

nag_glopt_nlp_multistart_sqp (e05ucc) is designed to find the global minimum of an arbitrary smooth function subject to constraints (which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints) by generating a number of different starting points and performing a local search from each using sequential quadratic programming.

2 Specification

```
#include <nag.h>
#include <nage05.h>
void nag_glopt_nlp_multistart_sqp (Integer n, Integer nclin, Integer ncnln,
     const double a[], Integer tda, const double bl[], const double bu[],
     void  (*confun)(Integer *mode, Integer ncnln, Integer n,
           Integer tdcjsl, const Integer needc[], const double x[],
           double c[], double cjsl[], Integer nstate, Nag_Comm *comm),
     void  (*objfun)(Integer *mode, Integer n, const double x[],
           double *objf, double objgrd[], Integer nstate, Nag_Comm *comm),
     Integer npts, double x[], Integer ldx,
     void  (*start)(Integer npts, double quas[], Integer n,
           Nag_Boolean repeat, const double bl[], const double bu[],
           Nag_Comm *comm, Integer *mode),
     Nag_Boolean repeat, Integer nb, double objf[], double objgrd[],
     Integer ldobjgrd, Integer iter[], double c[], Integer ldc,
     double cjac[], Integer ldcjac, Integer sdcjac, double r[], Integer ldr,
     Integer sdr, double clamda[], Integer ldclamda, Integer istate[],
     Integer ldistate, Integer iopts[], double opts[], Nag_Comm *comm,
     Integer info[], NagError *fail)
```

Before calling nag_glopt_nlp_multistart_sqp (e05ucc), the optional argument arrays **iopts** and **opts** MUST be initialized for use with nag_glopt_nlp_multistart_sqp (e05ucc) by calling nag_glopt_opt_set (e05zkc) with **optstr** set to '`Initialize = e05ucc`'. Optional arguments may be specified by calling nag_glopt_opt_set (e05zkc) before the call to nag_glopt_nlp_multistart_sqp (e05ucc).

The declared lenths of **iopts** and **opts** must be at least 740 and 485 respectively.

3 Description

The problem is assumed to be stated in the following form:

$$\underset{x \in R^n}{\text{minimize}}\ F(x) \quad \text{subject to} \quad l \leq \begin{pmatrix} x \\ A_L x \\ c(x) \end{pmatrix} \leq u, \tag{1}$$

where $F(x)$ (the *objective function*) is a nonlinear function, A_L is an n_L by n linear constraint matrix, and $c(x)$ is an n_N element vector of nonlinear constraint functions. (The matrix A_L and the vector $c(x)$ may be empty.) The objective function and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (This function will usually solve (1) if there are only isolated discontinuities away from the solution.)

nag_glopt_nlp_multistart_sqp (e05ucc) solves a user-specified number of local optimization problems with different starting points. You may specify the starting points via the function **start**. If a random number generator is used to generate the starting points then the argument **repeat** allows you to specify

whether a repeatable set of points are generated or whether different starting points are generated on different calls. The resulting local minima are ordered and the best **nb** results returned in order of ascending values of the resulting objective function values at the minima. Thus the value returned in position 1 will be the best result obtained. If a sufficient number of different points are chosen then this is likely to be be the global minimum. Please note that the default version of **start** uses a random number generator to generate the starting points.

4 References

Dennis J E Jr and Moré J J (1977) Quasi-Newton methods, motivation and theory *SIAM Rev.* **19** 46–89

Dennis J E Jr and Schnabel R B (1981) A new derivation of symmetric positive-definite secant updates *nonlinear programming* (eds O L Mangasarian, R R Meyer and S M Robinson) **4** 167–199 Academic Press

Dennis J E Jr and Schnabel R B (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* Prentice–Hall

Fletcher R (1987) *Practical Methods of Optimization* (2nd Edition) Wiley

Gill P E, Hammarling S, Murray W, Saunders M A and Wright M H (1986) Users' guide for LSSOL (Version 1.0) *Report SOL 86-1* Department of Operations Research, Stanford University

Gill P E, Murray W, Saunders M A and Wright M H (1984) Users' guide for SOL/QPSOL version 3.2 *Report SOL* 84–5 Department of Operations Research, Stanford University

Gill P E, Murray W, Saunders M A and Wright M H (1986a) Some theoretical properties of an augmented Lagrangian merit function *Report SOL* 86–6R Department of Operations Research, Stanford University

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

Gill P E, Murray W and Wright M H (1981) *Practical Optimization* Academic Press

Powell M J D (1974) Introduction to constrained optimization *Numerical Methods for Constrained Optimization* (eds P E Gill and W Murray) 1–28 Academic Press

Powell M J D (1983) Variable metric methods in constrained optimization *Mathematical Programming: the State of the Art* (eds A Bachem, M Grötschel and B Korte) 288–311 Springer–Verlag

5 Arguments

1: **n** – Integer *Input*

On entry: n, the number of variables.

Constraint: $\mathbf{n} > 0$.

2: **nclin** – Integer *Input*

On entry: n_L, the number of general linear constraints.

Constraint: $\mathbf{nclin} \geq 0$.

3: **ncnln** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

Constraint: $\mathbf{ncnln} \geq 0$.

4: **a**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **a** must be at least $\mathbf{nclin} \times \mathbf{tda}$.

On entry: the matrix A_L of general linear constraints in (1). That is, $\mathbf{A}[(i-1) \times \mathbf{tda} + j - 1]$ must contain the jth coefficient of the ith general linear constraint, for $j = 1, 2, \ldots, \mathbf{n}$ and $i = 1, 2, \ldots, \mathbf{nclin}$. If $\mathbf{nclin} = 0$ then **a** may be specified as **NULL**.

5: **tda** – Integer *Input*

On entry: the stride separating matrix column elements in the array **a**.

Constraints:

if $\mathbf{nclin} > 0$, $\mathbf{tda} \geq \mathbf{n}$.

6: **bl**[**n** + **nclin** + **ncnln**] – const double *Input*

7: **bu**[**n** + **nclin** + **ncnln**] – const double *Input*

On entry: **bl** must contain the lower bounds and **bu** the upper bounds for all the constraints in the following order. The first n elements of each array must contain the bounds on the variables, the next n_L elements the bounds for the general linear constraints (if any) and the next n_N elements the bounds for the general nonlinear constraints (if any). To specify a nonexistent lower bound (i.e., $l_j = -\infty$), set $\mathbf{bl}[j-1] \leq -bigbnd$, and to specify a nonexistent upper bound (i.e., $u_j = +\infty$), set $\mathbf{bu}[j-1] \geq bigbnd$; the default value of $bigbnd$ is 10^{20}, but this may be changed by the optional argument **Infinite Bound Size**. To specify the jth constraint as an equality, set $\mathbf{bl}[j-1] = \mathbf{bu}[j-1] = \beta$, say, where $|\beta| < bigbnd$.

Constraints:

$\mathbf{bl}[j-1] \leq \mathbf{bu}[j-1]$, for $j = 1, 2, \ldots, \mathbf{n} + \mathbf{nclin} + \mathbf{ncnln}$;
if $\mathbf{bl}[j-1] = \mathbf{bu}[j-1] = \beta$, $|\beta| < bigbnd$.

8: **confun** – function, supplied by the user *External Function*

confun must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \dfrac{\partial c}{\partial x}$) for a specified n-element vector x. If there are no nonlinear constraints (i.e., $\mathbf{ncnln} = 0$), **confun** will never be called by nag_glopt_nlp_multistart_sqp (e05ucc) and the NAG defined null void function pointer, **NULLFN**, may be supplied in the call instead. If there are nonlinear constraints, the first call to **confun** will occur before the first call to **objfun**.

The specification of **confun** is:

```
void  confun (Integer *mode, Integer ncnln, Integer n, Integer tdcjsl,
      const Integer needc[], const double x[], double c[],
      double cjsl[], Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: indicates which values must be assigned during each call of **confun**. Only the following values need be assigned, for each value of i such that $\mathbf{needc}[i-1] > 0$:

mode $= 0$
$\mathbf{c}[i-1]$.

mode $= 1$
All available elements in $\mathbf{CJSL}(i, j)$, for $j = 1, 2, \ldots, \mathbf{n}$ (see **cjsl** for the definition of **CJSL**).

mode $= 2$
$\mathbf{c}[i-1]$ and all available elements in $\mathbf{CJSL}(i, j)$, for $j = 1, 2, \ldots, \mathbf{n}$ (see **cjsl** for the definition of **CJSL**).

On exit: may be set to a negative value if you wish to abandon the solution to the current local minimization problem. In this case nag_glopt_nlp_multistart_sqp (e05ucc) will move to the next local minimization problem.

2: **ncnln** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

3: **n** – Integer *Input*

On entry: n, the number of variables.

4: **tdcjsl** – Integer *Input*

On entry: the stride separating matrix column elements in the array **cjsl**.

5: **needc**[**ncnln**] – const Integer *Input*

On entry: the indices of the elements of **c** and/or **cjsl** that must be evaluated by **confun**. If **needc**$[i-1] > 0$, **c**$[i-1]$ and/or the available elements of **CJSL**(i,j), for $j = 1, 2, \ldots,$ **n** (see argument **mode**) must be evaluated at x. See **cjsl** for the definition of **CJSL**.

6: **x**[**n**] – const double *Input*

On entry: x, the vector of variables at which the constraint functions and/or the available elements of the constraint Jacobian are to be evaluated.

7: **c**[**ncnln**] – double *Output*

On exit: if **needc**$[k-1] > 0$ and **mode** $= 0$ or 2, **c**$[k-1]$ must contain the value of $c_k(x)$. The remaining elements of **c**, corresponding to the non-positive elements of **needc**, need not be set.

8: **cjsl**[**ncnln** $\times$ **tdcjsl**] – double *Input/Output*

Note: where **CJSL**(k,j) appears in this document, it refers to the array element **cjsl**$[(k-1) \times$ **n** $+ j - 1]$.

cjsl may be regarded as a two-dimensional 'slice' in row order of the three-dimensional matrix **CJAC** stored in the array **cjac** of nag_glopt_nlp_multistart_sqp (e05ucc).

On entry: unless **Derivative Level** $= 2$ or 3, the elements of **cjsl** are set to special values which enable nag_glopt_nlp_multistart_sqp (e05ucc) to detect whether they are changed by **confun**.

On exit: if **needc**$[k-1] > 0$ and **mode** $= 1$ or 2, **CJSL**(k,j), for $j = 1, 2, \ldots,$ **n**, must contain the available elements of the vector ∇c_k given by

$$\nabla c_k = \left(\frac{\partial c_k}{\partial x_1}, \frac{\partial c_k}{\partial x_2}, \ldots, \frac{\partial c_k}{\partial x_n}\right)^{\mathrm{T}},$$

where $\dfrac{\partial c_k}{\partial x_j}$ is the partial derivative of the kth constraint with respect to the jth variable, evaluated at the point x. See also the argument **nstate**. The remaining **CJSL**(k,j), for $j = 1, 2, \ldots,$ **n**, corresponding to non-positive elements of **needc**, need not be set.

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

Note that constant nonzero elements do affect the values of the constraints. Thus, if **CJSL**(k,j) is set to a constant value, it need not be reset in subsequent calls to **confun**,

but the value $\mathbf{CJSL}(k,j) \times \mathbf{x}[j-1]$ must nonetheless be added to $\mathbf{c}[k-1]$. For example, if $\mathbf{CJSL}(1,1) = 2$ and $\mathbf{CJSL}(1,2) = -5$ then the term $2 \times \mathbf{x}[0] - 5 \times \mathbf{x}[1]$ must be included in the definition of $\mathbf{c}[0]$.

It must be emphasized that, if **Derivative Level** $= 0$ or 1, unassigned elements of **cjsl** are not treated as constant; they are estimated by finite differences, at nontrivial expense. If you do not supply a value for the optional argument **Difference Interval**, an interval for each element of x is computed automatically at the start of each local optimization. The automatic procedure can usually identify constant elements of **cjsl**, which are then computed once only by finite differences.

9: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$ then nag_glopt_nlp_multistart_sqp (e05ucc) is calling **confun** for the first time on the current local optimization problem. This argument setting allows you to save computation time if certain data must be calculated only once.

10: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type Nag_Comm; the following members are relevant to **confun**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_glopt_nlp_multistart_sqp (e05ucc) you may allocate memory and initialize these pointers with various quantities for use by **confun** when called from nag_glopt_nlp_multistart_sqp (e05ucc) (see Section 3.2.1.1 in the Essential Introduction).

confun should be tested separately before being used in conjunction with nag_glopt_nlp_multistart_sqp (e05ucc). See also the description of the optional argument **Verify**.

9: **objfun** – function, supplied by the user *External Function*

objfun must calculate the objective function $F(x)$ and (optionally) its gradient $g(x) = \dfrac{\partial F}{\partial x}$ for a specified n-vector x.

The specification of **objfun** is:

```
void  objfun (Integer *mode, Integer n, const double x[],
      double *objf, double objgrd[], Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: indicates which values must be assigned during each call of **objfun**. Only the following values need be assigned:

mode $= 0$
objf.

mode $= 1$
All available elements of **objgrd**.

mode $= 2$
objf and all available elements of **objgrd**.

On exit: may be set to a negative value if you wish to abandon the solution to the current local minimization problem. In this case nag_glopt_nlp_multistart_sqp (e05ucc) will move to the next local minimization problem.

2: **n** – Integer *Input*

On entry: n, the number of variables.

3: **x**[**n**] – const double *Input*

On entry: x, the vector of variables at which the objective function and/or all available elements of its gradient are to be evaluated.

4: **objf** – double * *Output*

On exit: if **mode** $= 0$ or 2, **objf** must be set to the value of the objective function at x.

5: **objgrd**[**n**] – double *Input/Output*

On entry: the elements of **objgrd** are set to special values which enable nag_glopt_nlp_multistart_sqp (e05ucc) to detect whether they are changed by **objfun**.

On exit: if **mode** $= 1$ or 2, **objgrd** must return the available elements of the gradient evaluated at x.

6: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$ then nag_glopt_nlp_multistart_sqp (e05ucc) is calling **objfun** for the first time on the current local optimization problem. This argument setting allows you to save computation time if certain data must be calculated only once.

7: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type Nag_Comm; the following members are relevant to **objfun**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_glopt_nlp_multistart_sqp (e05ucc) you may allocate memory and initialize these pointers with various quantities for use by **objfun** when called from nag_glopt_nlp_multistart_sqp (e05ucc) (see Section 3.2.1.1 in the Essential Introduction).

objfun should be tested separately before being used in conjunction with nag_glopt_nlp_multistart_sqp (e05ucc). See also the description of the optional argument **Verify**.

10: **npts** – Integer *Input*

On entry: the number of different starting points to be generated and used. The more points used, the more likely that the best returned solution will be a global minimum.

Constraint: $1 \leq \mathbf{nb} \leq \mathbf{npts}$.

11: **x**[**ldx** $\times$ **nb**] – double *Output*

Note: where $\mathbf{X}(j,i)$ appears in this document, it refers to the array element $\mathbf{x}[(i-1) \times \mathbf{ldx} + j - 1]$.

On exit: $\mathbf{X}(j,i)$ contains the final estimate of the ith solution, for $j = 1, 2, \ldots, \mathbf{n}$.

12: **ldx** – Integer *Input*

On entry: the first dimension of **X** as stored in the array **x**.

Constraint: **ldx** $\geq$ **n**.

13: **start** – function, supplied by the user *External Function*

start must calculate the **npts** starting points to be used by the local optimizer. If you do not wish to write a function specific to your problem then you can specify the NAG defined null void function pointer, **NULLFN**, in the call. In this case, a default function uses the NAG quasi-random number generators to distribute starting points uniformly across the domain. It is affected by the value of **repeat**.

The specification of **start** is:

```
void  start (Integer npts, double quas[], Integer n,
      Nag_Boolean repeat, const double bl[], const double bu[],
      Nag_Comm *comm, Integer *mode)
```

1: **npts** – Integer *Input*

On entry: indicates the number of starting points.

2: **quas**[**n** × **npts**] – double *Input/Output*

On entry: all elements of **quas** will have been set to zero, so only nonzero values need be set subsequently.

On exit: must contain the starting points for the **npts** local minimizations, i.e., **quas**$[(j-1) \times$ **npts** $+ i - 1]$ must contain the jth component of the ith starting point.

3: **n** – Integer *Input*

On entry: the number of variables.

4: **repeat** – Nag_Boolean *Input*

On entry: specifies whether a repeatable or non-repeatable sequence of points are to be generated.

5: **bl**[**n**] – const double *Input*

On entry: the lower bounds on the variables. These may be used to ensure that the starting points generated in some sense 'cover' the region, but there is no requirement that a starting point be feasible.

6: **bu**[**n**] – const double *Input*

On entry: the upper bounds on the variables. (See **bl**.)

7: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type Nag_Comm; the following members are relevant to **start**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_glopt_nlp_multistart_sqp (e05ucc) you may allocate memory and initialize these pointers with various quantities for use by **start** when called from nag_glopt_nlp_multistart_sqp (e05ucc) (see Section 3.2.1.1 in the Essential Introduction).

8: **mode** – Integer * *Input/Output*

On entry: **mode** will contain 0.

On exit: if you set **mode** to a negative value then nag_glopt_nlp_multistart_sqp (e05ucc) will terminate immediately with **fail**.**code** = NE_USER_STOP. Provided **fail** is not

> NAGERR_DEFAULT on entry to nag_glopt_nlp_multistart_sqp (e05ucc), **fail**.**errnum** will contain this value of **mode**.

14: **repeat** – Nag_Boolean *Input*

On entry: is passed as an argument to **start** and may be used to initialize a random number generator to a repeatable, or non-repeatable, sequence.

15: **nb** – Integer *Input*

On entry: the number of solutions to be returned. The function saves up to **nb** local minima ordered by increasing value of the final objective function. If the defining criterion for 'best solution' is only that the value of the objective function is as small as possible then **nb** should be set to 1. However, if you want to look at other solutions that may have desirable properties then setting $\mathbf{nb} > 1$ will produce **nb** local minima, ordered by increasing value of their objective functions at the minima.

Constraint: $1 \leq \mathbf{nb} \leq \mathbf{npts}$.

16: **objf**[**nb**] – double *Output*

On exit: **objf**$[i-1]$ contains the value of the objective function at the final iterate for the ith solution.

17: **objgrd**[**ldobjgrd** $\times$ **nb**] – double *Output*

Note: where **OBJGRD**(j,i) appears in this document, it refers to the array element **objgrd**$[(i-1) \times \mathbf{ldobjgrd} + j - 1]$.

On exit: **OBJGRD**(j,i) contains the gradient of the objective function for the ith solution at the final iterate (or its finite difference approximation), for $j = 1, 2, \ldots, \mathbf{n}$.

18: **ldobjgrd** – Integer *Input*

On entry: the first dimension of **OBJGRD** as stored in the array **objgrd**.

Constraint: **ldobjgrd** $\geq$ **n**.

19: **iter**[**nb**] – Integer *Output*

On exit: **iter**$[i-1]$ contains the number of major iterations performed to obtain the ith solution. If less than **nb** solutions are returned then **iter**$[\mathbf{nb}-1]$ contains the number of starting points that have resulted in a converged solution. If this is close to **npts** then this might be indicative that fewer than **nb** local minima exist.

20: **c**[**ldc** $\times$ **nb**] – double *Output*

Note: where **C**(j,i) appears in this document, it refers to the array element **c**$[(i-1) \times \mathbf{ldc} + j - 1]$.

On exit: if **ncnln** > 0, **C**(j,i) contains the value of the jth nonlinear constraint function c_j at the final iterate, for the ith solution, for $j = 1, 2, \ldots, \mathbf{ncnln}$.

If **ncnln** $= 0$, the array **c** is not referenced and may be specified as **NULL**.

21: **ldc** – Integer *Input*

On entry: the first dimension of **C** as stored in the array **c**.

Constraint: **ldc** $\geq$ **ncnln**.

22: **cjac**$[dim]$ – double *Output*

Note: the dimension, *dim*, of the array **cjac** must be at least **ldcjac** $\times$ **sdcjac** $\times$ **nb**.

Where **CJAC**(k, j, i) appears in this document, it refers to the array element **cjac**$[(i-1) \times$ **ldcjac** $\times$ **sdcjac** $+ (j-1) \times$ **ldcjac** $+ k - 1]$.

On exit: if **ncnln** > 0, **cjac** contains the Jacobian matrices of the nonlinear constraint functions at the final iterate for each of the returned solutions, i.e., **CJAC**(k, j, i) contains the partial derivative of the kth constraint function with respect to the jth variable, for $k = 1, 2, \ldots,$ **ncnln** and $j = 1, 2, \ldots,$ **n**, for the ith solution. (See the discussion of argument **cjsl** under **confun**.)

If **ncnln** $= 0$, the array **cjac** is not referenced and may be specified as **NULL**.

23: **ldcjac** – Integer *Input*

On entry: the first dimension of the matrix **CJAC** as stored in the array **cjac**.

Constraint: **ldcjac** $\geq$ **ncnln**.

24: **sdcjac** – Integer *Input*

On entry: the second dimension of the matrix **CJAC** as stored in the array **cjac**.

Constraint: if **ncnln** > 0, **sdcjac** $\geq$ **n**.

25: **r**$[dim]$ – double *Output*

Note: the dimension, *dim*, of the array **r** must be at least **ldr** $\times$ **sdr** $\times$ **nb**.

The element **R**(i, j, k) is stored in the array element **r**$[(k-1) \times$ **ldr** $\times$ **sdr** $+ (j-1) \times$ **ldr** $+ i - 1]$.

On exit: for each of the **nb** solutions **r** will contain a form of the Hessian; for the ith returned solution **R**$($**ldr**, **sdr**$, i)$ contains the Hessian that would be returned from the local minimizer. If **Hessian** $=$ 'NO', the default, each **R**$($**ldr**, **sdr**$, i)$ contains the upper triangular Cholesky factor R of $Q^{\mathrm{T}}HQ$, an estimate of the transformed and reordered Hessian of the Lagrangian at x. If **Hessian** $=$ 'YES', **R**$($**ldr**, **sdr**$, i)$ contains the upper triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

26: **ldr** – Integer *Input*

On entry: the first dimension of the matrix **R** as stored in the array **r**.

Constraint: **ldr** $\geq$ **n**.

27: **sdr** – Integer *Input*

On entry: the second dimension of the matrix **R** as stored in the array **r**.

Constraint: **sdr** $\geq$ **n**.

28: **clamda**[**ldclamda** $\times$ **nb**] – double *Output*

Note: where **CLAMDA**(j, i) appears in this document, it refers to the array element **clamda**$[(i-1) \times$ **ldclamda** $+ j - 1]$.

On exit: the values of the QP multipliers from the last QP subproblem solved for the ith solution. **CLAMDA**(j, i) should be non-negative if **ISTATE**$(j, i) = 1$ and non-positive if **ISTATE**$(j, i) = 2$.

29: **ldclamda** – Integer *Input*

On entry: the first dimension of **CLAMDA** as stored in the array **clamda**.

Constraint: **ldclamda** $\geq$ **n** $+$ **nclin** $+$ **ncnln**.

30: **istate**[**ldistate** $\times$ **nb**] – Integer *Output*

Note: where **ISTATE**(j, i) appears in this document, it refers to the array element **istate**$[(i-1) \times$ **ldistate** $+ j - 1]$.

On exit: **ISTATE**(j,i) contains the status of the constraints in the QP working set for the ith solution. The significance of each possible value of **ISTATE**(j,i) is as follows:

ISTATE(j,i)	**Meaning**
0	The constraint is satisfied to within the feasibility tolerance, but is not in the QP working set.
1	This inequality constraint is included in the QP working set at its lower bound.
2	This inequality constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This value of **istate** can occur only when **bl**$[j-1] =$ **bu**$[j-1]$.

31: **ldistate** – Integer *Input*

On entry: the first dimension of **ISTATE** as stored in the array **istate**.

Constraint: **ldistate** $\geq$ **n** + **nclin** + **ncnln**.

32: **iopts**[**740**] – Integer *Communication Array*

33: **opts**[**485**] – double *Communication Array*

The arrays **iopts** and **opts** MUST NOT be altered between calls to any of the functions nag_glopt_nlp_multistart_sqp (e05ucc) and nag_glopt_opt_set (e05zkc).

34: **comm** – Nag_Comm * *Communication Structure*

The NAG communication argument (see Section 3.2.1.1 in the Essential Introduction).

35: **info**[**nb**] – Integer *Output*

On exit: **info**$[i-1]$ contains one of 0, 1 or 6.

info$[i-1] = 1$
The final iterate x satisfies the first-order Kuhn–Tucker conditions (see Section 11.1) to the accuracy requested, but the sequence of iterates has not yet converged. The local optimizer was terminated because no further improvement could be made in the merit function (see Section 9.1).

info$[i-1] = 6$
x does not satisfy the first-order Kuhn–Tucker conditions (see Section 11.1) and no improved point for the merit function (see Section 9.1) could be found during the final linesearch.

This sometimes occurs because an overly stringent accuracy has been requested, i.e., the value of the optional argument **Optimality Tolerance** (default value $= \epsilon_R^{0.8}$, where ϵ_R is the value of the optional argument **Function Precision** (default value $= \epsilon^{0.9}$, where ϵ is the ***machine precision***)) is too small.

As usual 0 denotes success.

If **fail**.**code** = NW_SOME_SOLUTIONS on exit, then not all **nb** solutions have been found, and **info**$[$**nb** $- 1]$ contains the number of solutions actually found.

36: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_BOUND

On entry, $\mathbf{bl}[i-1] > \mathbf{bu}[i-1]$: $i = \langle value \rangle$.
Constraint: $\mathbf{bl}[i-1] \leq \mathbf{bu}[i-1]$, for all i.

NE_DERIV_ERRORS

User-supplied derivatives probably wrong.

The user-supplied derivatives of the objective function and/or nonlinear constraints appear to be incorrect.

Large errors were found in the derivatives of the objective function and/or nonlinear constraints. This value of **fail.code** will occur if the verification process indicated that at least one gradient or Jacobian element had no correct figures. You should refer to or enable the printed output to determine which elements are suspected to be in error.

As a first-step, you should check that the code for the objective and constraint values is correct – for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values $x = 0$ or $x = 1$ are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Gradient checking will be ineffective if the objective function uses information computed by the constraints, since they are not necessarily computed before each function evaluation.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

NE_INITIALIZATION

Failed to initialize optional argument arrays.

NE_INT

On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} > 0$.

On entry, $\mathbf{nclin} = \langle value \rangle$.
Constraint: $\mathbf{nclin} \geq 0$.

On entry, $\mathbf{ncnln} = \langle value \rangle$.
Constraint: $\mathbf{ncnln} \geq 0$.

NE_INT_2

On entry, $\mathbf{ldc} = \langle value \rangle$ and $\mathbf{ncnln} = \langle value \rangle$.
Constraint: $\mathbf{ldc} \geq \mathbf{ncnln}$.

On entry, $\mathbf{ldcjac} = \langle value \rangle$ and $\mathbf{ncnln} = \langle value \rangle$.
Constraint: $\mathbf{ldcjac} \geq \mathbf{ncnln}$.

On entry, $\mathbf{ldobjgrd} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{ldobjgrd} \geq \mathbf{n}$.

On entry, $\mathbf{ldr} = \langle value \rangle$ and $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{ldr} \geq \mathbf{n}$.

On entry, **ldx** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **ldx** ≥ **n**.

On entry, **nb** = ⟨*value*⟩ and **npts** = ⟨*value*⟩.
Constraint: 1 ≤ **nb** ≤ **npts**.

On entry, **sdr** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **sdr** ≥ **n**.

NE_INT_3

On entry, **ncnln** > 0, **sdcjac** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: if **ncnln** > 0, **sdcjac** ≥ **n**.

On entry, **tda** = ⟨*value*⟩, **nclin** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **tda** ≥ **n**.

NE_INT_4

On entry, **ldclamda** = ⟨*value*⟩, **n** = ⟨*value*⟩, **nclin** = ⟨*value*⟩ and **ncnln** = ⟨*value*⟩.
Constraint: **ldclamda** ≥ **n** + **nclin** + **ncnln**.

On entry, **ldistate** = ⟨*value*⟩, **n** = ⟨*value*⟩, **nclin** = ⟨*value*⟩ and **ncnln** = ⟨*value*⟩.
Constraint: **ldistate** ≥ **n** + **nclin** + **ncnln**.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

NE_LIN_NOT_FEASIBLE

No solution obtained. Linear constraints may be infeasible. nag_glopt_nlp_multistart_sqp (e05ucc) has terminated without finding any solutions. The majority of calls to the local optimizer have failed to find a feasible point for the linear constraints and bounds, which means that either no feasible point exists for the given value of the optional argument **Linear Feasibility Tolerance** (default value $\sqrt{\epsilon}$, where ϵ is the ***machine precision***), or no feasible point could be found in the number of iterations specified by the optional argument **Minor Iteration Limit**. You should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision σ, you should ensure that the value of the optional argument **Linear Feasibility Tolerance** is greater than σ. For example, if all elements of A_L are of order unity and are accurate to only three decimal places, **Linear Feasibility Tolerance** should be at least 10^{-3}.

NE_NO_SOLUTION

No solution obtained. Many potential solutions reach iteration limit. The **Iteration Limit** may be changed using nag_glopt_opt_set (e05zkc).

NE_NONLIN_NOT_FEASIBLE

nag_glopt_nlp_multistart_sqp (e05ucc) has failed to find any solutions. The majority of local optimizations could not find a feasible point for the nonlinear constraints. The problem may have no feasible solution. This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) No solution obtained. Nonlinear constraints may be infeasible.

NE_USER_STOP

User terminated computation from **start** procedure: **mode** = ⟨*value*⟩.

NW_SOME_SOLUTIONS

Only ⟨*value*⟩ solutions obtained.

Not all **nb** solutions have been found. **info**[**nb** − 1] contains the number actually found.

7 Accuracy

If **fail**.**code** = NE_NOERROR on exit and the value of **info**[$i - 1$] = 0, then the vector returned in the array **x** for solution i is an estimate of the solution to an accuracy of approximately **Optimality Tolerance**.

8 Parallelism and Performance

nag_glopt_nlp_multistart_sqp (e05ucc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_glopt_nlp_multistart_sqp (e05ucc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

In these implementations, this may make calls to the user supplied functions from within an OpenMP parallel region. Thus OpenMP directives within the user functions should be avoided, unless you are using the same OpenMP runtime library (which normally means using the same compiler) as that used to build your NAG Library implementation, as listed in the Installers' Note. You must also ensure that you use the NAG communication argument **comm** in a thread safe manner, which is best achieved by only using it to supply read-only data to the user functions.

Please consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

You should be wary of requesting much intermediate output from the local optimizer, since large volumes may be produced if **npts** is large.

If **NULLFN** is supplied an actual argument from **start** then the default NAG function makes use of the NAG quasi-random Sobol generator (nag_quasi_init (g05ylc) and nag_quasi_rand_uniform (g05ymc)). If this is used as an argument for **start**, by specifying **NULLFN** in the calling sequence (see the description of **start**) and **repeat** = Nag_FALSE then a randomly chosen value for **iskip** is used, otherwise **iskip** is set to 100. If **repeat** is set to Nag_FALSE and the program is executed several times, each time producing the same best answer, then there is increased probability that this answer is a global minimum. However, if it is important that identical results be obtained on successive runs, then **repeat** should be set to Nag_TRUE.

9.1 Description of the Printed Output

This section describes the intermediate printout and final printout produced by nag_glopt_nlp_multistart_sqp (e05ucc). The intermediate printout is a subset of the monitoring information produced by the function at every iteration (see Section 13). You can control the level of printed output (see the description of the optional arguments **Major Print Level** and **Minor Print Level**). Note that the intermediate printout and final printout are produced only if **Major Print Level** ≥ 10 or **Minor Print Level** ≥ 10.

The following line of summary output (< 80 characters) is produced at every major iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration for each starting point.

`Maj` is the major iteration count.

`Mnr` is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 11). Note that `Mnr` may be greater than the optional argument **Minor Iteration Limit** if some iterations are required for the feasibility phase.

`Step` is the step α_k taken along the computed search direction. On reasonably well-behaved local problems, the unit step (i.e., $\alpha_k = 1$) will be taken as the solution is approached.

`Merit Function` is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty arguments (see Section 11.3). As the solution is approached, `Merit Function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line) then the merit function is a large multiple of the constraint violations, weighted by the penalty arguments. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or the local optimizer terminates. Repeated failures will prevent a feasible point being found for the nonlinear constraints.

If there are no nonlinear constraints present (i.e., **ncnln** $= 0$) then this entry contains `Objective`, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

`Norm Gz` is $\|Z^{\mathrm{T}} g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 11.2). `Norm Gz` will be approximately zero in the neighbourhood of a solution.

`Violtn` is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if **ncnln** is zero). `Violtn` will be approximately zero in the neighbourhood of a solution.

`Cond Hz` is a lower bound on the condition number of the projected Hessian approximation H_Z ($H_Z = Z^{\mathrm{T}} H_{\mathrm{FR}} Z = R_Z^{\mathrm{T}} R_Z$; see (6)). The larger this number, the more difficult the local problem.

`M` is printed if the quasi-Newton update has been modified to ensure that the Hessian approximation is positive definite (see Section 11.4).

`I` is printed if the QP subproblem has no feasible point.

`C` is printed if central differences have been used to compute the unspecified objective and constraint gradients. If the value of `Step` is zero then the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is resolved with the central difference gradient and Jacobian.) If the value of `Step` is nonzero then central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 11.1).

`L` is printed if the linesearch has produced a relative change in x greater than the value defined by the optional argument **Step Limit**. If this output occurs frequently during later iterations of the run, optional argument **Step Limit** should be set to a larger value.

`R` is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned then the approximate Hessian is refactorized using column interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.

The final printout includes a listing of the status of every variable and constraint. The following describes the printout for each variable. A full stop (.) is printed for any numerical value that is zero.

`Varbl` gives the name (`V`) and index j, for $j = 1, 2, \ldots, n$, of the variable.

`State` gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower

bounds by more than the **Feasibility Tolerance**, `State` will be `++` or `--` respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)

A key is sometimes printed before `State`.

`A` *Alternative optimum possible*. The variable is active at one of its bounds, but its Lagrange multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound then there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case the values of the Lagrange multipliers might also change.

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

`I` *Infeasible*. The variable is currently violating one of its bounds by more than the **Feasibility Tolerance**.

`Value` is the value of the variable at the final iteration.

`Lower Bound` is the lower bound specified for the variable. `None` indicates that $\mathbf{bl}[j-1] \leq -bigbnd$.

`Upper Bound` is the upper bound specified for the variable. `None` indicates that $\mathbf{bu}[j-1] \geq bigbnd$.

`Lagr Mult` is the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR` unless $\mathbf{bl}[j-1] \leq -bigbnd$ and $\mathbf{bu}[j-1] \geq bigbnd$, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL` and non-positive if `State` is `UL`.

`Slack` is the difference between the variable `Value` and the nearer of its (finite) bounds $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$. A blank entry indicates that the associated variable is not bounded (i.e., $\mathbf{bl}[j-1] \leq -bigbnd$ and $\mathbf{bu}[j-1] \geq bigbnd$).

The meaning of the printout for linear and nonlinear constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$ are replaced by $\mathbf{bl}[n+j-1]$ and $\mathbf{bu}[n+j-1]$ respectively, and with the following changes in the heading:

`L Con` gives the name (`L`) and index j, for $j = 1, 2, \ldots, n_L$, of the linear constraint.

`N Con` gives the name (`N`) and index $(j - n_L)$, for $j = n_L + 1, \ldots, n_L + n_N$, of the nonlinear constraint.

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the `Slack` column to become positive.

Numerical values are output with a fixed number of digits; they are not guaranteed to be accurate to this precision.

10 Example

This example finds the global minimum of the two-dimensional Schwefel function:

$$\underset{\mathbf{x}\in R^2}{\text{minimize}} f = \sum_{j=1}^{2} x_j \sin\left(\sqrt{|x_j|}\right)$$

subject to the constraints:

$$
\begin{gathered}
-10000 < 3.0x_1 - 2.0x_2 < 10.0, \\
-1.0 < x_1^2 - x_2^2 + 3.0x_1x_2 < 500000.0, \\
-0.9 < \cos\Big((x_1/200)^2 + (x_2/100)\Big) < 0.9, \\
-500 \leq x_1 \leq 500, \\
-500 \leq x_2 \leq 500.
\end{gathered}
$$

10.1 Program Text

```
/* nag_glopt_nlp_multistart_sqp (e05ucc) Example Program.
 *
 * Copyright 2013 Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */

#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nage05.h>
#include <nagf16.h>
#include <nagg05.h>
#include <nagx04.h>

#ifdef __cplusplus
extern "C" {
#endif
  static void NAG_CALL schwefel_obj(Integer *mode, Integer n,
                                    const double *x, double *objf,
                                    double *objgrd, Integer nstate,
                                    Nag_Comm *comm);
  static void NAG_CALL schwefel_confun(Integer *mode, Integer ncnln,
                                       Integer n, Integer tdcjsl,
                                       const Integer *needc,
                                       const double *x, double *c,
                                       double *cjsl, Integer nstate,
                                       Nag_Comm *comm);
  static void NAG_CALL mystart(Integer npts, double quas[], Integer n,
                               Nag_Boolean repeat, const double bl[],
                               const double bu[], Nag_Comm *comm,
                               Integer *mode);
#ifdef __cplusplus
}
#endif

int main(void)
{

  Integer      exit_status = 0;
  Integer      print_all_solutions = 0;
  Integer      liopts = 740, lopts = 485, n = 2, nclin = 1, ncnln = 2;

  /* Scalars */
  Integer      i, ic, j, l, nb, npts, tda, ldcjac, sdcjac, ldr, sdr,
               ldx, ldobjgrd, ldclamda, ldistate, ldc;

  /* Arrays */
  static double ruser[3] = {-1.0, -1.0, -1.0};
  double       *a=0, *bl=0, *bu=0, *c=0, *cjac=0, *clamda=0, *objf=0,
               *objgrd=0, *r=0, *opts=0, *work=0, *x=0;
  Integer      *info=0, *istate=0, *iter=0, *iopts=0;
  char         nag_enum_arg[40];

  /* Nag Types */
  NagError     fail;
  Nag_Comm     comm;
  Nag_Boolean  repeat;
```

```
  INIT_FAIL(fail);

  printf("nag_glopt_nlp_multistart_sqp (e05ucc) Example Program Results\n\n");

  /* For communication with user-supplied functions: */
  comm.user = ruser;

  /* Skip heading in data file */
  scanf("%*[^\n] ");
  scanf("%ld%ld%*[^\n]", &nb, &npts);
  scanf("%39s%*[^\n]", nag_enum_arg);
  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value
   */
  repeat = (Nag_Boolean) nag_enum_name_to_value(nag_enum_arg);

  /* The minimum trailing dimension for a is tda = n (or 1). */
  if (nclin>0)
    {
      tda = n;
      if (!(a = NAG_ALLOC(nclin*tda, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    tda = 1;

#define A(I,J) a[(I-1)*tda + (J-1)]
#define X(I,J) x[(J-1)*ldx + (I-1)]
#define ISTATE(I,J) istate[(J-1)*ldistate + (I-1)]
#define CLAMDA(I,J) clamda[(J-1)*ldclamda + (I-1)]
#define C(I,J) c[(J-1)*ldc + (I-1)]

  ldx = n;
  ldobjgrd = n;
  ldc = ncnln;
  ldcjac = ncnln;

  if (ncnln>0)
    {
      sdcjac = n;
      if (
          !(c = NAG_ALLOC(ldc*nb, double)) ||
          !(cjac = NAG_ALLOC(ldcjac*sdcjac*nb, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    sdcjac = 0;

  ldr = n;
  sdr = n;
  ldclamda = n + nclin + ncnln;
  ldistate = n + nclin + ncnln;

  if (
      !(bl = NAG_ALLOC(n + nclin + ncnln, double)) ||
      !(bu = NAG_ALLOC(n + nclin + ncnln, double)) ||
      !(clamda = NAG_ALLOC(ldclamda*nb, double)) ||
      !(objf = NAG_ALLOC(nb, double)) ||
      !(objgrd = NAG_ALLOC(ldobjgrd*nb, double)) ||
      !(r = NAG_ALLOC(ldr*sdr*nb, double)) ||
      !(opts = NAG_ALLOC(lopts, double)) ||
      !(work = NAG_ALLOC(nclin, double)) ||
```

```
    !(x = NAG_ALLOC(ldx*nb, double)) ||
    !(info = NAG_ALLOC(nb, Integer)) ||
    !(istate = NAG_ALLOC(ldistate*nb, Integer)) ||
    !(iter = NAG_ALLOC(nb, Integer)) ||
    !(iopts = NAG_ALLOC(liopts, Integer))
    )
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

bl[0] = -500.0;
bl[1] = -500.0;
bl[2] = -10000.0;
bl[3] = -1.0;
bl[4] = -0.9;
bu[0] = 500.0;
bu[1] = 500.0;
bu[2] = 10.0;
bu[3] = 500000.0;
bu[4] = 0.9;
A(1, 1) = 3.0;
A(1, 2) = -2.0;

/* Initialize nag_glopt_nlp_multistart_sqp (e05ucc).
 * nag_glopt_opt_set (e05zkc).
 * Option setting routine for global optimization.
 */
nag_glopt_opt_set("Initialize = e05ucc", iopts, liopts, opts, lopts, &fail);
if (fail.code != NE_NOERROR) {
  printf("Error from nag_glopt_opt_set (e05zkc).\n%s\n", fail.message);
  exit_status = 1;
  goto END;
}

/* Solve the problem with repeatable random starting points using
 * nag_glopt_nlp_multistart_sqp (e05ucc).
 * Global optimization using multi-start, nonlinear constraints.
 */
nag_glopt_nlp_multistart_sqp(n, nclin, ncnln, a, tda, bl, bu,
                             schwefel_confun, schwefel_obj, npts, x, ldx,
                             mystart, repeat, nb, objf, objgrd, ldobjgrd,
                             iter, c, ldc, cjac, ldcjac, sdcjac, r, ldr, sdr,
                             clamda, ldclamda, istate, ldistate, iopts, opts,
                             &comm, info, &fail);
/* Check for error exits. */
switch (fail.code)
  {
  case NE_NOERROR:
    l = nb;
    break;
  case NW_SOME_SOLUTIONS:
    l = info[nb-1];
    printf("Only %ld solutions found\n", l);
    break;
  default:
    exit_status = 2;
    printf("Error from nag_glopt_nlp_multistart_sqp (e05ucc)\n%s\n",
           fail.message);
    goto END;
  }

for (i=1; i<=l; i++) {
  printf("Solution number %ld\n\n", i);
  printf("Local minimization exited with code %ld\n", info[i-1]);
  printf("\nVarbl  Istate   Value         Lagr Mult\n\n\n");

  for (j=1; j<=n; j++)
    printf("V %3ld %3ld    %14.6g  %12.4g\n", j, ISTATE(j,i),
           X(j,i), CLAMDA(j,i));
```

```
    if (nclin>0) {
      printf("\nL Con  Istate   Value         Lagr Mult\n\n");

      /* nag_dgemv (f16pac) performs the matrix vector multiplication A*x
       * (linear constraint values) and puts the result in
       * the first nclin locations of work.
       */
      nag_dgemv(Nag_RowMajor, Nag_NoTrans, nclin, n, 1.0, a, tda, &X(1,i), 1,
                0.0, work, 1, &fail);
      if (fail.code != NE_NOERROR) {
        printf("Error from nag_dgemv (f16pac).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }

      for (j = n+1; j <= n+nclin; j++)
        printf("L %3ld %3ld    %14.6g  %12.4g\n", j-n,
               ISTATE(j,i), work[j-n-1], CLAMDA(j,i));
    }

    if (ncnln>0) {
      printf("\n\nN Con  Istate   Value         Lagr Mult\n\n");

      for (j = n+nclin+1; j <= n+nclin+ncnln; j++) {
        ic = j - n - nclin;
        printf("N %3ld %3ld    %14.6g  %12.4g\n", ic,
               ISTATE(j,i), C(ic,i), CLAMDA(j,i));
      }

    }

    printf("\n\nFinal objective value = %15.7g\n", objf[i-1]);

    printf("\nQP multipliers\n");
    for (j = 1; j <= n+nclin+ncnln; j++)
      printf("%12.4e\n", CLAMDA(j,i));

    if (l==1) goto END;

    if (print_all_solutions==0) {
      printf("\n(Printing of further solutions suppressed)\n");
      goto END;
    }

    printf("\n");
    for (j = 0; j < 61; j++)
      printf("-");
    printf("\n");
  }

 END:
  NAG_FREE(a);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(c);
  NAG_FREE(cjac);
  NAG_FREE(clamda);
  NAG_FREE(objf);
  NAG_FREE(objgrd);
  NAG_FREE(r);
  NAG_FREE(opts);
  NAG_FREE(work);
  NAG_FREE(x);
  NAG_FREE(info);
  NAG_FREE(istate);
  NAG_FREE(iter);
  NAG_FREE(iopts);
  return exit_status;
}
```

```
static void NAG_CALL schwefel_obj(Integer *mode, Integer n, const double *x,
                                  double *objf, double *objgrd, Integer nstate,
                                  Nag_Comm *comm)
{
  /* Scalars */
  Integer i;

  if (comm->user[0] == -1.0)
    {
      printf("(User-supplied callback schwefel_obj, first invocation.)\n");
      comm->user[0] = 0.0;
    }

  if (nstate == 1)
    {
      /* This is the first call.
       * Take any special action here if desired.
       */
    }

  if (*mode==0 || *mode==2) {
    /* Evaluate the objective function. */
    *objf = 0.0;
    for (i = 0; i < n; i++)
      *objf += x[i]*sin(sqrt(fabs(x[i])));
  }

  if (*mode==1 || *mode==2) {
    /* Calculate the gradient of the objective function. */
    for (i = 0; i < n; i++) {
      double t;
      t = sqrt(fabs(x[i]));
      objgrd[i] = sin(t) + 0.5*t*cos(t);
    }
  }
}

static void NAG_CALL schwefel_confun(Integer *mode, Integer ncnln, Integer n,
                                     Integer tdcjsl, const Integer *needc,
                                     const double *x, double *c, double *cjsl,
                                     Integer nstate, Nag_Comm *comm)
{

  /* Scalars */
  double  t1, t2;
  Integer k;
  Nag_Boolean evalc, evalcjsl;
  if (comm->user[1] == -1.0)
    {
      printf("(User-supplied callback schwefel_confun, first invocation.)\n");
      comm->user[1] = 0.0;
    }

  if (nstate == 1)
    {
      /* This is the first call.
       * Take any special action here if desired.
       */
    }

  /* mode: what is required - constraints, derivatives, or both? */
  evalc = (*mode == 0 || *mode == 2) ? Nag_TRUE : Nag_FALSE;
  evalcjsl = (*mode == 1 || *mode == 2) ? Nag_TRUE : Nag_FALSE;

  for (k = 1; k <= ncnln; k++) {
    if (needc[k - 1] <= 0) continue;
    if (evalc == Nag_TRUE) {
      /* Constraint values are required. */
      switch (k) {
      case 1:
        c[k - 1] = pow(x[0], 2.0) - pow(x[1], 2.0) + 3.0*x[0]*x[1];
```

```
        break;
      case 2:
        c[k - 1] = cos(pow((x[0]/200.0), 2.0) + (x[1]/100.0));
        break;
      default:
        /* This constraint is not coded (there are only two).
         * Terminate.
         */
        *mode = -1;
        break;
      }
    }
    if (*mode < 0) break;
    if (evalcjsl == Nag_TRUE) {
      /* Constraint derivatives are required. */
#define CJSL(K, J) cjsl[(K-1)*tdcjsl + (J-1)]
      switch (k) {
      case 1:
        CJSL(k, 1) = 2.0*x[0] + 3.0*x[1];
        CJSL(k, 2) = -2.0*x[1] + 3.0*x[0];
        break;
      case 2:
        t1 = x[0]/200.0;
        t2 = x[1]/100.0;
        CJSL(k, 1) = -sin(pow(t1, 2.0) + t2) * (2.0*t1)/200.0;
        CJSL(k, 2) = -sin(pow(t1, 2.0) + t2)/100.0;
        break;
      }
#undef CJSL
    }
  }
}

static void NAG_CALL mystart(Integer npts, double quas[], Integer n,
                             Nag_Boolean repeat, const double bl[],
                             const double bu[], Nag_Comm *comm,
                             Integer *mode)
{
  /* Only nonzero elements of quas need to be specified here. */
  Integer i, j;
  if (comm->user[2] == -1.0)
    {
      printf("(User-supplied callback mystart, first invocation.)\n");
      comm->user[2] = 0.0;
    }
#define QUAS(J, I) quas[(J-1)*npts + (I-1)]
  if (repeat == Nag_TRUE) {
    /* Generate a uniform spread of points between bl and bu. */
    for (j = 1; j <= npts; j++)
      for (i = 1; i <= n; i++)
        QUAS(i,j) = bl[i-1] + (bu[i-1]-bl[i-1])*(double)(j-1)/(double)(npts-1);
  }
  else {
    /* Generate a non-repeatable spread of points between bl and bu. */
    Nag_BaseRNG genid;
    Integer lstate, subid;
    Integer *state=0;
    NagError fail;

    INIT_FAIL(fail);

    genid = Nag_WichmannHill_I;
    subid = 53;
    lstate = -1;

    nag_rand_init_nonrepeatable(genid, subid, NULL, &lstate, &fail);
    if (fail.code != NE_NOERROR) {
      *mode = -1;
      return;
    }
```

```
    if (!(state = NAG_ALLOC(lstate, Integer))) {
      *mode = -1;
      return;
    }

    nag_rand_init_nonrepeatable(genid, subid, state, &lstate, &fail);
    if (fail.code != NE_NOERROR) {
      *mode = -1;
      goto END;
    }

    for (j = 2; j <= npts; j++)
      for (i = 1; i <= n; i++) {
        nag_rand_uniform(1, bl[i-1], bu[i-1], state, &QUAS(i, j), &fail);
        if (fail.code != NE_NOERROR) {
          *mode = -1;
          goto END;
        }
      }
  END:
    NAG_FREE(state);
  }
#undef QUAS
}
```

10.2 Program Data

```
nag_glopt_nlp_multistart_sqp (e05ucc) Example Program Data
10 1000                                 : nb, npts
Nag_TRUE                                : repeat
```

10.3 Program Results

```
nag_glopt_nlp_multistart_sqp (e05ucc) Example Program Results

(User-supplied callback mystart, first invocation.)
(User-supplied callback schwefel_confun, first invocation.)
(User-supplied callback schwefel_obj, first invocation.)
Solution number 1

Local minimization exited with code 0

Varbl  Istate   Value         Lagr Mult


V   1   0          -394.151             0
V   2   0          -433.491             0

L Con  Istate   Value         Lagr Mult

L   1   0          -315.472             0


N Con  Istate   Value         Lagr Mult

N   1   0            480024             0
N   2   2               0.9        -718.9


Final objective value =       -731.7064

QP multipliers
  0.0000e+00
  0.0000e+00
  0.0000e+00
  0.0000e+00
 -7.1894e+02

(Printing of further solutions suppressed)
```

11 Algorithmic Details

This section contains a detailed description of the method used by nag_glopt_nlp_multistart_sqp (e05ucc).

11.1 Overview

nag_glopt_nlp_multistart_sqp (e05ucc) uses a local optimizer that is essentially identical to the function NPSOL described in Gill *et al.* (1986b).

For the local optimizer, at a solution of (1), some of the constraints will be *active*, i.e., satisfied exactly. An active simple bound constraint implies that the corresponding variable is *fixed* at its bound, and hence the variables are partitioned into *fixed* and *free* variables. Let C denote the m by n matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by n_{FX}, with n_{FR} $(n_{\mathrm{FR}} = n - n_{\mathrm{FX}})$ the number of free variables. The subscripts 'FX' and 'FR' on a vector or matrix will denote the vector or matrix composed of the elements corresponding to fixed or free variables.

A point x is a *first-order Kuhn–Tucker point* for (1) (see Powell (1974)) if the following conditions hold:

(i) x is feasible;

(ii) there exist vectors ξ and λ (*the Lagrange multiplier vectors for the bound and general constraints*) such that

$$g = C^{\mathrm{T}}\lambda + \xi \tag{2}$$

where g is the gradient of F evaluated at x, and $\xi_j = 0$ if the jth variable is free.

(iii) The Lagrange multiplier corresponding to an inequality constraint active at its lower bound must be non-negative, and non-positive for an inequality constraint active at its upper bound.

Let Z denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of C_{FR}; i.e., $C_{\mathrm{FR}}Z = 0$. An equivalent statement of the condition (2) in terms of Z is

$$Z^{\mathrm{T}}g_{\mathrm{FR}} = 0.$$

The vector $Z^{\mathrm{T}}g_{\mathrm{FR}}$ is termed the *projected gradient* of F at x. Certain additional conditions must be satisfied in order for a first-order Kuhn–Tucker point to be a solution of (1) (see Powell (1974)).

The local optimizer implements a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher (1987), Gill *et al.* (1981) and Powell (1983).

The basic structure of the local optimizer involves *major* and *minor* iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to x^*, a first-order Kuhn–Tucker point of (1). At a typical major iteration, the new iterate $\bar{x}$ is defined by

$$\bar{x} = x + \alpha p \tag{3}$$

where x is the current iterate, the non-negative scalar α is the *step length,* and p is the *search direction.* (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Also associated with each major iteration are estimates of the Lagrange multipliers and a prediction of the active set.

The search direction p in (3) is the solution of a quadratic programming subproblem of the form

$$\underset{p}{\text{minimize}}\ g^{\mathrm{T}}p + \tfrac{1}{2}p^{\mathrm{T}}Hp \quad \text{subject to} \quad \bar{l} \leq \begin{Bmatrix} p \\ A_L p \\ A_N p \end{Bmatrix} \leq \bar{u}, \tag{4}$$

where g is the gradient of F at x, the matrix H is a positive definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 11.4), and A_N is the Jacobian matrix of c evaluated at x. (Finite difference estimates may be used for g and A_N; see the optional argument **Derivative Level**.) Let l in (1) be partitioned into three sections: l_B, l_L and l_N, corresponding to the bound, linear and nonlinear constraints. The vector $\bar{l}$ in (4) is similarly partitioned, and is defined as

$$\bar{l}_B = l_B - x, \quad \bar{l}_L = l_L - A_L x, \quad \text{and} \quad \bar{l}_N = l_N - c,$$

where c is the vector of nonlinear constraints evaluated at x. The vector $\bar{u}$ is defined in an analogous fashion.

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities `Bnd`, `Lin` and `Nln` in the monitoring file output of nag_glopt_nlp_multistart_sqp (e05ucc); see Section 13.) The local optimizer repeatedly solves as major iterations quadratic programming problems. These are themselves iterative procedures and comprise the minor iterations. (More details about solving the subproblem are given in Section 11.2.)

Certain matrices associated with the QP subproblem are relevant in the major iterations. Let the subscripts 'FX' and 'FR' refer to the *predicted* fixed and free variables, and let C denote the m by n matrix of gradients of the general linear and nonlinear constraints in the predicted active set. First, we have available the TQ factorization of C_{FR}:

$$C_{\mathrm{FR}} Q_{\mathrm{FR}} = \begin{pmatrix} 0 & T \end{pmatrix}, \tag{5}$$

where T is a nonsingular m by m reverse-triangular matrix (i.e., $t_{ij} = 0$ if $i + j < m$), and the nonsingular n_{FR} by n_{FR} matrix Q_{FR} is the product of orthogonal transformations (see Gill *et al.* (1984)). Second, we have the upper triangular Cholesky factor R of the *transformed and reordered* Hessian matrix

$$R^{\mathrm{T}} R = H_Q \equiv Q^{\mathrm{T}} \tilde{H} Q, \tag{6}$$

where $\tilde{H}$ is the Hessian H with rows and columns permuted so that the free variables are first, and Q is the n by n matrix

$$Q = \begin{pmatrix} Q_{\mathrm{FR}} & \\ & I_{\mathrm{FX}} \end{pmatrix} \tag{7}$$

with I_{FX} the identity matrix of order n_{FX}. If the columns of Q_{FR} are partitioned so that

$$Q_{\mathrm{FR}} = \begin{pmatrix} Z & Y \end{pmatrix},$$

the n_Z ($n_Z \equiv n_{\mathrm{FR}} - m$) columns of Z form a basis for the null space of C_{FR}. The matrix Z is used to compute the projected gradient $Z^{\mathrm{T}} g_{\mathrm{FR}}$ at the current iterate. (The values `Nz` and `Norm Gz` printed by nag_glopt_nlp_multistart_sqp (e05ucc) give n_Z and $\|Z^{\mathrm{T}} g_{\mathrm{FR}}\|$; see Section 13.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (4) is identical to the correct active set in a neighbourhood of x^*. In the local optimizer underlying nag_glopt_nlp_multistart_sqp (e05ucc), this feature is exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints in the local optimizer also saves computation in factorizing C_{FR} and H_Q.

Once p has been computed, the major iteration proceeds by determining a step length α that produces a 'sufficient decrease' in an augmented Lagrangian *merit function* (see Section 11.3). Finally, the approximation to the transformed Hessian matrix H_Q is updated using a modified BFGS quasi-Newton update (see Section 11.4) to incorporate new curvature information obtained in the move from x to $\bar{x}$.

On entry to the local optimizer, an iterative procedure is executed, starting with the user-supplied initial point, to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by optional argument **Linear Feasibility Tolerance**). If no feasible point exists for the bound and linear constraints, (1) has no solution and the local optimizer terminates. Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite difference interval (see the discussion of optional argument **Difference Interval**). In contrast to the bounds and linear constraints, it must be emphasized that *the nonlinear constraints will not generally be satisfied until an optimal point* is reached.

Facilities are provided to check whether the user-supplied gradients appear to be correct (see the description of the optional argument **Verify**). In general, the check is provided at the first point that is feasible with respect to the linear constraints and bounds. However, you may request that the check be performed at the initial point.

In summary, the local method of nag_glopt_nlp_multistart_sqp (e05ucc) first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem;

(b) a linesearch with an augmented Lagrangian merit function; and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian function.

These three procedures are described in more detail in Sections 11.2 to 11.4.

11.2 Solution of the Quadratic Programming Subproblem

The search direction p is obtained by solving (4) using a method (see Gill *et al.* (1986)) that was specifically designed to be used within an SQP algorithm for nonlinear programming. This method is based on a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the *feasibility phase*), and minimizing the quadratic objective function within the feasible region (the *optimality phase*). The computations in both phases are performed by the same functions. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let p denote the current estimate of the solution of (4); the new iterate $\bar{p}$ is defined by

$$\bar{p} = p + \sigma d \tag{8}$$

where, as in (3), σ is a non-negative step length and d is a search direction.

At the beginning of each iteration of the QP method, a *working* set is defined of constraints (general and bound) that are satisfied exactly. The vector d is then constructed so that the values of constraints in the working set remain *unaltered* for any move along d. For a bound constraint in the working set, this property is achieved by setting the corresponding element of d to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the elements associated with the fixed and free variables.

Let C denote the sub-matrix of rows of

$$\begin{pmatrix} A_L \\ A_N \end{pmatrix}$$

corresponding to general constraints in the working set. The general constraints in the working set will remain unaltered if

$$C_{\mathrm{FR}} d_{\mathrm{FR}} = 0, \tag{9}$$

which is equivalent to defining d_{FR} as

$$d_{\mathrm{FR}} = Z d_Z \tag{10}$$

for some vector d_Z, where Z is the matrix associated with the TQ factorization (5) of C_{FR}.

The definition of d_Z in (10) depends on whether the current p is feasible. If not, d_Z is zero except for an element γ in the jth position, where j and γ are chosen so that the sum of infeasibilities is decreasing along d. (For further details, see Gill *et al.* (1986).) In the feasible case, d_Z satisfies the equations

$$R_Z^{\mathrm{T}} R_Z d_Z = -Z^{\mathrm{T}} q_{\mathrm{FR}}, \tag{11}$$

where R_Z is the Cholesky factor of $Z^{\mathrm{T}} H_{\mathrm{FR}} Z$ and q is the gradient of the quadratic objective function ($q = g + Hp$). (The vector $Z^{\mathrm{T}} q_{\mathrm{FR}}$ is the projected gradient of the QP.) With (11), $p + d$ is the minimizer of the quadratic objective function subject to treating the constraints in the working set as equalities.

If the QP projected gradient is zero, the current point is a constrained stationary point in the subspace defined by the working set. During the feasibility phase, the projected gradient will usually be zero only at a vertex (although it may vanish at non-vertices in the presence of constraint dependencies). During the optimality phase, a zero projected gradient implies that p minimizes the quadratic objective function when the constraints in the working set are treated as equalities. In either case, Lagrange multipliers are computed. Given a positive constant δ of the order of the ***machine precision***, the Lagrange multiplier μ_j corresponding to an inequality constraint in the working set is said to be *optimal* if $\mu_j \leq \delta$ when the jth constraint is at its *upper bound*, or if $\mu_j \geq -\delta$ when the associated constraint is at its *lower bound*. If any multiplier is nonoptimal, the current objective function (either the true objective or the sum of infeasibilities) can be reduced by deleting the corresponding constraint from the working set.

If optimal multipliers occur during the feasibility phase and the sum of infeasibilities is nonzero, no feasible point exists. The QP algorithm will then continue iterating to determine the minimum sum of infeasibilities. At this point, the Lagrange multiplier μ_j will satisfy $-(1+\delta) \leq \mu_j \leq \delta$ for an inequality constraint at its upper bound, and $-\delta \leq \mu_j \leq (1+\delta)$ for an inequality at its lower bound. The Lagrange multiplier for an equality constraint will satisfy $\left|\mu_j\right| \leq 1+\delta$.

The choice of step length σ in the QP iteration (8) is based on remaining feasible with respect to the satisfied constraints. During the optimality phase, if $p+d$ is feasible, σ will be taken as unity. (In this case, the projected gradient at $\bar{p}$ will be zero.) Otherwise, σ is set to σ_{M}, the step to the 'nearest' constraint, which is added to the working set at the next iteration.

Each change in the working set leads to a simple change to C_{FR}: if the status of a general constraint changes, a *row* of C_{FR} is altered; if a bound constraint enters or leaves the working set, a *column* of C_{FR} changes. Explicit representations are recurred of the matrices T, Q_{FR} and R, and of the vectors $Q^{\mathrm{T}}q$ and $Q^{\mathrm{T}}g$.

11.3 The Merit Function

After computing the search direction as described in Section 11.2, each major iteration proceeds by determining a step length α in (3) that produces a 'sufficient decrease' in the augmented Lagrangian merit function

$$L(x,\lambda,s) = F(x) - \sum_i \lambda_i(c_i(x)-s_i) + \tfrac{1}{2}\sum_i \rho_i(c_i(x)-s_i)^2, \tag{12}$$

where x, λ and s vary during the linesearch. The summation terms in (12) involve only the *nonlinear* constraints. The vector λ is an estimate of the Lagrange multipliers for the nonlinear constraints of (1). The non-negative *slack variables* $\{s_i\}$ allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (4) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector ρ of *penalty arguments* is initialized to zero at the beginning of the first major iteration. Thereafter, selected elements are increased whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of ρ (the printed quantity `Penalty`; see Section 13) is generally nondecreasing, although each ρ_i may be reduced a limited number of times.

The merit function (12) and its global convergence properties are described in Gill *et al.* (1986a).

11.4 The Quasi-Newton Update

The matrix H in (4) is a *positive definite quasi-Newton* approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel (1983).) At the end of each major iteration, a new Hessian approximation $\bar{H}$ is defined as a rank-two modification of H. In the local optimizer used by nag_glopt_nlp_multistart_sqp (e05ucc), the BFGS (Broyden–Fletcher–Goldfarb–Shanno) quasi-Newton update is used:

$$\bar{H} = H - \frac{1}{s^{\mathrm{T}}Hs}Hss^{\mathrm{T}}H + \frac{1}{y^{\mathrm{T}}s}yy^{\mathrm{T}}, \tag{13}$$

where $s = \bar{x} - x$ (the change in x).

In the local optimizer, H is required to be positive definite. If H is positive definite, $\bar{H}$ defined by (13) will be positive definite if and only if $y^{\mathrm{T}}s$ is positive (see Dennis and Moré (1977)). Ideally, y in (13) would be taken as y_L, the change in gradient of the Lagrangian function

$$y_L = \bar{g} - \bar{A}_N^{\mathrm{T}}\mu_N - g + A_N^{\mathrm{T}}\mu_N, \tag{14}$$

where μ_N denotes the QP multipliers associated with the nonlinear constraints of the original problem. If $y_L^{\mathrm{T}}s$ is not sufficiently positive, an attempt is made to perform the update with a vector y of the form

$$y = y_L + \sum_{i=1}^{m_N} \omega_i (a_i(\hat{x})c_i(\hat{x}) - a_i(x)c_i(x)),$$

where $\omega_i \geq 0$. If no such vector can be found, the update is performed with a scaled y_L; in this case, `M` is printed to indicate that the update was modified.

Rather than modifying H itself, the Cholesky factor of the *transformed Hessian* H_Q (6) is updated, where Q is the matrix from (5) associated with the active set of the QP subproblem. The update (13) is equivalent to the following update to H_Q:

$$\bar{H}_Q = H_Q - \frac{1}{s_Q^{\mathrm{T}}H_Q s_Q} H_Q s_Q s_Q^{\mathrm{T}} H_Q + \frac{1}{y_Q^{\mathrm{T}}s_Q} y_Q y_Q^{\mathrm{T}}, \tag{15}$$

where $y_Q = Q^{\mathrm{T}}y$, and $s_Q = Q^{\mathrm{T}}s$. This update may be expressed as a *rank-one* update to R (see Dennis and Schnabel (1981)).

12 Optional Arguments

Several optional arguments in nag_glopt_nlp_multistart_sqp (e05ucc) define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_glopt_nlp_multistart_sqp (e05ucc) these optional arguments have associated *default values* that are appropriate for most problems. Therefore you need only specify those optional arguments whose values are to be different from their default values.

The remainder of this section can be skipped if you wish to use the default values for all optional arguments. The following is a list of the optional arguments available and a full description of each optional argument is provided in Section 12.1.

Central Difference Interval
Crash Tolerance
Defaults
Derivative Level
Difference Interval
Feasibility Tolerance
Function Precision
Hessian
Infinite Bound Size
Infinite Step Size
Iteration Limit
Iters
Itns
Linear Feasibility Tolerance
Line Search Tolerance
List
Major Iteration Limit
Major Print Level
Minor Iteration Limit

Minor Print Level
Monitoring File
Nolist
Nonlinear Feasibility Tolerance
Optimality Tolerance
Out_Level
Print Level
Punch Unit
Start Constraint Check At Variable
Start Objective Check At Variable
Step Limit
Stop Constraint Check At Variable
Stop Objective Check At Variable
Verify
Verify Constraint Gradients
Verify Gradients
Verify Level
Verify Objective Gradients

Optional arguments may be specified by calling nag_glopt_opt_set (e05zkc) before a call to nag_glopt_nlp_multistart_sqp (e05ucc). Before calling nag_glopt_nlp_multistart_sqp (e05ucc), the optional argument arrays **iopts** and **opts** MUST be initialized for use with nag_glopt_nlp_multistart_sqp (e05ucc) by calling nag_glopt_opt_set (e05zkc) with **optstr** set to '`Initialize = e05ucc`'.

All optional arguments not specified are set to their default values. Optional arguments specified are unaltered by nag_glopt_nlp_multistart_sqp (e05ucc) (unless they define invalid values) and so remain in effect for subsequent calls to nag_glopt_nlp_multistart_sqp (e05ucc).

12.1 Description of the Optional Arguments

For each option, we give a summary line, a description of the optional argument and details of constraints.

The summary line contains:

the keywords, where the minimum abbreviation of each keyword is underlined (if no characters of an optional qualifier are underlined, the qualifier may be omitted)

a parameter value, where the letters a, i and r denote options that take character, integer and real values respectively

the default value, where the symbol ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)), and ϵ_r denotes the relative precision of the objective function **Function Precision**, and $bigbnd$ signifies the value of **Infinite Bound Size**

Keywords and character values are case insensitive, however they must be separated by at least one whitespace.

Optional arguments used to specify files have type Nag_FileID (see Section 3.2.1.1 in the Essential Introduction). This ID value must either be set to 0 (the default value) in which case there will be no output, or will be as returned by a call of nag_open_file (x04acc).

For nag_glopt_nlp_multistart_sqp (e05ucc) the maximum length of the argument **cvalue** used by nag_glopt_opt_get (e05zlc) is 11.

Central Difference Interval r Default values are computed

If the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate, the value of r is used as the difference interval for every element of x. The switch

to central differences is indicated by `C` at the end of each line of intermediate printout produced by the major iterations (see Section 9.1). The use of finite differences is discussed further under the optional argument **Difference Interval**.

If you supply a value for this optional argument, a small value between 0.0 and 1.0 is appropriate.

Crash Tolerance r Default $= 0.01$

This value is used when the local minimizer selects an initial working set. If $0 \leq r \leq 1$, 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 $a_j^{\mathrm{T}}x \geq l$ will be included in the initial working set if $\left|a_j^{\mathrm{T}}x - l\right| \leq r(1 + |l|)$. If $r < 0$ or $r > 1$, the default value is used.

Defaults

This special keyword is used to reset all optional arguments to their default values, and any random state stored in **state** will be destroyed.

Any option value given with this keyword will be ignored. This optional argument cannot be queried or got.

Derivative Level i Default $= 3$

This argument indicates which derivatives are provided in user-supplied functions **objfun** and **confun**. The possible choices for i are the following.

i	Meaning
3	All elements of the objective gradient and the constraint Jacobian are provided.
2	All elements of the constraint Jacobian are provided, but some elements of the objective gradient are not specified.
1	All elements of the objective gradient are provided, but some elements of the constraint Jacobian are not specified.
0	Some elements of both the objective gradient and the constraint Jacobian are not specified.

The value $i = 3$ should be used whenever possible, since nag_glopt_nlp_multistart_sqp (e05ucc) is more reliable (and will usually be more efficient) when all derivatives are exact.

If $i = 0$ or 2, nag_glopt_nlp_multistart_sqp (e05ucc) will estimate the unspecified elements of the objective gradient, using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to **objfun** is required for each unspecified element. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* (1981), for a discussion of limiting accuracy).

If $i = 0$ or 1, nag_glopt_nlp_multistart_sqp (e05ucc) will approximate unspecified elements of the constraint Jacobian. One call to **confun** is needed for each variable for which partial derivatives are not available. For example, if the Jacobian has the form

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

where '$*$' indicates an element provided by you and '?' indicates an unspecified element, the local minimizer will call **confun** twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to **confun**.)

At times, central differences are used rather than forward differences, in which case twice as many calls to **objfun** and **confun** are needed. (The switch to central differences is not under your control.)

If $i < 0$ or $i > 3$, the default value is used.

Difference Interval r Default values are computed

This option defines an interval used to estimate derivatives by finite differences in the following circumstances:

(a) For verifying the objective and/or constraint gradients (see the description of the optional argument **Verify**).

(b) For estimating unspecified elements of the objective gradient or the constraint Jacobian.

In general, a derivative with respect to the jth variable is approximated using the interval δ_j, where $\delta_j = r\left(1 + |\hat{x}_j|\right)$, with $\hat{x}$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to $O(r)$. See Gill *et al.* (1981) for a discussion of the accuracy in finite difference approximations.

If a difference interval is not specified, a finite difference interval will be computed automatically for each variable by a procedure that requires up to six calls of **confun** and **objfun** for each element. This option is recommended if the function is badly scaled or you wish to have the local minimizer determine constant elements in the objective and constraint gradients (see the descriptions of **confun** and **objfun** in Section 5).

If you supply a value for this optional argument, a small value between 0.0 and 1.0 is appropriate.

Feasibility Tolerance r Default $= \sqrt{\epsilon}$

The scalar r defines the maximum acceptable *absolute* violations in linear and nonlinear constraints at a 'feasible' point; i.e., a constraint is considered satisfied if its violation does not exceed r. If $r < \epsilon$ or $r \geq 1$, the default value is used. Using this keyword sets both optional arguments **Linear Feasibility Tolerance** and **Nonlinear Feasibility Tolerance** to r, if $\epsilon \leq r < 1$. (Additional details are given under the descriptions of these optional arguments.)

Function Precision r Default $= \epsilon^{0.9}$

This argument defines ϵ_r, which is intended to be a measure of the accuracy with which the problem functions $F(x)$ and $c(x)$ can be computed. If $r < \epsilon$ or $r \geq 1$, the default value is used.

The value of ϵ_r should reflect the relative precision of $1 + |F(x)|$; i.e., ϵ_r acts as a relative precision when $|F|$ is large, and as an absolute precision when $|F|$ is small. For example, if $F(x)$ is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for ϵ_r would be 10^{-6}. In contrast, if $F(x)$ is typically of order 10^{-4} and the first six significant digits are known to be correct, an appropriate value for ϵ_r would be 10^{-10}. The choice of ϵ_r can be quite complicated for badly scaled problems; see Chapter 8 of Gill *et al.* (1981) for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of ϵ_r should be large enough so that nag_glopt_nlp_multistart_sqp (e05ucc) will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Hessian a Default $=$ NO

This option controls the contents of the upper triangular matrix R (see Section 5). nag_glopt_nlp_multistart_sqp (e05ucc) works exclusively with the *transformed and reordered* Hessian H_Q (6), and hence extra computation is required to form the Hessian itself. If **Hessian** $=$ 'NO', **r** contains the Cholesky factor of the transformed and reordered Hessian. If **Hessian** $=$ 'YES', the Cholesky factor of the approximate Hessian itself is formed and stored in **r**.

Infinite Bound Size r Default $= 10^{20}$

This 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 ∞ (and similarly any lower bound less than or equal to $-infbnd$ will be regarded as $-\infty$).

Constraint: $r_{\max}^{\frac{1}{4}} \leq infbnd \leq r_{\max}^{\frac{1}{2}}$.

Infinite Step Size r Default $= \max\left(bigbnd, 10^{20}\right)$

If $r > 0$, r specifies the magnitude of the change in variables that is treated as a step to an unbounded solution. If the change in x during an iteration would exceed the value of r, the objective function is considered to be unbounded below in the feasible region. If $r \leq 0$, the default value is used.

Line Search Tolerance r Default $= 0.9$

The value r $(0 \leq r < 1)$ controls the accuracy with which the step α taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of r, the more accurate the linesearch). The default value $r = 0.9$ requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If there are no nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations – for example, if the objective function is cheap to evaluate, or if a substantial number of derivatives are unspecified. If $r < 0$ or $r \geq 1$, the default value is used.

Linear Feasibility Tolerance r_1 Default $= \sqrt{\epsilon}$
Nonlinear Feasibility Tolerance r_2 Default $= \epsilon^{0.33}$ or $\sqrt{\epsilon}$

The default value of r_2 is $\epsilon^{0.33}$ if **Derivative Level** $= 0$ or 1, and $\sqrt{\epsilon}$ otherwise.

The scalars r_1 and r_2 define the maximum acceptable *absolute* violations in linear and nonlinear constraints at a 'feasible' point; i.e., a linear constraint is considered satisfied if its violation does not exceed r_1, and similarly for a nonlinear constraint and r_2. If $r_m < \epsilon$ or $r_m \geq 1$, the default value is used, for $m = 1, 2$.

On entry to the local optimizer an iterative procedure is executed in order to find a point that satisfies the linear constraints and bounds on the variables to within the tolerance r_1. All subsequent iterates will satisfy the linear constraints to within the same tolerance (unless r_1 is comparable to the finite difference interval).

For nonlinear constraints, the feasibility tolerance r_2 defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, the value of optional argument **Nonlinear Feasibility Tolerance** acts as a partial termination criterion for the iterative sequence generated by the local minimizer (see the discussion of optional argument **Optimality Tolerance**).

These tolerances should reflect the precision of the corresponding constraints. For example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify r_1 as 10^{-6}.

List
Nolist Default

For nag_glopt_nlp_multistart_sqp (e05ucc), normally each optional argument specification is not printed as it is supplied. Optional argument **Nolist** may be used to suppress the printing and optional argument **List** may be used to turn on printing.

Major Iteration Limit i Default $= \max(50, 3(n + n_L) + 10n_N)$
Iteration Limit
Iters
Itns

The value of i specifies the maximum number of major iterations allowed before termination of each local subproblem. Setting $i = 0$ and **Major Print Level** > 0 means that the workspace needed by each local minimization will be computed and printed, but no iterations will be performed. If $i < 0$, the default value is used.

Major Print Level i Default $= 0$
Print Level i

The value of i controls the amount of printout produced by the major iterations of nag_glopt_nlp_multistart_sqp (e05ucc), as indicated below. A detailed description of the printed output

is given in Section 9.1 (summary output at each major iteration and the final solution) and Section 13 (monitoring information at each major iteration). (See also the description of the optional argument **Minor Print Level**.)

The following printout is sent to `stdout`:

i	**Output**
0	No output.

For the other values described below, the arguments used by the local minimizer are displayed in addition to intermediate and final output.

i	**Output**
1	The final solution only.
5	One line of summary output (< 80 characters; see Section 9.1) for each major iteration (no printout of the final solution).
≥ 10	The final solution and one line of summary output for each major iteration.

The following printout is sent to the file associated with the FileID defined by the optional argument **Monitoring File**:

i	**Output**
< 5	No output.
≥ 5	One long line of output (> 80 characters; see Section 13) for each major iteration (no printout of the final solution).
≥ 20	At each major iteration, the objective function, the Euclidean norm of the nonlinear constraint violations, the values of the nonlinear constraints (the vector c), the values of the linear constraints (the vector $A_L x$), and the current values of the variables (the vector x).
≥ 30	At each major iteration, the diagonal elements of the matrix T associated with the TQ factorization (5) (see Section 11.1) of the QP working set, and the diagonal elements of R, the triangular factor of the transformed and reordered Hessian (6) (see Section 11.1).

Minor Iteration Limit i Default $= \max(50, 3(n + n_L + n_N))$

The value of i specifies the maximum number of iterations for finding a feasible point with respect to the bounds and linear constraints (if any). The value of i also specifies the maximum number of minor iterations for the optimality phase of each QP subproblem. If $i \leq 0$, the default value is used.

Minor Print Level i Default $= 0$

The value of i controls the amount of printout produced by the minor iterations of nag_glopt_nlp_multistart_sqp (e05ucc) (i.e., the iterations of the quadratic programming algorithm), as indicated below. A detailed description of the printed output is given in Section 9.1 (summary output at each minor iteration and the final QP solution) and Section 13 (monitoring information at each minor iteration). (See also the description of the optional argument **Major Print Level**.) The following printout is sent to `stdout`:

i	**Output**
0	No output.
1	The final QP solution only.
5	One line of summary output (< 80 characters; see Section 9.1) for each minor iteration (no printout of the final QP solution).
≥ 10	The final QP solution and one line of summary output for each minor iteration.

The following printout is sent to the file associated with the FileID defined by the optional argument **Monitoring File**:

i	Output
< 5	No output.
≥ 5	One long line of output (> 80 characters; see Section 9.1) for each minor iteration (no printout of the final QP solution).
≥ 20	At each minor iteration, the current estimates of the QP multipliers, the current estimate of the QP search direction, the QP constraint values, and the status of each QP constraint.
≥ 30	At each minor iteration, the diagonal elements of the matrix T associated with the TQ factorization (5) (see Section 11.1) of the QP working set, and the diagonal elements of the Cholesky factor R of the transformed Hessian (6) (see Section 11.1).

Monitoring File Default $= -1$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

i is of the type Nag_FileID and is obtained by a call to nag_open_file (x04acc).

If $i \geq 0$ and **Major Print Level** ≥ 5 or $i \geq 0$ and **Minor Print Level** ≥ 5, monitoring information produced by nag_glopt_nlp_multistart_sqp (e05ucc) at every iteration is sent to a file with ID i. If $i < 0$ and/or **Major Print Level** < 5 and **Minor Print Level** < 5, no monitoring information is produced.

Optimality Tolerance r Default $= \epsilon_R^{0.8}$

The argument r ($\epsilon_R \leq r < 1$) specifies the accuracy to which you wish the final iterate to approximate a solution of each local problem. Broadly speaking, r indicates the number of correct figures desired in the objective function at the solution. For example, if r is 10^{-6} and a local minimization terminates successfully, the final value of F should have approximately six correct figures. If $r < \epsilon_r$ or $r \geq 1$, the default value is used.

The local optimizer will terminate successfully if the iterative sequence of x values is judged to have converged and the final point satisfies the first-order Kuhn–Tucker conditions (see Section 11.1). The sequence of iterates is considered to have converged at x if

$$\alpha\|p\| \leq \sqrt{r}(1 + \|x\|), \tag{16}$$

where p is the search direction and α the step length from (3). An iterate is considered to satisfy the first-order conditions for a minimum if

$$\left\|Z^{\mathrm{T}} g_{\mathrm{FR}}\right\| \leq \sqrt{r}(1 + \max(1 + |F(x)|, \|g_{\mathrm{FR}}\|)) \tag{17}$$

and

$$\left|res_j\right| \leq ftol \quad \text{for all} \quad j, \tag{18}$$

where $Z^{\mathrm{T}} g_{\mathrm{FR}}$ is the projected gradient (see Section 11.1), g_{FR} is the gradient of $F(x)$ with respect to the free variables, res_j is the violation of the jth active nonlinear constraint, and $ftol$ is the **Nonlinear Feasibility Tolerance**.

Out_Level i Default $= 0$

This option defines the amount of extra information to be sent to the Fortran unit number defined by **Punch Unit**. The possible choices for i are the following:

i	Meaning
0	No extra output.
1	Updated solutions only. This is useful during long runs to observe progress.
2	Successful start points only. This is useful to save the starting points that gave rise to the final solution.

3 Both updated solutions and successful start points.

Punch Unit i Default $= 6$

This option allows you to send information arising from an appropriate setting of **Out_Level** to be sent to the Fortran unit number defined by **Punch Unit**. If you wish this file to be different to the standard output unit (6) where other output is displayed then this file should be attached by calling nag_open_file (x04acc) prior to calling nag_glopt_nlp_multistart_sqp (e05ucc).

Start Objective Check At Variable	i_1	Default $= 1$
Stop Objective Check At Variable	i_2	Default $= n$
Start Constraint Check At Variable	i_3	Default $= 1$
Stop Constraint Check At Variable	i_4	Default $= n$

These keywords take effect only if **Verify Level** > 0. They may be used to control the verification of gradient elements computed by **objfun** and/or Jacobian elements computed by **confun**. For example, if the first 30 elements of the objective gradient appeared to be correct in an earlier run, so that only element 31 remains questionable, it is reasonable to specify **Start Objective Check At Variable** $= 31$. If the first 30 variables appear linearly in the objective, so that the corresponding gradient elements are constant, the above choice would also be appropriate.

If $i_{2m-1} \leq 0$ or $i_{2m-1} > \min(n, i_{2m})$, the default value is used, for $m = 1, 2$. If $i_{2m} \leq 0$ or $i_{2m} > n$, the default value is used, for $m = 1, 2$.

Step Limit r Default $= 2.0$

If $r > 0, r$ specifies the maximum change in variables at the first step of the linesearch. In some cases, such as $F(x) = ae^{bx}$ or $F(x) = ax^b$, even a moderate change in the elements of x can lead to floating-point overflow. The argument r is therefore used to encourage evaluation of the problem functions at meaningful points. Given any major iterate x, the first point $\tilde{x}$ at which F and c are evaluated during the linesearch is restricted so that

$$\|\tilde{x} - x\|_2 \leq r\big(1 + \|x\|_2\big).$$

The linesearch may go on and evaluate F and c at points further from x if this will result in a lower value of the merit function (indicated by `L` at the end of each line of output produced by the major iterations; see Section 9.1). If `L` is printed for most of the iterations, r should be set to a larger value.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at wild values. The default value **Step Limit** $= 2.0$ should not affect progress on well-behaved functions, but values such as 0.1 or 0.01 may be helpful when rapidly varying functions are present. If a small value of **Step Limit** is selected, a good starting point may be required. An important application is to the class of nonlinear least squares problems. If $r \leq 0$, the default value is used.

Verify Level	i	Default $= 0$
Verify	i	
Verify Constraint Gradients	i	
Verify Gradients	i	
Verify Objective Gradients	i	

These keywords refer to finite difference checks on the gradient elements computed by **objfun** and **confun**. The possible choices for i are as follows:

i	**Meaning**
-1	No checks are performed.
0	Only a 'cheap' test will be performed.
≥ 1	Individual gradient elements will also be checked using a reliable (but more expensive) test.

It is possible to specify **Verify Level** $= 0$ to 3 in several ways. For example, the nonlinear objective gradient (if any) will be verified if either **Verify Objective Gradients** or **Verify Level** $= 1$ is specified. The constraint gradients will be verified if **Verify** $=$ 'YES' or **Verify Level** $= 2$ or **Verify** is specified.

Similarly, the objective and the constraint gradients will be verified if **Verify** = 'YES' or **Verify Level** = 3 or **Verify** is specified.

If $0 \leq i \leq 3$, gradients will be verified at the first point that satisfies the linear constraints and bounds.

If $i = 0$, only a 'cheap' test will be performed, requiring one call to **objfun** and (if appropriate) one call to **confun**.

If $1 \leq i \leq 3$, a more reliable (but more expensive) check will be made on individual gradient elements, within the ranges specified by the **Start Constraint Check At Variable** and **Stop Constraint Check At Variable** keywords. A result of the form `OK` or `BAD?` is printed by nag_glopt_nlp_multistart_sqp (e05ucc) to indicate whether or not each element appears to be correct.

If $10 \leq i \leq 13$, the action is the same as for $i - 10$, except that it will take place at the user-specified initial value of x.

If $i < -1$ or $4 \leq i \leq 9$ or $i > 13$, the default value is used.

We suggest that **Verify Level** = 3 be used whenever a new function function is being developed.

13 Description of Monitoring Information

This section describes the long line of output (> 80 characters) which forms part of the monitoring information produced by nag_glopt_nlp_multistart_sqp (e05ucc). (See also the description of the optional arguments **Major Print Level**, **Minor Print Level** and **Monitoring File**.) You can control the level of printed output.

When **Major Print Level** ≥ 5 and **Monitoring File** ≥ 0, the following line of output is produced at every major iteration of nag_glopt_nlp_multistart_sqp (e05ucc) on the file specified by **Monitoring File**. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` is the major iteration count.

`Mnr` is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 11). Note that `Mnr` may be greater than the optional argument **Minor Iteration Limit** if some iterations are required for the feasibility phase.

`Step` is the step α_k taken along the computed search direction. On reasonably well-behaved local problems, the unit step (i.e., $\alpha_k = 1$) will be taken as the solution is approached.

`Nfun` is the cumulative number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the gradients by finite differences are not included. `Nfun` is printed as a guide to the amount of work required for the linesearch.

`Merit Function` is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty arguments (see Section 11.3). As the solution is approached, `Merit Function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line) then the merit function is a large multiple of the constraint violations, weighted by the penalty arguments. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or the local optimizer terminates. Repeated failures will prevent a feasible point being found for the nonlinear constraints.

If there are no nonlinear constraints present (i.e., **ncnln** = 0) then this entry contains `Objective`, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

`Norm Gz` is $\|Z^{\mathrm{T}}g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 11.2). `Norm Gz` will be approximately zero in the neighbourhood of a solution.

`Violtn` is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if **ncnln** is zero). `Violtn` will be approximately zero in the neighbourhood of a solution.

`Nz` is the number of columns of Z (see Section 11.2). The value of `Nz` is the number of variables minus the number of constraints in the predicted active set; i.e., $\mathtt{Nz} = n - (\mathtt{Bnd} + \mathtt{Lin} + \mathtt{Nln})$.

`Bnd` is the number of simple bound constraints in the predicted active set.

`Lin` is the number of general linear constraints in the predicted working set.

`Nln` is the number of nonlinear constraints in the predicted active set (not printed if **ncnln** is zero).

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

`Cond H` is a lower bound on the condition number of the Hessian approximation H.

`Cond Hz` is a lower bound on the condition number of the projected Hessian approximation H_Z ($H_Z = Z^{\mathrm{T}}H_{\mathrm{FR}}Z = R_Z^{\mathrm{T}}R_Z$; see (6)). The larger this number, the more difficult the local problem.

`Cond T` is a lower bound on the condition number of the matrix of predicted active constraints.

`Conv` is a three-letter indication of the status of the three convergence tests (16)–(18) defined in the description of the optional argument **Optimality Tolerance**. Each letter is `T` if the test is satisfied and `F` otherwise. The three tests indicate whether:

(i) the sequence of iterates has converged;

(ii) the projected gradient (`Norm Gz`) is sufficiently small; and

(iii) the norm of the residuals of constraints in the predicted active set (`Violtn`) is small enough.

If any of these indicators is `F` for a successful local minimization you should check the solution carefully.

`M` is printed if the quasi-Newton update has been modified to ensure that the Hessian approximation is positive definite (see Section 11.4).

`I` is printed if the QP subproblem has no feasible point.

`C` is printed if central differences have been used to compute the unspecified objective and constraint gradients. If the value of `Step` is zero then the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is resolved with the central difference gradient and Jacobian.) If the value of `Step` is nonzero then central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 11.1).

`L` is printed if the linesearch has produced a relative change in x greater than the value defined by the optional argument **Step Limit**. If this output occurs frequently during later iterations of the run, optional argument **Step Limit** should be set to a larger value.

`R` is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned then the approximate Hessian is refactorized using column

interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.