NAG Library Function Document

nag_opt_bounds_2nd_deriv (e04lbc)

1 Purpose

nag_opt_bounds_2nd_deriv (e04lbc) is a comprehensive modified-Newton algorithm for finding:

- an unconstrained minimum of a function of several variables
- a minimum of a function of several variables subject to fixed upper and/or lower bounds on the variables.

First and second derivatives are required. nag_opt_bounds_2nd_deriv (e04lbc) is intended for objective functions which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2 Specification

```c
#include <nag.h>
#include <nage04.h>

void nag_opt_bounds_2nd_deriv (Integer n,
    void (*objfun)(Integer n, const double x[], double *objf, double g[],
        Nag_Comm *comm),
    void (*hessfun)(Integer n, const double x[], double h[], double hd[],
        Nag_Comm *comm),
    Nag_BoundType bound, double bl[], double bu[], double x[], double *objf,
    double g[], Nag_E04_Opt *options, Nag_Comm *comm, NagError *fail)
```

3 Description

nag_opt_bounds_2nd_deriv (e04lbc) is applicable to problems of the form:

\[
\text{Minimize } F(x_1, x_2, \ldots, x_n) \\
\text{subject to } l_j \leq x_j \leq u_j, \quad j = 1, 2, \ldots, n.
\]

Special provision is made for unconstrained minimization (i.e., problems which actually have no bounds on the \( x_j \)), problems which have only non-negativity bounds, and problems in which \( l_1 = l_2 = \cdots = l_n \) and \( u_1 = u_2 = \cdots = u_n \). It is possible to specify that a particular \( x_j \) should be held constant. You must supply a starting point, a function \textbf{objfun} to calculate the value of \( F(x) \) and its first derivatives \( \frac{\partial F}{\partial x_j} \) at any point \( x \), and a function \textbf{hessfun} to calculate the second derivatives \( \frac{\partial^2 F}{\partial x_i \partial x_j} \).

A typical iteration starts at the current point \( x \) where \( n_z \) (say) variables are free from both their bounds. The vector of first derivatives of \( F(x) \) with respect to the free variables, \( g_z \), and the matrix of second derivatives with respect to the free variables, \( H \), are obtained. (These both have dimension \( n_z \).) The equations

\[(H + E)p_z = -g_z\]

are solved to give a search direction \( p_z \). (The matrix \( E \) is chosen so that \( H + E \) is positive definite.) \( p_z \) is then expanded to an \( n \)-vector \( p \) by the insertion of appropriate zero elements; \( \alpha \) is found such that \( F(x + \alpha p) \) is approximately a minimum (subject to the fixed bounds) with respect to \( \alpha \), and \( x \) is replaced by \( x + \alpha p \). (If a saddle point is found, a special search is carried out so as to move away from the saddle point.) If any variable actually reaches a bound, it is fixed and \( n_z \) is reduced for the next iteration.
There are two sets of convergence criteria – a weaker and a stronger. Whenever the weaker criteria are satisfied, the Lagrange-multipliers are estimated for all active constraints. If any Lagrange-multiplier estimate is significantly negative, then one of the variables associated with a negative Lagrange-multiplier estimate is released from its bound and the next search direction is computed in the extended subspace (i.e., \( n_z \) is increased). Otherwise, minimization continues in the current subspace until the stronger criteria are satisfied. If at this point there are no negative or near-zero Lagrange-multiplier estimates, the process is terminated.

If you specify that the problem is unconstrained, \texttt{nag_opt_bounds_2nd_deriv} (e04lbc) sets the \( l_j \) to \(-10^{10}\) and the \( u_j \) to \(10^{10}\). Thus, provided that the problem has been sensibly scaled, no bounds will be encountered during the minimization process and \texttt{nag_opt_bounds_2nd_deriv} (e04lbc) will act as an unconstrained minimization algorithm.

4 References

Gill P E and Murray W (1973) Safeguarded steplength algorithms for optimization using descent methods \textit{NPL Report NAC 37} National Physical Laboratory


Gill P E and Murray W (1976) Minimization subject to bounds on the variables \textit{NPL Report NAC 72} National Physical Laboratory

5 Arguments

1: \( n \) – Integer \hspace{1cm} \textit{Input}

\textit{On entry:} the number \( n \) of independent variables.

\textit{Constraint:} \( n \geq 1 \).

2: \texttt{objfun} – function, supplied by the user \hspace{1cm} \textit{External Function}

\texttt{objfun} must evaluate the function \( F(x) \) and its first derivatives \( \frac{\partial F}{\partial x_j} \) at any point \( x \). (However, if you do not wish to calculate \( F(x) \) or its first derivatives at a particular \( x \), there is the option of setting an argument to cause \texttt{nag_opt_bounds_2nd_deriv} (e04lbc) to terminate immediately.)

The specification of \texttt{objfun} is:

\begin{verbatim}
void objfun (Integer n, const double x[], double *objf, double g[],
            Nag_Comm *comm)
1: \( n \) – Integer \hspace{1cm} \textit{Input}
    \textit{On entry:} \( n \), the number of variables.

2: \( x[n] \) – const double \hspace{1cm} \textit{Input}
    \textit{On entry:} the point \( x \) at which the value of \( F \), or \( F \) and \( \frac{\partial F}{\partial x_j} \), are required.

3: \texttt{objf} – double * \hspace{1cm} \textit{Output}
    \textit{On exit:} \texttt{objf} must set \texttt{objf} to the value of the objective function \( F \) at the current point \( x \). If it is not possible to evaluate \( F \) then \texttt{objf} should assign a negative value to \texttt{comm->flag}; \texttt{nag_opt_bounds_2nd_deriv} (e04lbc) will then terminate.
\end{verbatim}
On exit: **objfun** must set `g[j – 1]` to the value of the first derivative \( \frac{\partial F}{\partial x_j} \) at the current point \( x \), for \( j = 1, 2, \ldots, n \). If it is not possible to evaluate the first derivatives then **objfun** should assign a negative value to **comm**→flag; **nag_opt_bounds_2nd_deriv** (e04lbc) will then terminate.

5: **comm** – `Nag_Comm *`

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

- **flag** – `Integer`  
  On exit: if **objfun** resets **comm**→flag to some negative number then **nag_opt_bounds_2nd_deriv** (e04lbc) will terminate immediately with the error indicator NE_USER_STOP. If **fail** is supplied to **nag_opt_bounds_2nd_deriv** (e04lbc), **fail**.errnum will be set to your setting of **comm**→flag.

- **first** – `Nag_Boolean`
  On entry: will be set to **Nag_TRUE** on the first call to **objfun** and **Nag_FALSE** for all subsequent calls.

- **nf** – `Integer`  
  On entry: the number of evaluations of the objective function; this value will be equal to the number of calls made to **objfun** (including the current one).

- **user** – `double *`
- **iuser** – `Integer *`
- **p** – Pointer

The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

Before calling **nag_opt_bounds_2nd_deriv** (e04lbc) these pointers may be allocated memory and initialized with various quantities for use by **objfun** when called from **nag_opt_bounds_2nd_deriv** (e04lbc).

**Note:** **objfun** should be tested separately before being used in conjunction with **nag_opt_bounds_2nd_deriv** (e04lbc). The array \( x \) must not be changed by **objfun**.

3: **hessfun** – function, supplied by the user  

*External Function*

**hessfun** must calculate the second derivatives of \( F(x) \) at any point \( x \). (As with **objfun** there is the option of causing **nag_opt_bounds_2nd_deriv** (e04lbc) to terminate immediately.)

The specification of **hessfun** is:

```c
void hessfun (Integer n, const double x[], double h[], double hd[],
              Nag_Comm *comm)
```

1: **n** – `Integer`  
  On entry: the number \( n \) of variables.

2: **x[n]** – `const double`
  On entry: the point \( x \) at which the second derivatives of \( F \) are required.

3: **h[n × (n – 1)/2]** – `double`
  On exit: **hessfun** must place the strict lower triangle of the second derivative matrix of \( F \) (evaluated at the point \( x \)) in \( h \), stored by rows, i.e., set
\[
\mathbf{h}[i-1](i-2)/2 + j-1 = \frac{\partial^2 F}{\partial x_i \partial x_j} \mid x, \quad \text{for } i = 2, 3, \ldots, n \text{ and } j = 1, 2, \ldots, i-1.
\]

(The upper triangle is not required because the matrix is symmetric.) If it is not possible to evaluate the elements of \( \mathbf{h} \) then \textbf{hessfun} should assign a negative value to \textbf{comm}→\textbf{flag}; \textbf{nag_opt_bounds_2nd_deriv} (e04lbc) will then terminate.

4: \( \mathbf{hd}[n] \) – double

\textit{Input/Output}

\textit{On entry:} the value of \( \frac{\partial F}{\partial x_j} \) at the point \( x \), for \( j = 1, 2, \ldots, n \). These values may be useful in the evaluation of the second derivatives.

\textit{On exit:} unless \textbf{comm}→\textbf{flag} is reset to a negative number \textbf{hessfun} must place the diagonal elements of the second derivative matrix of \( F \) (evaluated at the point \( x \)) in \( \mathbf{hd} \), i.e., set

\[
\mathbf{hd}[j-1] = \frac{\partial^2 F}{\partial x_j^2}\mid x, \quad \text{for } j = 1, 2, \ldots, n.
\]

If it is not possible to evaluate the elements of \( \mathbf{hd} \) then \textbf{hessfun} should assign a negative value to \textbf{comm}→\textbf{flag}; \textbf{nag_opt_bounds_2nd_deriv} (e04lbc) will then terminate.

5: \textbf{comm} – 

\textit{Pointer to structure of type Nag_Comm; the following members are relevant to \textbf{objfun}.}

\textbf{flag} – Integer

\textit{Output}

\textit{On exit:} if \textbf{hessfun} resets \textbf{comm}→\textbf{flag} to some negative number then \textbf{nag_opt_bounds_2nd_deriv} (e04lbc) will terminate immediately with the error indicator \texttt{NE_USER_STOP}. If \textbf{fail} is supplied to \textbf{nag_opt_bounds_2nd_deriv} (e04lbc) \textbf{fail.errnum} will be set to your setting of \textbf{comm}→\textbf{flag}.

\textbf{first} – Nag_Boolean

\textit{Input}

\textit{On entry:} will be set to \texttt{Nag_TRUE} on the first call to \textbf{hessfun} and \texttt{Nag_FALSE} for all subsequent calls.

\textbf{nf} – Integer

\textit{Input}

\textit{On entry:} the number of calculations of the objective function; this value will be equal to the number of calls made to \textbf{hessfun} including the current one.

\textbf{user} – double *

\textbf{iuser} – Integer *

\textbf{p} – Pointer

The type Pointer will be \texttt{void *} with a C compiler that defines \texttt{void *} and \texttt{char *} otherwise.

Before calling \textbf{nag_opt_bounds_2nd_deriv} (e04lbc) these pointers may be allocated memory and initialized with various quantities for use by \textbf{hessfun} when called from \textbf{nag_opt_bounds_2nd_deriv} (e04lbc).

\textbf{Note:} \textbf{hessfun} should be tested separately before being used in conjunction with \textbf{nag_opt_bounds_2nd_deriv} (e04lbc). The array \( x \) must \textbf{not} be changed by \textbf{hessfun}.

4: \textbf{bound} – Nag_BoundType

\textit{Input}

\textit{On entry:} indicates whether the problem is unconstrained or bounded and, if it is bounded, whether the facility for dealing with bounds of special forms is to be used. \textbf{bound} should be set to one of the following values:

\textbf{bound} = Nag_Bounds

If the variables are bounded and you will be supplying all the \( l_j \) and \( u_j \) individually.
bound = Nag_NoBounds
    If the problem is unconstrained.

bound = Nag_BoundsZero
    If the variables are bounded, but all the bounds are of the form $0 \leq x_j$.

bound = Nag_BoundsEqual
    If all the variables are bounded, and $l_1 = l_2 = \cdots = l_n$ and $u_1 = u_2 = \cdots = u_n$.

Constraint: bound = Nag_Bounds, Nag_NoBounds, Nag_BoundsZero or Nag_BoundsEqual.

5: bl[n] – double
    Input/Output

On entry: the lower bounds $l_j$.

If bound = Nag_Bounds, you must set $bl[j-1]$ to $l_j$, for $j=1,2,\ldots,n$. (If a lower bound is not required for any $x_j$, the corresponding $bl[j-1]$ should be set to a large negative number, e.g., $-10^{10}$.)

If bound = Nag_BoundsEqual, you must set $bl[0]$ to $l_1$; nag_opt_bounds_2nd_deriv (e04lbc) will then set the remaining elements of $bl$ equal to $bl[0]$.

If bound = Nag_NoBounds or Nag_BoundsZero, $bl$ will be initialized by nag_opt_bounds_2nd_deriv (e04lbc).

On exit: the lower bounds actually used by nag_opt_bounds_2nd_deriv (e04lbc), e.g., if bound = Nag_BoundsZero, $bl[0] = bl[1] = \cdots = bl[n-1] = 0.0$.

6: bu[n] – double
    Input/Output

On entry: the upper bounds $u_j$.

If bound = Nag_Bounds, you must set $bu[j-1]$ to $u_j$, for $j=1,2,\ldots,n$. (If an upper bound is not required for any $x_j$, the corresponding $bu[j-1]$ should be set to a large positive number, e.g., $10^{10}$.)

If bound = Nag_BoundsEqual, you must set $bu[0]$ to $u_1$; nag_opt_bounds_2nd_deriv (e04lbc) will then set the remaining elements of $bu$ equal to $bu[0]$.

If bound = Nag_NoBounds or Nag_BoundsZero, $bu$ will be initialized by nag_opt_bounds_2nd_deriv (e04lbc).

On exit: the upper bounds actually used by nag_opt_bounds_2nd_deriv (e04lbc), e.g., if bound = Nag_BoundsZero, $bu[0] = bu[1] = \cdots = bu[n-1] = 10^{10}$.

7: x[n] – double
    Input/Output

On entry: $x[j-1]$ must be set to a guess at the $j$th component of the position of the minimum, for $j=1,2,\ldots,n$.

On exit: the final point $x^*$. Thus, if fail.code = NE_NOERROR on exit, $x[j-1]$ is the $j$th component of the estimated position of the minimum.

8: objf – double *
    Output

On exit: the function value at the final point given in $x$.

9: g[n] – double
    Output

On exit: the first derivative vector corresponding to the final point in $x$. The elements of $g$ corresponding to free variables should normally be close to zero.

10: options – Nag_E04_Opt *
    Input/Output

On entry/exit: a pointer to a structure of type Nag_E04_Opt whose members are optional arguments for nag_opt_bounds_2nd_deriv (e04lbc). These structure members offer the means of
adjusting some of the argument values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 11.

If any of these optional arguments are required then the structure **options** should be declared and initialized by a call to `nag_opt_init (e04xxc)` and supplied as an argument to `nag_opt_bounds_2nd_deriv (e04lbc)`. However, if the optional arguments are not required the NAG defined null pointer, `E04_DEFAULT`, can be used in the function call.

11:  **comm** – `Nag_Comm *`  

   **Note:** **comm** is a NAG defined type (see Section 3.2.1.1 in the Essential Introduction).

   *On entry/exit:* structure containing pointers for communication to user-supplied functions; see the description of `objfun` and `hessfun` for details. If you do not need to make use of this communication feature the null pointer `E04COMM_NULL` may be used in the call to `nag_opt_bounds_2nd_deriv (e04lbc)`; **comm** will then be declared internally for use in calls to user-supplied functions.

12:  **fail** – `NagError *`  

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

5.1 **Description of Printed Output**

Intermediate and final results are printed out by default. The level of printed output can be controlled with the structure member **options.print_level** (see Section 11.2). The default, `options.print_level = Nag_Soln_Iter` provides a single line of output at each iteration and the final result. This section describes the default printout produced by `nag_opt_bounds_2nd_deriv (e04lbc)`.

The following line of output is produced at each iteration. In all cases the values of the quantities printed are those in effect on completion of the given iteration.

- **Itn** the iteration count, \( k \).
- **Nfun** the cumulative number of calls made to `objfun`.
- **Objective** the value of the objective function, \( F(x(k)) \).
- **Norm g** the Euclidean norm of the projected gradient vector, \( \|g_L(x(k))\| \).
- **Norm x** the Euclidean norm of \( x(k) \).
- **Norm(x(k-1)-x(k))** the Euclidean norm of \( x(k-1) - x(k) \).
- **Step** the step \( \alpha(k) \) taken along the computed search direction \( p(k) \).
- **Cond H** the ratio of the largest to the smallest element of the diagonal factor \( D \) of the projected Hessian matrix. This quantity is usually a good estimate of the condition number of the projected Hessian matrix. (If no variables are currently free, this value will be zero.)
- **PosDef** indicates whether the second derivative matrix \( H \) for the current subspace is positive definite (**Yes**) or not (**No**).

The printout of the final result consists of:

- **x** the final point, \( x^* \).
- **g** the final projected gradient vector, \( g_L(x^*) \).
- **Status** the final state of the variable with respect to its bound(s).
6 Error Indicators and Warnings

When one of NE_USER_STOP, NE_INT_ARG_LT, NE_BOUND, NE_DERIV_ERRORS, NE_OPT_NOT_INIT, NE_BAD_PARAM, NE_2_REAL_ARG_LT, NE_INVALID_INT_RANGE_1, NE_INVALID_REAL_RANGE_EF, NE_INVALID_REAL_RANGE_FF and NE_ALLOC_FAIL occurs, no values will have been assigned by nag_opt_bounds_2nd_deriv (e04lbc) to objf or to the elements of g, options.state, options.hesl, or options.hesd.

An exit of fail.code = NW_TOO_MANY_ITER, NW_LAGRANGE_MULT_ZERO and NW_COND_MIN may also be caused by mistakes in objfun, by the formulation of the problem or by an awkward function. If there are no such mistakes, it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure.

**NE_2_REAL_ARG_LT**

On entry, options.step_max = \( \langle \text{value} \rangle \) while options.optim_tol = \( \langle \text{value} \rangle \). These arguments must satisfy \( \text{options.step_max} \geq \text{options.optim_tol} \).

**NE_ALLOC_FAIL**

Dynamic memory allocation failed.

**NE_BAD_PARAM**

On entry, argument bound had an illegal value.

On entry, argument options.print_level had an illegal value.

**NE_BOUND**

The lower bound for variable \( \langle \text{value} \rangle \) (array element bl[\langle \text{value} \rangle]) is greater than the upper bound.

**NE_DERIV_ERRORS**

Large errors were found in the derivatives of the objective function.

**NE_INT_ARG_LT**

On entry, \( n \) must not be less than 1: \( n = \langle \text{value} \rangle \).

**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_INVALID_INT_RANGE_1**

Value \( \langle \text{value} \rangle \) given to options.max_iter is not valid. Correct range is \( \text{options.max_iter} \geq 0 \).

**NE_INVALID_REAL_RANGE_EF**

Value \( \langle \text{value} \rangle \) given to options.optim_tol is not valid. Correct range is \( \epsilon \leq \text{options.optim_tol} < 1.0 \).

**NE_INVALID_REAL_RANGE_FF**

Value \( \langle \text{value} \rangle \) given to options.linesearch_tol is not valid. Correct range is \( 0.0 \leq \text{options.linesearch_tol} < 1.0 \).

**NE_NOT_APPEND_FILE**

Cannot open file \( \langle \text{string} \rangle \) for appending.

**NE_NOT_CLOSE_FILE**

Cannot close file \( \langle \text{string} \rangle \).
Options structure not initialized.

User requested termination, user flag value = (value).

This exit occurs if you set comm → flag to a negative value in objfun or hessfun. If fail is supplied, the value of fail.errnum will be the same as your setting of comm → flag.

Error occurred when writing to file (string).

NE_COND_MIN

The conditions for a minimum have not all been satisfied, but a lower point could not be found.

Provided that, on exit, the first derivatives of $F(x)$ with respect to the free variables are sufficiently small, and that the estimated condition number of the second derivative matrix is not too large, this error exit may simply mean that, although it has not been possible to satisfy the specified requirements, the algorithm has in fact found the minimum as far as the accuracy of the machine permits. This could be because options.optim_tol has been set so small that rounding error in objfun makes attainment of the convergence conditions impossible.

If the estimated condition number of the second derivative matrix at the final point is large, it could be that the final point is a minimum but that the smallest eigenvalue of the second derivative matrix is so close to zero that it is not possible to recognize the point as a minimum.

All the Lagrange-multiplier estimates which are not indisputably positive lie close to zero.

However, it is impossible either to continue minimizing on the current subspace or to find a feasible lower point by releasing and perturbing any of the fixed variables. You should investigate as for NW_COND_MIN.

The maximum number of iterations, (value), have been performed.

If steady reductions in $F(x)$, were monitored up to the point where this exit occurred, then the exit probably occurred simply because options.max_iter was set too small, so the calculations should be restarted from the final point held in $x$. This exit may also indicate that $F(x)$ has no minimum.

A successful exit (fail.code = NE_NOERROR) is made from nag_opt_bounds_2nd_deriv (e04lbc) when $H^{(k)}$ is positive definite and when (B1, B2 and B3) or B4 hold, where

$B1 \equiv \alpha^{(k)} \times \| p^{(k)} \| < (\text{options.optim_tol} + \sqrt{\epsilon}) \times \left(1.0 + \| x^{(k)} \| \right)$

$B2 \equiv \left| F^{(k)} - F^{(k-1)} \right| < (\text{options.optim_tol}^2 + \epsilon) \times \left(1.0 + \left| F^{(k)} \right| \right)$

$B3 \equiv \| g^{(k)} \| < \left( \epsilon^{1/3} + \text{options.optim_tol} \right) \times \left(1.0 + \left| F^{(k)} \right| \right)$

$B4 \equiv \| g^{(k)} \| < 0.01 \times \sqrt{\epsilon}.$

(Quantities with superscript $k$ are the values at the $k$th iteration of the quantities mentioned in Section 3; $\epsilon$ is the machine precision, denotes the Euclidean norm and options.optim_tol is described in Section 11.)

If fail.code = NE_NOERROR, then the vector in $x$ on exit, $x_{\text{sol}}$, is almost certainly an estimate of the position of the minimum, $x_{\text{true}}$, to the accuracy specified by options.optim_tol.
If \texttt{fail.code} = NW_COND_MIN or NW_LAGRANGE_MULT_ZERO, \(x_{\text{sol}}\) may still be a good estimate of \(x_{\text{true}}\), but the following checks should be made. Let the largest of the first \(n_z\) elements of the optional argument \texttt{options.hesd} be \texttt{options.hesd}[b], let the smallest be \texttt{options.hesd}[s], and define \(\kappa = \texttt{options.hesd}[b]/\texttt{options.hesd}[s]\). The scalar \(\kappa\) is usually a good estimate of the condition number of the projected Hessian matrix at \(x_{\text{sol}}\). If

(a) the sequence \(\{F(x^{(k)})\}\) converges to \(F(x_{\text{sol}})\) at a superlinear or fast linear rate,
(b) \(\|g_z(x_{\text{sol}})\|^2 < 10.0 \times \epsilon\), and
(c) \(\kappa < 1.0/\|g_z(x_{\text{sol}})\|\),

then it is almost certain that \(x_{\text{sol}}\) is a close approximation to the position of a minimum. When (b) is true, then usually \(F(x_{\text{sol}})\) is a close approximation to \(F(x_{\text{true}})\). The quantities needed for these checks are all available in the results printout from \texttt{nag_opt_bounds_2nd_deriv (e04lbc)}; in particular the final value of Cond H gives \(\kappa\).

Further suggestions about confirmation of a computed solution are given in the e04 Chapter Introduction.

8 Parallelism and Performance

Not applicable.

9 Further Comments

9.1 Timing

The number of iterations required depends on the number of variables, the behaviour of \(F(x)\), the accuracy demanded and the distance of the starting point from the solution. The number of multiplications performed in an iteration of \texttt{nag_opt_bounds_2nd_deriv (e04lbc)} is \(n_z^3/6 + O(n_z^2)\). In addition, each iteration makes one call of \texttt{hessfun} and at least one call of \texttt{objfun}. So, unless \(F(x)\) and its derivatives can be evaluated very quickly, the run time will be dominated by the time spent in \texttt{objfun}.

9.2 Scaling

Ideally, the problem should be scaled so that, at the solution, \(F(x)\) and the corresponding values of the \(x_j\) are each in the range \((-1, +1)\), and so that at points one unit away from the solution, \(F(x)\) differs from its value at the solution by approximately one unit. This will usually imply that the Hessian matrix at the solution is well conditioned. It is unlikely that you will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that \texttt{nag_opt_bounds_2nd_deriv (e04lbc)} will take less computer time.

9.3 Unconstrained Minimization

If a problem is genuinely unconstrained and has been scaled sensibly, the following points apply:

(a) \(n_z\) will always be \(n_z\),
(b) the optional arguments \texttt{options.hesl} and \texttt{options.hesd} will be factors of the full approximate second derivative matrix with elements stored in the natural order,
(c) the elements of \texttt{g} should all be close to zero at the final point,
(d) the \texttt{Status} values given in the printout from \texttt{nag_opt_bounds_2nd_deriv (e04lbc)}, and in the optional argument \texttt{options.state} on exit are unlikely to be of interest (unless they are negative, which would indicate that the modulus of one of the \(x_j\) has reached \(10^{10}\) for some reason),
(e) \texttt{Norm g} simply gives the norm of the first derivative vector.
10 Example
This example minimizes the function

\[ F = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4 \]

subject to the bounds

\[ \begin{align*}
1 & \leq x_1 \leq 3 \\
-2 & \leq x_2 \leq 0 \\
1 & \leq x_4 \leq 3
\end{align*} \]

starting from the initial guess \((1.46, -0.82, 0.57, 1.21)^T\).

The options structure is declared and initialized by nag_opt_init (e04xcc). One option value is read from a data file by use of nag_opt_read (e04xyc). The memory freeing function nag_opt_free (e04xzc) is used to free the memory assigned to the pointers in the option structure. You must not use the standard C function free() for this purpose.

10.1 Program Text

```c
/* nag_opt_bounds_2nd_deriv (e04lbc) Example Program. */
* Copyright 2014 Numerical Algorithms Group.
* Mark 6 revised, 2000.
* Mark 7 revised, 2001.
*/
#include <nag.h>
#include <stdio.h>
#include <string.h>
#include <nag_stdlib.h>
#include <math.h>
#include <nage04.h>
#ifdef __cplusplus
extern "C" { 
#endif
static void NAG_CALL funct(Integer n, const double xc[], double *fc,
    double gc[], Nag_Comm *comm);
static void NAG_CALL h(Integer n, const double xc[], double fhesl[],
    double fhesd[], Nag_Comm *comm);
#ifndef __cplusplus
}
#endif
int main(void)
{
    const char *optionsfile = "e04lbce.opt";
    static double ruser[2] = {-1.0, -1.0};
    Integer exit_status = 0;
    Nag_Boolean print;
    Integer n = 4;
    Nag_Comm comm;
    Nag_E04_Opt options;
    double *bl = 0, *bu = 0, *f = 0, *g = 0, *x = 0;
    NagError fail;
    INIT_FAIL(fail);

    printf("nag_opt_bounds_2nd_deriv (e04lbc) Example Program Results\n");
    /* For communication with user-supplied functions: */
```
comm.user = ruser;

if (n >= 1)
{
    if (!((x = NAG_ALLOC(n, double)) ||
           (bl = NAG_ALLOC(n, double)) ||
           (bu = NAG_ALLOC(n, double)) ||
           (g = NAG_ALLOC(n, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }
}
else
{
    printf("Invalid n.\n");
    exit_status = 1;
    return exit_status;
}

b1[0] = 1.0;
bu[0] = 3.0;
b1[1] = -2.0;
bu[1] = 0.0;

/* x[2] is not bounded, so we set b1[2] to a large negative
 * number and bu[2] to a large positive number */
b1[2] = -1e6;
bu[2] = 1e6;
b1[3] = 1.0;
bu[3] = 3.0;

/* Set up starting point */
x[0] = 3.0;
x[1] = -1.0;
x[2] = 0.0;
x[3] = 1.0;

print = Nag_TRUE;

nag_opt_init(&options);

nag_opt_read("e04lbc", optionsfile, &options, print, "stdout", &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_opt_read (e04xyc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

nag_opt_bounds_2nd_deriv(n, funct, h, Nag_Bounds, bl, bu, x, &f, g,
                       &options, &comm, &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error or warning from 
    "nag_opt_bounds_2nd_deriv (e04lbc).\n%s\n", fail.message);
    if (fail.code != NW_COND_MIN)
        exit_status = 1;
}

/* Free memory allocated by nag_opt_bounds_deriv (e04kbc) to pointers hesd, 
 * hesl and state. */

nag_opt_free (e04xzc).
* Memory freeing function for use with option setting
*/
    nag_opt_free(&options, "all", &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_opt_bounds_2nd_deriv (e04lbc).
 fail.message);
        exit_status = 1;
        goto END;
    }

END:
NAG_FREE(x);
NAG_FREE(bl);
NAG_FREE(bu);
NAG_FREE(g);
return exit_status;
}

static void NAG_CALL funct(Integer n, const double xc[], double *fc,
    double gc[], Nag_Comm *comm)
{
    /* Function to evaluate objective function and its 1st derivatives. */
    double term1, term1_sq;
    double term2, term2_sq;
    double term3, term3_sq, term3_cu;
    double term4, term4_sq, term4_cu;
    if (comm->user[0] == -1.0)
    {
        printf("(User-supplied callback funct, first invocation.)\n");
        fflush(stdout);
        comm->user[0] = 0.0;
    }
    term1 = xc[0] + 10.0*xc[1];
    term1_sq = term1*term1;
    term2 = xc[2] - xc[3];
    term2_sq = term2*term2;
    term3 = xc[1] - 2.0*xc[2];
    term3_sq = term3*term3;
    term3_cu = term3*term3_sq;
    term4 = xc[0] - xc[3];
    term4_sq = term4*term4;
    term4_cu = term4_sq*term4;
    *fc = term1_sq + 5.0*term2_sq
        + term3_sq*term3_sq + 10.0*term4_sq*term4_sq;
    gc[0] = 2.0*term1 + 40.0*term4_cu;
    gc[1] = 20.0*term1 + 4.0*term3_cu;
    gc[2] = 10.0*term2 - 8.0*term3_cu;
    gc[3] = -10.0*term2 - 40.0*term4_cu;
}

/* funct */

static void NAG_CALL h(Integer n, const double xc[], double fhesl[],
    double fhesd[], Nag_Comm *comm)
{
    /* Routine to evaluate 2nd derivatives */
    double term3_sq; 
    double term4_sq; 
    if (comm->user[1] == -1.0)
    {
        printf("(User-supplied callback h, first invocation.)\n");
        fflush(stdout);
        comm->user[1] = 0.0;
    }
}
term4_sq = (xc[0] - xc[3])*(xc[0] - xc[3]);

fhesd[0] = 2.0 + 120.0*term4_sq;
fhesd[1] = 200.0 + 12.0*term3_sq;
fhesd[2] = 10.0 + 48.0*term3_sq;
fhesd[3] = 10.0 + 120.0*term4_sq;

fhesl[0] = 20.0;
fhesl[1] = 0.0;
fhesl[2] = -24.0*term3_sq;
fhesl[3] = -120.0*term4_sq;
fhesl[4] = 0.0;
fhesl[5] = -10.0;

/ * h */

10.2 Program Data

nag_opt_bounds_2nd_deriv (e04lbc) Example Program Optional Parameters

begin e04lbc
  print_level = Nag_Soln
end

10.3 Program Results

nag_opt_bounds_2nd_deriv (e04lbc) Example Program Results

Optional parameter setting for e04lbc.
--------------------------------------
Option file: e04lbce.opt
print_level set to Nag_Soln
Parameters to e04lbc
-------------------
Number of variables........... 4

optim_tol............... 1.05e-07 linesearch_tol........... 9.00e-01
step_max................ 1.00e+05 max_iter.............. 200
print_level........... Nag_Soln machine precision...... 1.11e-16
deriv_check........... Nag_TRUE
outfile............... stdout

Memory allocation:
state................ Nag hesl................ Nag hesd.............. Nag
(User-supplied callback funct, first invocation.)
(User-supplied callback h, first invocation.)

Final solution:
Itn Nfun Objective Norm g Norm x Norm step Step CondH PosDef
10 14 2.4338e+00 1.3e-09 1.5e+00 2.4e-11 1.0e+00 4.4e+00 Yes

Variable x g Status
1 1.0000e+00 2.9535e-01 Lower Bound
2 -8.5233e-02 -5.8675e-10 Free
3 4.0930e-01 1.1735e-09 Free
4 1.0000e+00 5.9070e+00 Lower Bound

Error or warning from nag_opt_bounds_2nd_deriv (e04lbc).
NW_COND_MIN:
The conditions for a minimum have not all been satisfied but a lower
point could not be found.
11 Optional Arguments

A number of optional input and output arguments to nag_opt_bounds_2nd_deriv (e04lbc) are available through the structure argument options, type Nag_E04_Opt. An argument may be selected by assigning an appropriate value to the relevant structure member; those arguments not selected will be assigned default values. If no use is to be made of any of the optional arguments you should use the NAG defined null pointer, E04_DEFAULT, in place of options when calling nag_opt_bounds_2nd_deriv (e04lbc); the default settings will then be used for all arguments.

Before assigning values to options directly the structure must be initialized by a call to the function nag_opt_init (e04xxc). Values may then be assigned to the structure members in the normal C manner.

Option settings may also be read from a text file using the function nag_opt_read (e04xyc) in which case initialization of the options structure will be performed automatically if not already done. Any subsequent direct assignment to the options structure must not be preceded by initialization.

If assignment of functions and memory to pointers in the options structure is required, then this must be done directly in the calling program; they cannot be assigned using nag_opt_read (e04xyc).

11.1 Optional Argument Checklist and Default Values

For easy reference, the following list shows the members of options which are valid for nag_opt_bounds_2nd_deriv (e04lbc) together with their default values where relevant. The number $\epsilon$ is a generic notation for machine precision (see nag_machine_precision (X02AJC)).

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean list</td>
<td>Nag_TRUE</td>
</tr>
<tr>
<td>Nag_PrintType print_level</td>
<td>Nag_Soln_Iter</td>
</tr>
<tr>
<td>char outfile[80]</td>
<td>stdout</td>
</tr>
<tr>
<td>void (*print_fun)()</td>
<td>NULL</td>
</tr>
<tr>
<td>Boolean deriv_check</td>
<td>Nag_TRUE</td>
</tr>
<tr>
<td>Integer max_iter</td>
<td>50$n$</td>
</tr>
<tr>
<td>double optim_tol</td>
<td>$10\sqrt{\epsilon}$</td>
</tr>
<tr>
<td>double linesearch_tol</td>
<td>0.9 (0.0 if $n$ = 1)</td>
</tr>
<tr>
<td>double step_max</td>
<td>1000000.0</td>
</tr>
<tr>
<td>Integer *state</td>
<td>size $n$</td>
</tr>
<tr>
<td>double *hesl</td>
<td>size max$(n(n-1)/2,1)$</td>
</tr>
<tr>
<td>double *hesd</td>
<td>size $n$</td>
</tr>
<tr>
<td>Integer iter</td>
<td></td>
</tr>
<tr>
<td>Integer nf</td>
<td></td>
</tr>
</tbody>
</table>

11.2 Description of the Optional Arguments

list – Nag_Boolean

On entry: if options.list = Nag_TRUE the argument settings in the call to nag_opt_bounds_2nd_deriv (e04lbc) will be printed.

Default = Nag_TRUE

print_level – Nag_PrintType

On entry: the level of results printout produced by nag_opt_bounds_2nd_deriv (e04lbc). The following values are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nag_NoPrint</td>
<td>No output.</td>
</tr>
<tr>
<td>Nag_Soln</td>
<td>The final solution.</td>
</tr>
<tr>
<td>Nag_Iter</td>
<td>One line of output for each iteration.</td>
</tr>
<tr>
<td>Nag_Soln_Iter</td>
<td>The final solution and one line of output for each iteration.</td>
</tr>
<tr>
<td>Nag_Soln_Iter_Full</td>
<td>The final solution and detailed printout at each iteration.</td>
</tr>
</tbody>
</table>
Details of each level of results printout are described in Section 11.3.

**Constraint:** 
- `options.print_level` = Nag_NoPrint, Nag_Soln, Nag_Iter, Nag_Soln_Iter or Nag_Soln_Iter_Full.

- `outfile` = const char[80] 
  Default = stdout
  
  *On entry:* the name of the file to which results should be printed. If `options.outfile[0] = '\0'` then the stdout stream is used.

- `print_fun` = pointer to function 
  Default = NULL
  
  *On entry:* printing function defined by you; the prototype of `options.print_fun` is
  ```c
  void (*print_fun)(const Nag_Search_State *st, Nag_COmm *comm);
  ```
  See Section 11.3.1 below for further details.

- `deriv_check` = Nag_Boolean 
  Default = Nag_TRUE
  
  *On entry:* if `options.deriv_check = Nag_TRUE` a check of the derivatives defined by `objfun` and `hessfun` will be made at the starting point `x`. A starting point of `x = 0` or `x = 1` should be avoided if this test is to be meaningful.

- `max_iter` = Integer 
  Default = 50n
  
  *On entry:* the limit on the number of iterations allowed before termination.
  
  **Constraint:** `options.max_iter ≥ 0`.

- `optim_tol` = double 
  Default = `10√ε`
  
  *On entry:* the accuracy in `x` to which the solution is required. If `x_true` is the true value of `x` at the minimum, then `x_sol`, the estimated position prior to a normal exit, is such that
  ```markdown
  \|x_sol - x_true\| < options.optim_tol \times (1.0 + \|x_true\|),
  ```
  where \(\|y\| = \left(\sum_{j=1}^{n} y_j^2\right)^{1/2}\). For example, if the elements of `x_sol` are not much larger than 1.0 in modulus and if `options.optim_tol` is set to `10^{-5}`, then `x_sol` is usually accurate to about five decimal places. (For further details see Section 9.) If the problem is scaled roughly as described in Section 9 and \(ε\) is the **machine precision**, then \(√ε\) is probably the smallest reasonable choice for `options.optim_tol` (This is because, normally, to machine accuracy, \(F(x + √εe_j) = F(x)\) where `e_j` is any column of the identity matrix.)
  
  **Constraint:** \(ε ≤ options.optim_tol < 1.0\).

- `linesearch_tol` = double 
  Default = 0.9 if `n > 1`, and 0.0 otherwise
  
  *On entry:* every iteration of `nag_opt_bounds_2nd_deriv` (e04lbc) involves a linear minimization (i.e., minimization of \(F(x + αp)\) with respect to `α`). `options.linesearch_tol` specifies how accurately these linear minimizations are to be performed. The minimum with respect to `α` will be located more accurately for small values of `options.linesearch_tol` (say 0.01) than for large values (say 0.9).

  Although accurate linear minimizations will generally reduce the number of iterations performed by `nag_opt_bounds_2nd_deriv` (e04lbc), they will increase the number of function evaluations required for each iteration. On balance, it is usually more efficient to perform a low accuracy linear minimization. A smaller value such as 0.01 may be worthwhile:
  
  (a) if `objfun` takes so little computer time that it is worth using extra calls of `objfun` to reduce the number of iterations and associated matrix calculations
  
  (b) if calls to `hessfun` are expensive compared with calls to `objfun`.
  
  (c) if \(F(x)\) is a penalty or barrier function arising from a constrained minimization problem (since such problems are very difficult to solve).
If $n = 1$, the default for `options.linesearch_tol` = 0.0 (if the problem is effectively one-dimensional then `options.linesearch_tol` should be set to 0.0 even though $n > 1$; i.e., if for all except one of the variables the lower and upper bounds are equal).

**Constraint:** $0.0 \leq \text{options.linesearch_tol} < 1.0$.

### `step_max` – double

**Default** = 100000.0

**On entry:** an estimate of the Euclidean distance between the solution and the starting point supplied by you. (For maximum efficiency a slight overestimate is preferable.) `nag_opt_bounds_2nd_deriv (e04lbc)` will ensure that, for each iteration,

$$
\left( \sum_{j=1}^{n} \left[ x_j^{(k)} - x_j^{(k-1)} \right]^2 \right)^{1/2} \leq \text{options.step_max},
$$

where $k$ is the iteration number. Thus, if the problem has more than one solution, `nag_opt_bounds_2nd_deriv (e04lbc)` is most likely to find the one nearest the starting point. On difficult problems, a realistic choice can prevent the sequence of $x^{(k)}$ entering a region where the problem is ill-behaved and can also help to avoid possible overflow in the evaluation of $F(x)$. However, an underestimate of `options.step_max` can lead to inefficiency.

**Constraint:** `options.step_max` $\geq$ `options.optim_tol`.

### `state` – Integer *

**Default memory** = $n$

**On exit:** `options.state` contains information about which variables are on their bounds and which are free at the final point given in $x$. If $x_j$ is:

(a) fixed on its upper bound, `options.state[j - 1]` is $-1$;
(b) fixed on its lower bound, `options.state[j - 1]` is $-2$;
(c) effectively a constant (i.e., $l_j = u_j$), `options.state[j - 1]` is $-3$;
(d) free, `options.state[j - 1]` gives its position in the sequence of free variables.

### `hesl` – double *

**Default memory** = $\max(n(n - 1)/2, 1)$

**On exit:** during the determination of a direction $p_z$ (see Section 3), $H + E$ is decomposed into the product $LDL^T$, where $L$ is a unit lower triangular matrix and $D$ is a diagonal matrix. (The matrices $H$, $E$, $L$ and $D$ are all of dimension $n_z$, where $n_z$ is the number of variables free from their bounds. $H$ consists of those rows and columns of the full second derivative matrix which relate to free variables. $E$ is chosen so that $H + E$ is positive definite.)

`options.hesl` and `options.hesd` are used to store the factors $L$ and $D$. The elements of the strict lower triangle of $L$ are stored row by row in the first $n_z(n_z - 1)/2$ positions of `options.hesl`. The diagonal elements of $D$ are stored in the first $n_z$ positions of `options.hesd`.

In the last factorization before a normal exit, the matrix $E$ will be zero, so that `options.hesl` and `options.hesd` will contain, on exit, the factors of the final second derivative matrix $H$. The elements of `options.hesd` are useful for deciding whether to accept the result produced by `nag_opt_bounds_2nd_deriv (e04lbc)` (see Section 9).

### `iter` – Integer

**On exit:** the number of iterations which have been performed in `nag_opt_bounds_2nd_deriv (e04lbc)`.

### `nf` – Integer

**On exit:** the number of times the residuals have been evaluated (i.e., number of calls of `objfun`).
11.3 Description of Printed Output

The level of printed output can be controlled with the structure members `options.list` and `options.print_level` (see Section 11.2). If `options.list = Nag_TRUE` then the argument values to `nag_opt_bounds_2nd_deriv (e04lbc)` are listed, whereas the printout of results is governed by the value of `options.print_level`. The default of `options.print_level = Nag_Soln_Iter` provides a single line of output at each iteration and the final result. This section describes all of the possible levels of results printout available from `nag_opt_bounds_2nd_deriv (e04lbc)`.

When `options.print_level = Nag_Iter` or `Nag_Soln_Iter` the following line of output is produced on completion of each iteration.

- **Itn**: the iteration count, \( k \).
- **Nfun**: the cumulative number of calls made to `objfun`.
- **Objective**: the value of the objective function, \( F(x^{(k)}) \).
- **Norm g**: the Euclidean norm of the projected gradient vector, \( ||g_c(x^{(k)})|| \).
- **Norm x**: the Euclidean norm of \( x^{(k)} \).
- **Norm(x(k-1)-x(k))**: the Euclidean norm of \( x^{(k-1)} - x^{(k)} \).
- **Step**: the step \( \alpha^{(k)} \) taken along the computed search direction \( p^{(k)} \).
- **Cond H**: the ratio of the largest to the smallest element of the diagonal factor \( D \) of the projected Hessian matrix. This quantity is usually a good estimate of the condition number of the projected Hessian matrix. (If no variables are currently free, this value will be zero.)
- **PosDef**: indicates whether the second derivative matrix for the current subspace, \( H \), is positive definite (`Yes`) or not (`No`).

When `options.print_level = Nag_Soln_Iter_Full` more detailed results are given at each iteration. Additional values output are

- **x**: the current point \( x^{(k)} \).
- **g**: the current projected gradient vector, \( g_c(x^{(k)}) \).
- **Status**: the current state of the variable with respect to its bound(s).

If `options.print_level = Nag_Soln`, `Nag_Soln_Iter` or `Nag_Soln_Iter_Full` the final result is printed out. This consists of:

- **x**: the final point, \( x^* \).
- **g**: the final projected gradient vector, \( g_c(x^*) \).
- **Status**: the final state of the variable with respect to its bound(s).

If `options.print_level = Nag_NoPrint` then printout will be suppressed; you can print the final solution when `nag_opt_bounds_2nd_deriv (e04lbc)` returns to the calling program.

11.3.1 Output of results via a user-defined printing function

You may also specify your own print function for output of iteration results and the final solution by use of the `options.print_fun` function pointer, which has prototype

```c
void (*print_fun)(const Nag_Search_State *st, Nag_Comm *comm);
```

The rest of this section can be skipped if the default printing facilities provide the required functionality.

When a user-defined function is assigned to `options.print_fun` this will be called in preference to the internal print function of `nag_opt_bounds_2nd_deriv (e04lbc)`. Calls to the user-defined function are
again controlled by means of the `options.print_level` member. Information is provided through `st` and `comm`, the two structure arguments to `options.print_fun`.

If `comm->it_prt` = Nag_TRUE then the results on completion of an iteration of `nag_opt_bounds_2nd_deriv (e04lbc)` are contained in the members of `st`. If `comm->sol_prt` = Nag_TRUE then the final results from `nag_opt_bounds_2nd_deriv (e04lbc)`, including details of the final iteration, are contained in the members of `st`. In both cases, the same members of `st` are set, as follows:

**iter** – Integer

The current iteration count, \( k \), if `comm->it_prt` = Nag_TRUE; the final iteration count, \( k \), if `comm->sol_prt` = Nag_TRUE.

**n** – Integer

The number of variables.

**x** – double *

The coordinates of the point \( x^{(k)} \).

**f** – double *

The value of the objective function at \( x^{(k)} \).

**g** – double *

The value of \( \frac{\partial F}{\partial x_j} \) at \( x^{(k)} \), \( j = 1, 2, \ldots, n \).

**gpj_norm** – double

The Euclidean norm of the projected gradient \( g_z \) at \( x^{(k)} \).

**step** – double

The step \( \alpha^{(k)} \) taken along the search direction \( p^{(k)} \).

**cond** – double

The estimate of the condition number of the projected Hessian matrix, see Section 11.3.

**xk_norm** – double

The Euclidean norm of \( x^{(k-1)} - x^{(k)} \).

**state** – Integer *

The status of variables \( x_j \), for \( j = 1, 2, \ldots, n \), with respect to their bounds. See Section 11.2 for a description of the possible status values.

**posdef** – Nag_Boolean

Will be Nag_TRUE if the second derivative matrix \( H \) for the current subspace is positive definite, and Nag_FALSE otherwise.

**nf** – Integer

The cumulative number of calls made to `objfun`.

The relevant members of the structure `comm` are:

**it_prt** – Nag_Boolean

Will be Nag_TRUE when the print function is called with the results of the current iteration.

**sol_prt** – Nag_Boolean

Will be Nag_TRUE when the print function is called with the final result.
user – double *
iuser – Integer *
p – Pointer

Pointers for communication of user information. If used they must be allocated memory either before entry to nag_opt_bounds_2nd_deriv (e04lbc) or during a call to objfun or options.print_fun. The type Pointer will be void * with a C compiler that defines void * and char * otherwise.