NAG Library Function Document

nag_opt_sparse_nlp_jacobian (e04vjc)

1 Purpose

nag_opt_sparse_nlp_jacobian (e04vjc) may be used before nag_opt_sparse_nlp_solve (e04vhc) to determine the sparsity pattern for the Jacobian.

2 Specification

```
#include <nag.h>
#include <nage04.h>
void nag_opt_sparse_nlp_jacobian (Integer nf, Integer n,
     void (*usrfun)(Integer *status, Integer n, const double x[],
          Integer needf, Integer nf, double f[], Integer needg, Integer leng,
          double g[], Nag_Comm *comm),
     Integer iafun[], Integer javar[], double a[], Integer lena,
     Integer *nea, Integer igfun[], Integer jgvar[], Integer leng,
     Integer *neg, const double x[], const double xlow[],
     const double xupp[], Nag_E04State *state, Nag_Comm *comm,
     NagError *fail)
```

3 Description

When using nag_opt_sparse_nlp_solve (e04vhc), if you set the optional parameter **Derivative Option** $= 0$ and **usrfun** provides none of the derivatives, you may need to call nag_opt_sparse_nlp_jacobian (e04vjc) to determine the input arrays **iafun**, **javar**, **a**, **igfun** and **jgvar**. These arrays define the pattern of nonzeros in the Jacobian matrix. A typical sequence of calls could be

```
e04vgc (&state, ... );
 e04vjc (nf, n, ... );
 e04vlc ("Derivative Option = 0", &state, ... );
 e04vhc (start, nf, ... );
```

nag_opt_sparse_nlp_jacobian (e04vjc) determines the sparsity pattern for the Jacobian and identifies the constant elements automatically. To do so, nag_opt_sparse_nlp_jacobian (e04vjc) approximates the problem functions, $F(x)$, at three random perturbations of the given initial point x. If an element of the approximate Jacobian is the same at all three points, then it is taken to be constant. If it is zero, it is taken to be identically zero. Since the random points are not chosen close together, the heuristic will correctly classify the Jacobian elements in the vast majority of cases. In general, nag_opt_sparse_nlp_jacobian (e04vjc) finds that the Jacobian can be permuted to the form:

$$\begin{pmatrix} G(x) & A_3 \\ A_2 & A_4 \end{pmatrix},$$

where A_2, A_3 and A_4 are constant. Note that $G(x)$ might contain elements that are also constant, but nag_opt_sparse_nlp_jacobian (e04vjc) must classify them as nonlinear. This is because nag_opt_sparse_nlp_solve (e04vhc) 'removes' linear variables from the calculation of F by setting them to zero before calling **usrfun**. A knowledgeable user would be able to move such elements from $F(x)$ in **usrfun** and enter them as part of **iafun**, **javar** and **a** for nag_opt_sparse_nlp_solve (e04vhc).

4 References

Hock W and Schittkowski K (1981) *Test Examples for Nonlinear Programming Codes. Lecture Notes in Economics and Mathematical Systems* **187** Springer–Verlag

5 Arguments

Note: all optional parameters are described in detail in Section 12.1 in nag_opt_sparse_nlp_solve (e04vhc).

1: **nf** – Integer *Input*

On entry: nf, the number of problem functions in $F(x)$, including the objective function (if any) and the linear and nonlinear constraints. Simple upper and lower bounds on x can be defined using the arguments **xlow** and **xupp** and should not be included in F.

Constraint: **nf** > 0.

2: **n** – Integer *Input*

On entry: n, the number of variables.

Constraint: **n** > 0.

3: **usrfun** – function, supplied by the user *External Function*

usrfun must define the problem functions $F(x)$. This function is passed to nag_opt_sparse_nlp_jacobian (e04vjc) as the external argument **usrfun**.

The specification of **usrfun** is:

```
void usrfun (Integer *status, Integer n, const double x[],
     Integer needf, Integer nf, double f[], Integer needg,
     Integer leng, double g[], Nag_Comm *comm)
```

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

On entry: indicates the first call to **usrfun**.

status $= 0$
There is nothing special about the current call to **usrfun**.

status $= 1$
nag_opt_sparse_nlp_jacobian (e04vjc) is calling your function for the *first* time. Some data may need to be input or computed and saved.

On exit: may be used to indicate that you are unable to evaluate F at the current x. (For example, the problem functions may not be defined there).

nag_opt_sparse_nlp_jacobian (e04vjc) evaluates $F(x)$ at random perturbation of the initial point x, say x_p. If the functions cannot be evaluated at x_p, you can set **status** $= -1$, nag_opt_sparse_nlp_jacobian (e04vjc) will use another random perturbation.

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

2: **n** – Integer *Input*

On entry: n, the number of variables, as defined in the call to nag_opt_sparse_nlp_jacobian (e04vjc).

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

On entry: the variables x at which the problem functions are to be calculated. The array x must not be altered.

4: **needf** – Integer *Input*

On entry: indicates if **f** must be assigned during the call to **usrfun** (see **f**).

5: **nf** – Integer *Input*

On entry: nf, the number of problem functions.

6: **f**[**nf**] – double *Input/Output*

On entry: this will be set by nag_opt_sparse_nlp_jacobian (e04vjc).

On exit: the computed $F(x)$ according to the setting of **needf**.

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

If **needf** > 0, the components of $F(x)$ must be calculated and assigned to **f**. nag_opt_sparse_nlp_jacobian (e04vjc) will always call **usrfun** with **needf** > 0.

To simplify the code, you may ignore the value of **needf** and compute $F(x)$ on every entry to **usrfun**.

7: **needg** – Integer *Input*

On entry: nag_opt_sparse_nlp_jacobian (e04vjc) will call **usrfun** with **needg** $= 0$ to indicate that **g** is not required.

8: **leng** – Integer *Input*

On entry: the dimension of the array **g**.

9: **g**[**leng**] – double *Input/Output*

On entry: concerns the calculations of the derivatives of the function $f(x)$.

On exit: nag_opt_sparse_nlp_jacobian (e04vjc) will always call **usrfun** with **needg** $= 0$: **g** is not required to be set on exit but must be declared correctly.

10: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_opt_sparse_nlp_jacobian (e04vjc) you may allocate memory and initialize these pointers with various quantities for use by **usrfun** when called from nag_opt_sparse_nlp_jacobian (e04vjc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

4: **iafun**[**lena**] – Integer *Output*
5: **javar**[**lena**] – Integer *Output*
6: **a**[**lena**] – double *Output*

On exit: define the coordinates (i,j) and values A_{ij} of the nonzero elements of the linear part A of the function $F(x) = f(x) + Ax$.

In particular, **nea** triples $(\mathbf{iafun}[k-1], \mathbf{javar}[k-1], \mathbf{a}[k-1])$ define the row and column indices $i = \mathbf{iafun}[k-1]$ and $j = \mathbf{javar}[k-1]$ of the element $A_{ij} = \mathbf{a}[k-1]$.

7: **lena** – Integer *Input*

On entry: the dimension of the arrays **iafun**, **javar** and **a** that hold $\left(i,j,A_{ij}\right)$. **lena** should be an *overestimate* of the number of elements in the linear part of the Jacobian.

Constraint: **lena** ≥ 1.

8: **nea** – Integer * *Output*

On exit: is the number of nonzero entries in A such that $F(x) = f(x) + Ax$.

9: **igfun**[**leng**] – Integer *Output*
10: **jgvar**[**leng**] – Integer *Output*

On exit: define the coordinates (i, j) of the nonzero elements of G, the nonlinear part of the derivatives $J(x) = G(x) + A$ of the function $F(x) = f(x) + Ax$.

11: **leng** – Integer *Input*

On entry: the dimension of the arrays **igfun** and **jgvar** that define the varying Jacobian elements $\left(i, j, G_{ij}\right)$. **leng** should be an *overestimate* of the number of elements in the nonlinear part of the Jacobian.

Constraint: **leng** ≥ 1.

12: **neg** – Integer * *Output*

On exit: the number of nonzero entries in G.

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

On entry: an initial estimate of the variables x. The contents of x will be used by nag_opt_sparse_nlp_jacobian (e04vjc) in the call of **usrfun**, and so each element of **x** should be within the bounds given by **xlow** and **xupp**.

14: **xlow**[**n**] – const double *Input*
15: **xupp**[**n**] – const double *Input*

On entry: contain the lower and upper bounds l_x and u_x on the variables x.

To specify a nonexistent lower bound $[l_x]_j = -\infty$, set **xlow**$[j-1] \leq -bigbnd$, where $bigbnd$ is the optional parameter **Infinite Bound Size**. To specify a nonexistent upper bound **xupp**$[j-1] \geq bigbnd$.

To fix the jth variable (say, $x_j = \beta$, where $|\beta| < bigbnd$), set **xlow**$[j-1] =$ **xupp**$[j-1] = \beta$.

16: **state** – Nag_E04State * *Communication Structure*

state contains internal information required for functions in this suite. It must not be modified in any way.

17: **comm** – Nag_Comm *

The NAG communication argument (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

Internal error: memory allocation failed when attempting to allocate workspace sizes ⟨*value*⟩, ⟨*value*⟩ and ⟨*value*⟩. Please contact NAG.

NE_ALLOC_INSUFFICIENT

Internal memory allocation was insufficient. Please contact NAG.

NE_ARRAY_TOO_SMALL

Either **lena** or **leng** is too small. Increase both of them and corresponding array sizes. **lena** $= \langle value \rangle$ and **leng** $= \langle value \rangle$.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ had an illegal value.

NE_E04VGC_NOT_INIT

The initialization function nag_opt_sparse_nlp_init (e04vgc) has not been called.

NE_INT

On entry, **lena** $= \langle value \rangle$.
Constraint: **lena** ≥ 1.

On entry, **leng** $= \langle value \rangle$.
Constraint: **leng** ≥ 1.

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_JACOBIAN_STRUCTURE_FAIL

Cannot estimate Jacobian structure at given point **x**.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

NE_USER_STOP

User-supplied function **usrfun** requested termination.

You have indicated the wish to terminate the call to nag_opt_sparse_nlp_jacobian (e04vjc) by setting ***status*** *to a value* < -1 *on exit from* ***usrfun****.*

NE_USRFUN_UNDEFINED

User-supplied function **usrfun** indicates that functions are undefined near given point **x**.

You have indicated that the problem functions are undefined by setting **status** $= -1$ *on exit from* ***usrfun****. This exit occurs if nag_opt_sparse_nlp_jacobian (e04vjc) is unable to find a point at which the functions are defined.*

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_opt_sparse_nlp_jacobian (e04vjc) 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.

Please consult the x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

This example shows how to call nag_opt_sparse_nlp_jacobian (e04vjc) to determine the sparsity pattern of the Jacobian before calling nag_opt_sparse_nlp_solve (e04vhc) to solve a sparse nonlinear programming problem without providing the Jacobian information in **usrfun**.

It is a reformulation of Problem 74 from Hock and Schittkowski (1981) and involves the minimization of the nonlinear function

$$f(x) = 10^{-6}x_3^3 + \tfrac{2}{3} \times 10^{-6}x_4^3 + 3x_3 + 2x_4$$

subject to the bounds

$$\begin{aligned} -0.55 \le x_1 \le 0.55, \\ -0.55 \le x_2 \le 0.55, \\ 0 \le x_3 \le 1200, \\ 0 \le x_4 \le 1200, \end{aligned}$$

to the nonlinear constraints

$$\begin{aligned} 1000\sin(-x_1 - 0.25) + 1000\sin(-x_2 - 0.25) - x_3 &= -894.8, \\ 1000\sin(x_1 - 0.25) + 1000\sin(x_1 - x_2 - 0.25) - x_4 &= -894.8, \\ 1000\sin(x_2 - 0.25) + 1000\sin(x_2 - x_1 - 0.25) &= -1294.8, \end{aligned}$$

and to the linear constraints

$$\begin{aligned} -x_1 + x_2 \ge -0.55, \\ x_1 - x_2 \ge -0.55. \end{aligned}$$

The initial point, which is infeasible, is

$$x_0 = \begin{pmatrix} 0, & 0, & 0, & 0 \end{pmatrix}^{\mathrm{T}},$$

and $f(x_0) = 0$.

The optimal solution (to five figures) is

$$x^* = (0.11887, -0.39623, 679.94, 1026.0)^{\mathrm{T}},$$

and $f(x^*) = 5126.4$. All the nonlinear constraints are active at the solution.

The formulation of the problem combines the constraints and the objective into a single vector (F).

$$F = \begin{pmatrix} 1000\sin(-x_1 - 0.25) + 1000\sin(-x_2 - 0.25) - x_3 \\ 1000\sin(x_1 - 0.25) + 1000\sin(x_1 - x_2 - 0.25) - x_4 \\ 1000\sin(x_2 - 0.25) + 1000\sin(x_2 - x_1 - 0.25) \\ -x_1 + x_2 \\ x_1 - x_2 \\ 10^{-6}x_3^3 + \tfrac{2}{3} \times 10^{-6}x_4^3 + 3x_3 + 2x_4 \end{pmatrix}$$

10.1 Program Text

```
/* nag_opt_sparse_nlp_jacobian (e04vjc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nage04.h>

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL usrfun(Integer *status, Integer n, const double x[],
                              Integer needf, Integer nf, double f[],
                              Integer needg, Integer leng, double g[],
                              Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

int main(void)
{
  /* Scalars */
  double objadd, sinf;
  Integer exit_status = 0;
  Integer i, lena, leng, n, nea, neg, nf, nfname, ninf, ns, nxname, objrow;

  /* Arrays */
  char **fnames = 0, prob[9], **xnames = 0;
  double *a = 0, *f = 0, *flow = 0, *fmul = 0, *fupp = 0, *x = 0;
  double *xlow = 0, *xmul = 0, *xupp = 0;
  Integer *fstate = 0, *iafun = 0, *igfun = 0, *javar = 0, *jgvar = 0;
  Integer *xstate = 0;

  /* Nag Types */
  Nag_E04State state;
  NagError fail;
  Nag_Comm comm;
  Nag_Start start;

  /* By default e04vhc does not print monitoring information.
     Define SHOW_MONITORING_INFO to turn it on - see further below. */
#ifdef SHOW_MONITORING_INFORMATION
  Nag_FileID fileid;
#endif

  exit_status = 0;
  INIT_FAIL(fail);

  printf("nag_opt_sparse_nlp_jacobian (e04vjc) Example Program Results\n");

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%" NAG_IFMT "%*[^\n] ", &n, &nf);
#else
  scanf("%" NAG_IFMT "%" NAG_IFMT "%*[^\n] ", &n, &nf);
```

```
#endif
  if (n > 0 && nf > 0) {
    nfname = 1;
    nxname = 1;
    lena = 300;
    leng = 300;
    /* Allocate memory */
    if (!(fnames = NAG_ALLOC(nfname, char *)) ||
        !(xnames = NAG_ALLOC(nxname, char *)) ||
        !(a = NAG_ALLOC(lena, double)) ||
        !(f = NAG_ALLOC(nf, double)) ||
        !(flow = NAG_ALLOC(nf, double)) ||
        !(fmul = NAG_ALLOC(nf, double)) ||
        !(fupp = NAG_ALLOC(nf, double)) ||
        !(x = NAG_ALLOC(n, double)) ||
        !(xlow = NAG_ALLOC(n, double)) ||
        !(xmul = NAG_ALLOC(n, double)) ||
        !(xupp = NAG_ALLOC(n, double)) ||
        !(fstate = NAG_ALLOC(nf, Integer)) ||
        !(iafun = NAG_ALLOC(lena, Integer)) ||
        !(igfun = NAG_ALLOC(leng, Integer)) ||
        !(javar = NAG_ALLOC(lena, Integer)) ||
        !(jgvar = NAG_ALLOC(leng, Integer)) ||
        !(xstate = NAG_ALLOC(n, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  }
  else {
    printf("Invalid n or nf\n");
    exit_status = 1;
    goto END;
  }

  /* nag_opt_sparse_nlp_init (e04vgc).
   * Initialization function for nag_opt_sparse_nlp_solve
   * (e04vhc)
   */
  nag_opt_sparse_nlp_init(&state, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Initialization of nag_opt_sparse_nlp_init (e04vgc) failed.\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  /* Read the bounds on the variables. */
  for (i = 1; i <= n; ++i) {
#ifdef _WIN32
    scanf_s("%lf%lf%*[^\n] ", &xlow[i - 1], &xupp[i - 1]);
#else
    scanf("%lf%lf%*[^\n] ", &xlow[i - 1], &xupp[i - 1]);
#endif
  }

  for (i = 1; i <= n; ++i) {
    x[i - 1] = 0.;
  }

  /* Illustrate how to pass information to the user-supplied
     function usrfun via the comm structure */
  comm.p = 0;

  /* Determine the Jacobian structure. */
  /* nag_opt_sparse_nlp_jacobian (e04vjc).
   * Determine the pattern of nonzeros in the Jacobian matrix
   * for nag_opt_sparse_nlp_solve (e04vhc)
   */
  nag_opt_sparse_nlp_jacobian(nf, n, usrfun, iafun, javar, a, lena, &nea,
```

```
                              igfun, jgvar, leng, &neg, x, xlow, xupp,
                              &state, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("nag_opt_sparse_nlp_jacobian (e04vjc) failed to determine the"
           " Jacobian structure\n");
    exit_status = 1;
    goto END;
  }

  /* Print the Jacobian structure. */

  printf("\n");
  printf("NEA (the number of nonzero entries in A) = %3" NAG_IFMT "\n", nea);

  printf("  I     IAFUN(I)   JAVAR(I)          A(I)\n");
  printf("----    --------   --------   -----------\n");

  for (i = 1; i <= nea; ++i) {
    printf("%3" NAG_IFMT "%10" NAG_IFMT "%10" NAG_IFMT "%18.4e\n", i,
           iafun[i - 1], javar[i - 1], a[i - 1]);
  }

  printf("\n");
  printf("NEG (the number of nonzero entries in G) = %3" NAG_IFMT "\n", neg);
  printf("  I     IGFUN(I)   JGVAR(I)\n");
  printf("----    --------   --------\n");

  for (i = 1; i <= neg; ++i) {
    printf("%3" NAG_IFMT "%10" NAG_IFMT "%10" NAG_IFMT "\n", i, igfun[i - 1],
           jgvar[i - 1]);
  }

  /* Now that we have the determined the structure of the
   * Jacobian, set up the information necessary to solve
   * the optimization problem.
   */
  start = Nag_Cold;
#ifdef _WIN32
  strcpy_s(prob, (unsigned)_countof(prob), "        ");
#else
  strcpy(prob, "        ");
#endif
  objadd = 0.0;
  for (i = 1; i <= n; ++i) {
    x[i - 1] = 0.;
    xstate[i - 1] = 0;
    xmul[i - 1] = 0.;
  }
  for (i = 1; i <= nf; ++i) {
    f[i - 1] = 0.;
    fstate[i - 1] = 0;
    fmul[i - 1] = 0.;
  }

  /* The row containing the objective function. */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%*[^\n] ", &objrow);
#else
  scanf("%" NAG_IFMT "%*[^\n] ", &objrow);
#endif

  /* Read the bounds on the functions. */
  for (i = 1; i <= nf; ++i) {
#ifdef _WIN32
    scanf_s("%lf%lf%*[^\n] ", &flow[i - 1], &fupp[i - 1]);
#else
    scanf("%lf%lf%*[^\n] ", &flow[i - 1], &fupp[i - 1]);
#endif
  }

#ifdef SHOW_MONITORING_INFO
```

```
  /* Call nag_open_file (x04acc) to set the print file fileid */
  /* nag_open_file (x04acc).
   * Open unit number for reading, writing or appending, and
   * associate unit with named file
   */
  nag_open_file("", 2, &fileid, &fail);
  if (fail.code != NE_NOERROR) {
    exit_status = 2;
    goto END;
  }
  /* nag_opt_sparse_nlp_option_set_integer (e04vmc).
   * Set a single option for nag_opt_sparse_nlp_solve (e04vhc)
   * from an integer argument
   */
  nag_opt_sparse_nlp_option_set_integer("Print file", fileid, &state, &fail);
  if (fail.code != NE_NOERROR) {
    exit_status = 1;
    goto END;
  }
#endif

  /* Tell nag_opt_sparse_nlp_solve (e04vhc) that we supply no derivatives in
   *  usrfun. */
  /* nag_opt_sparse_nlp_option_set_string (e04vlc).
   * Set a single option for nag_opt_sparse_nlp_solve (e04vhc)
   * from a character string
   */
  nag_opt_sparse_nlp_option_set_string("Derivative option 0", &state, &fail);
  if (fail.code != NE_NOERROR) {
    exit_status = 1;
    goto END;
  }
  for (i = 1; i <= nfname; ++i) {
    fnames[i - 1] = NAG_ALLOC(9, char);
#ifdef _WIN32
    strcpy_s(fnames[i - 1], 9, "");
#else
    strcpy(fnames[i - 1], "");
#endif
  }

  for (i = 1; i <= nxname; ++i) {
    xnames[i - 1] = NAG_ALLOC(9, char);
#ifdef _WIN32
    strcpy_s(xnames[i - 1], 9, "");
#else
    strcpy(xnames[i - 1], "");
#endif
  }

  /* Solve the problem. */
  /* nag_opt_sparse_nlp_solve (e04vhc).
   * General sparse nonlinear optimizer
   */
  fflush(stdout);
  nag_opt_sparse_nlp_solve(start, nf, n, nxname, nfname, objadd, objrow, prob,
                           usrfun, iafun, javar, a, lena, nea, igfun, jgvar,
                           leng, neg, xlow, xupp, (const char **) xnames,
                           flow, fupp, (const char **) fnames, x, xstate,
                           xmul, f, fstate, fmul, &ns, &ninf, &sinf, &state,
                           &comm, &fail);
  if (n > 0 && nf > 0) {
    for (i = 0; i < nxname; i++)
      NAG_FREE(xnames[i]);
    for (i = 0; i < nfname; i++)
      NAG_FREE(fnames[i]);
  }
  if (fail.code == NE_NOERROR || fail.code == NW_NOT_FEASIBLE) {
    printf("\n");
    printf("Final objective value = %11.1f\n", f[objrow - 1]);
    printf("Optimal X = ");
```

```
    for (i = 1; i <= n; ++i)
      printf("%9.2f%s", x[i - 1], i % 7 == 0 || i == n ? "\n" : " ");
  }
  else {
    printf("Error message from nag_opt_sparse_nlp_solve (e04vhc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }
  fflush(stdout);

  if (fail.code != NE_NOERROR)
    exit_status = 2;

END:
  NAG_FREE(fnames);
  NAG_FREE(xnames);
  NAG_FREE(a);
  NAG_FREE(f);
  NAG_FREE(flow);
  NAG_FREE(fmul);
  NAG_FREE(fupp);
  NAG_FREE(x);
  NAG_FREE(xlow);
  NAG_FREE(xmul);
  NAG_FREE(xupp);
  NAG_FREE(fstate);
  NAG_FREE(iafun);
  NAG_FREE(igfun);
  NAG_FREE(javar);
  NAG_FREE(jgvar);
  NAG_FREE(xstate);

  return exit_status;
}

static void NAG_CALL usrfun(Integer *status, Integer n, const double x[],
                            Integer needf, Integer nf, double f[],
                            Integer needg, Integer leng, double g[],
                            Nag_Comm *comm)
{
  /* Parameter adjustments */
#define X(I) x[(I) -1]
#define F(I) f[(I) -1]

  /* Check whether information came from the main program
     via the comm structure. Even if it was, we ignore it
     in this example. */
  if (comm->p)
    printf("Pointer %p was passed to usrfun via the comm struct\n", comm->p);

  /* Function Body */
  if (needf > 0) {
    F(1) = sin(-X(1) - .25) * 1e3 + sin(-X(2) - .25) * 1e3 - X(3);
    F(2) = sin(X(1) - .25) * 1e3 + sin(X(1) - X(2) - .25) * 1e3 - X(4);
    F(3) = sin(X(2) - X(1) - .25) * 1e3 + sin(X(2) - .25) * 1e3;
    F(4) = -X(1) + X(2);
    F(5) = X(1) - X(2);
    F(6) = X(3) * (X(3) * X(3)) * 1e-6 + X(4) * (X(4) * X(4)) * 2e-6 / 3.
           + X(3) * 3 + X(4) * 2;
  }

  return;
} /* usrfun */
```

10.2 Program Data

```
nag_opt_sparse_nlp_jacobian (e04vjc) Example Program Data
 4   6               : Values of n and nf
-0.55E0    0.55e0  : Bounds on the variables, XLOW(i), XUPP(i), for i = 1 to n
-0.55E0    0.55E0
 0.0E0   1200.0E0
 0.0E0   1200.0E0

 6                   : Value of objrow
 -894.8E0 -894.8E0 : Bounds on the functions, FLOW(i), FUPP(i), for i = 1 to nf
 -894.8E0 -894.8E0
-1294.8E0 -1294.8E0
-0.55E0    1.0E25
-0.55E0    1.0E25
-1.0E25    1.0E25
```

10.3 Program Results

```
nag_opt_sparse_nlp_jacobian (e04vjc) Example Program Results

NEA (the number of nonzero entries in A) =   4
  I     IAFUN(I)   JAVAR(I)          A(I)
----    --------   --------   -----------
  1         4         1      -1.0000e+00
  2         5         1       1.0000e+00
  3         4         2       1.0000e+00
  4         5         2      -1.0000e+00

NEG (the number of nonzero entries in G) =  10
  I     IGFUN(I)   JGVAR(I)
----    --------   --------
  1         1         1
  2         2         1
  3         3         1
  4         1         2
  5         2         2
  6         3         2
  7         6         3
  8         6         4
  9         1         3
 10         2         4

Final objective value =      5126.5
Optimal X =      0.12     -0.40    679.95   1026.07
```