NAG Library Function Document

nag_real_symm_sparse_eigensystem_iter (f12fbc)

Note: *this function uses* **optional arguments** *to define choices in the problem specification. If you wish to use* default *settings for all of the optional arguments, then the option setting function nag_real_symm_sparse_eigensystem_option (f12fdc) need not be called. If, however, you wish to reset some or all of the settings please refer to Section 11 in nag_real_symm_sparse_eigensystem_option (f12fdc) for a detailed description of the specification of the optional arguments.*

1 Purpose

nag_real_symm_sparse_eigensystem_iter (f12fbc) is an iterative solver in a suite of functions consisting of nag_real_symm_sparse_eigensystem_init (f12fac), nag_real_symm_sparse_eigensystem_iter (f12fbc), nag_real_symm_sparse_eigensystem_sol (f12fcc), nag_real_symm_sparse_eigensystem_option (f12fdc) and nag_real_symm_sparse_eigensystem_monit (f12fec). It is used to find some of the eigenvalues (and optionally the corresponding eigenvectors) of a standard or generalized eigenvalue problem defined by real symmetric matrices.

2 Specification

```
#include <nag.h>
#include <nagf12.h>

void nag_real_symm_sparse_eigensystem_iter (Integer *irevcm, double resid[],
     double v[], double **x, double **y, double **mx, Integer *nshift,
     double comm[], Integer icomm[], NagError *fail)
```

3 Description

The suite of functions is designed to calculate some of the eigenvalues, λ, (and optionally the corresponding eigenvectors, x) of a standard eigenvalue problem $Ax = \lambda x$, or of a generalized eigenvalue problem $Ax = \lambda Bx$ of order n, where n is large and the coefficient matrices A and B are sparse, real and symmetric. The suite can also be used to find selected eigenvalues/eigenvectors of smaller scale dense, real and symmetric problems.

nag_real_symm_sparse_eigensystem_iter (f12fbc) is a **reverse communication** function, based on the ARPACK routine **dsaupd**, using the Implicitly Restarted Arnoldi iteration method, which for symmetric problems reduces to a variant of the Lanczos method. The method is described in Lehoucq and Sorensen (1996) and Lehoucq (2001) while its use within the ARPACK software is described in great detail in Lehoucq *et al.* (1998). An evaluation of software for computing eigenvalues of sparse symmetric matrices is provided in Lehoucq and Scott (1996). This suite of functions offers the same functionality as the ARPACK software for real symmetric problems, but the interface design is quite different in order to make the option setting clearer and to simplify the interface of nag_real_symm_sparse_eigensystem_iter (f12fbc).

The setup function nag_real_symm_sparse_eigensystem_init (f12fac) must be called before nag_real_symm_sparse_eigensystem_iter (f12fbc), the reverse communication iterative solver. Options may be set for nag_real_symm_sparse_eigensystem_iter (f12fbc) by prior calls to the option setting function nag_real_symm_sparse_eigensystem_option (f12fdc) and a post-processing function nag_real_symm_sparse_eigensystem_sol (f12fcc) must be called following a successful final exit from nag_real_symm_sparse_eigensystem_iter (f12fbc). nag_real_symm_sparse_eigensystem_monit (f12fec), may be called following certain flagged, intermediate exits from nag_real_symm_sparse_eigensystem_iter (f12fbc) to provide additional monitoring information about the computation.

nag_real_symm_sparse_eigensystem_iter (f12fbc) uses **reverse communication**, i.e., it returns repeatedly to the calling program with the argument **irevcm** (see Section 5) set to specified values which require the calling program to carry out one of the following tasks:

– compute the matrix-vector product $y = \mathrm{OP}x$, where OP is defined by the computational mode;

– compute the matrix-vector product $y = Bx$;

– notify the completion of the computation;

– allow the calling program to monitor the solution.

The problem type to be solved (standard or generalized), the spectrum of eigenvalues of interest, the mode used (regular, regular inverse, shifted inverse, Buckling or Cayley) and other options can all be set using the option setting function nag_real_symm_sparse_eigensystem_option (f12fdc).

4 References

Lehoucq R B (2001) Implicitly restarted Arnoldi methods and subspace iteration *SIAM Journal on Matrix Analysis and Applications* **23** 551–562

Lehoucq R B and Scott J A (1996) An evaluation of software for computing eigenvalues of sparse nonsymmetric matrices *Preprint MCS-P547-1195* Argonne National Laboratory

Lehoucq R B and Sorensen D C (1996) Deflation techniques for an implicitly restarted Arnoldi iteration *SIAM Journal on Matrix Analysis and Applications* **17** 789–821

Lehoucq R B, Sorensen D C and Yang C (1998) *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods* SIAM, Philidelphia

5 Arguments

Note: this function uses **reverse communication.** Its use involves an initial entry, intermediate exits and re-entries, and a final exit, as indicated by the argument **irevcm**. Between intermediate exits and re-entries, **all arguments other than x and y must remain unchanged**.

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

On initial entry: **irevcm** $= 0$, otherwise an error condition will be raised.

On intermediate re-entry: must be unchanged from its previous exit value. Changing **irevcm** to any other value between calls will result in an error.

On intermediate exit: has the following meanings.

irevcm $= -1$
: The calling program must compute the matrix-vector product $y = \mathrm{OP}x$, where x is stored in **x** and the result y is placed in **y**.

irevcm $= 1$
: The calling program must compute the matrix-vector product $y = \mathrm{OP}x$. This is similar to the case **irevcm** $= -1$ except that the result of the matrix-vector product Bx (as required in some computational modes) has already been computed and is available in **mx**.

irevcm $= 2$
: The calling program must compute the matrix-vector product $y = Bx$, where x is stored in **x** and y is placed in **y**.

irevcm $= 3$
: Compute the **nshift** real and imaginary parts of the shifts where the real parts are to be placed in the first **nshift** locations of the array **y** and the imaginary parts are to be placed in the first **nshift** locations of the array **mx**. Only complex conjugate pairs of shifts may be applied and the pairs must be placed in consecutive locations. This value of **irevcm** will only arise if the optional argument **Supplied Shifts** is set in a prior call to nag_real_symm_sparse_eigensystem_option (f12fdc) which is intended for experienced users only; the default and recommended option is to use exact shifts (see Lehoucq *et al.* (1998) for details and guidance on the choice of shift strategies).

irevcm $= 4$

Monitoring step: a call to nag_real_symm_sparse_eigensystem_monit (f12fec) can now be made to return the number of Arnoldi iterations, the number of converged Ritz values, their real and imaginary parts, and the corresponding Ritz estimates.

On final exit: **irevcm** $= 5$: nag_real_symm_sparse_eigensystem_iter (f12fbc) has completed its tasks. The value of **fail** determines whether the iteration has been successfully completed, or whether errors have been detected. On successful completion nag_real_symm_sparse_eigensystem_sol (f12fcc) must be called to return the requested eigenvalues and eigenvectors (and/or Schur vectors).

Constraint: on initial entry, **irevcm** $= 0$; on re-entry **irevcm** must remain unchanged.

2: **resid**[dim] – double *Input/Output*

Note: the dimension, dim, of the array **resid** must be at least **n** (see nag_real_symm_sparse_eigensystem_init (f12fac)).

On initial entry: need not be set unless the option **Initial Residual** has been set in a prior call to nag_real_symm_sparse_eigensystem_option (f12fdc) in which case **resid** should contain an initial residual vector, possibly from a previous run.

On intermediate re-entry: must be unchanged from its previous exit. Changing **resid** to any other value between calls may result in an error exit.

On intermediate exit: contains the current residual vector.

On final exit: contains the final residual vector.

3: **v**[**n** $\times$ **ncv**] – double *Input/Output*

The ith element of the jth basis vector is stored in location **v**[**n** $\times (i-1)+j-1$], for $i = 1, 2, \ldots,$ **n** and $j = 1, 2, \ldots,$ **ncv**.

On initial entry: need not be set.

On intermediate re-entry: must be unchanged from its previous exit.

On intermediate exit: contains the current set of Arnoldi basis vectors.

On final exit: contains the final set of Arnoldi basis vectors.

4: **x** – double ** *Input/Output*

On initial entry: need not be set, it is used as a convenient mechanism for accessing elements of **comm**.

On intermediate re-entry: is not normally changed.

On intermediate exit: contains the vector x when **irevcm** returns the value -1, $+1$ or 2.

On final exit: does not contain useful data.

5: **y** – double ** *Input/Output*

On initial entry: need not be set, it is used as a convenient mechanism for accessing elements of **comm**.

On intermediate re-entry: must contain the result of $y = \mathrm{OP}x$ when **irevcm** returns the value -1 or $+1$. It must contain the real parts of the computed shifts when **irevcm** returns the value 3.

On intermediate exit: does not contain useful data.

On final exit: does not contain useful data.

6: **mx** – double ** *Input/Output*

On initial entry: need not be set, it is used as a convenient mechanism for accessing elements of **comm**.

On intermediate re-entry: it must contain the imaginary parts of the computed shifts when **irevcm** returns the value 3.

On intermediate exit: contains the vector Bx when **irevcm** returns the value $+1$.

On final exit: does not contain any useful data.

7: **nshift** – Integer * *Output*

On intermediate exit: if the option **Supplied Shifts** is set and **irevcm** returns a value of 3, **nshift** returns the number of complex shifts required.

8: **comm**[*dim*] – double *Communication Array*

Note: the dimension, *dim*, of the array **comm** must be at least $\max(1, \mathbf{lcomm})$ (see nag_real_symm_sparse_eigensystem_init (f12fac)).

On initial entry: must remain unchanged following a call to the setup function nag_real_symm_sparse_eigensystem_init (f12fac).

On exit: contains data defining the current state of the iterative process.

9: **icomm**[*dim*] – Integer *Communication Array*

Note: the dimension, *dim*, of the array **icomm** must be at least $\max(1, \mathbf{licomm})$ (see nag_real_symm_sparse_eigensystem_init (f12fac)).

On initial entry: must remain unchanged following a call to the setup function nag_real_symm_sparse_eigensystem_init (f12fac).

On exit: contains data defining the current state of the iterative process.

10: **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.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

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

NE_BOTH_ENDS_1

Eigenvalues from both ends of the spectrum were requested, but the number of eigenvalues (see **nev** in nag_real_symm_sparse_eigensystem_init (f12fac)) requested is one.

NE_INT

The maximum number of iterations ≤ 0, the option **Iteration Limit** has been set to ⟨*value*⟩.

NE_INTERNAL_EIGVAL_FAIL

Error in internal call to compute eigenvalues and corresponding error bounds of the current upper Hessenberg matrix. Please contact NAG.

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 3.6.6 in the Essential Introduction for further information.

NE_MAX_ITER

The maximum number of iterations has been reached. The maximum number of iterations $= \langle value \rangle$. The number of converged eigenvalues $= \langle value \rangle$. The post-processing function nag_real_symm_sparse_eigensystem_sol (f12fcc) may be called to recover the converged eigenvalues at this point. Alternatively, the maximum number of iterations may be increased by a call to the option setting function nag_real_symm_sparse_eigensystem_option (f12fdc) and the reverse communication loop restarted. A large number of iterations may indicate a poor choice for the values of **nev** and **ncv**; it is advisable to experiment with these values to reduce the number of iterations (see nag_real_symm_sparse_eigensystem_init (f12fac)).

NE_NO_LANCZOS_FAC

Could not build a Lanczos factorization. The size of the current Lanczos factorization $= \langle value \rangle$.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_NO_SHIFTS_APPLIED

No shifts could be applied during a cycle of the implicitly restarted Lanczos iteration.

NE_OPT_INCOMPAT

The options **Generalized** and **Regular** are incompatible.

NE_ZERO_INIT_RESID

The option **Initial Residual** was selected but the starting vector held in **resid** is zero.

7 Accuracy

The relative accuracy of a Ritz value, λ, is considered acceptable if its Ritz estimate $\leq$ **Tolerance** $\times |\lambda|$. The default **Tolerance** used is the ***machine precision*** given by nag_machine_precision (X02AJC).

8 Parallelism and Performance

nag_real_symm_sparse_eigensystem_iter (f12fbc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_real_symm_sparse_eigensystem_iter (f12fbc) 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

For this function two examples are presented, with a main program and two example problems given in Example 1 (ex1) and Example 2 (ex2).

Example 1 (ex1)

The example solves $Ax = \lambda x$ in shift-invert mode, where A is obtained from the standard central difference discretization of the one-dimensional Laplacian operator $\frac{\partial^2 u}{\partial x^2}$ with zero Dirichlet boundary conditions. Eigenvalues closest to the shift $\sigma = 0$ are sought.

Example 2 (ex2)

This example illustrates the use of nag_real_symm_sparse_eigensystem_iter (f12fbc) to compute the leading terms in the singular value decomposition of a real general matrix A. The example finds a few of the largest singular values (σ) and corresponding right singular values (ν) for the matrix A by solving the symmetric problem:

$$(A^{\mathrm{T}}A)\nu = \sigma\nu.$$

Here A is the m by n real matrix derived from the simplest finite difference discretization of the two-dimensional kernal $k(s,t)dt$ where

$$k(s,t) = \begin{cases} s(t-1) & \text{if } 0 \leq s \leq t \leq 1 \\ t(s-1) & \text{if } 0 \leq t < s \leq 1 \end{cases}.$$

Note: this formulation is appropriate for the case $m \geq n$. Reverse the rules of A and A^{T} in the case of $m < n$.

10.1 Program Text

```
/* nag_real_symm_sparse_eigensystem_iter (f12fbc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 8, 2005.
 */

#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <stdio.h>
#include <nagf12.h>
#include <nagf16.h>

static void my_dgttrf(Integer, double *, double *, double *,
                      double *, Integer *, Integer *);
static void my_dgttrs(Integer, double *, double *, double *,
                      double *, Integer *, double *, double *);
static void av(Integer, Integer, double *, double *);
static void atv(Integer, Integer, double *, double *);

static int ex1(void), ex2(void);

int main(void)
{
  Integer  exit_status_ex1 = 0;
  Integer  exit_status_ex2 = 0;

  printf("nag_real_symm_sparse_eigensystem_iter (f12fbc) Example "
         "Program Results\n");
  exit_status_ex1 = ex1();
  exit_status_ex2 = ex2();

  return (exit_status_ex1 == 0 && exit_status_ex2 == 0) ? 0 : 1;
}

int ex1(void)
{
  /* Constants */
  Integer  licomm = 140, imon = 0;
  /* Scalars */
  double   estnrm, h2, sigma;
  Integer  exit_status = 0, info, irevcm, j, lcomm, n, nconv, ncv;
```

```
  Integer  nev, niter, nshift;
  /* Nag types */
  NagError fail;
  /* Arrays */
  double   *dd = 0, *dl = 0, *du = 0, *du2 = 0, *comm = 0, *eigest = 0;
  double   *eigv = 0, *resid = 0, *v = 0;
  Integer  *icomm = 0, *ipiv = 0;
  /* Pointers */
  double   *mx = 0, *x = 0, *y = 0;

  INIT_FAIL(fail);

  printf("\nExample 1\n");
  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read values for nx, nev and cnv from data file. */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &n, &nev, &ncv);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &n, &nev, &ncv);
#endif

  /* Allocate memory */
  lcomm = 3*n + ncv*ncv + 8*ncv + 60;
  if (!(dd = NAG_ALLOC(n, double)) ||
      !(dl = NAG_ALLOC(n, double)) ||
      !(du = NAG_ALLOC(n, double)) ||
      !(du2 = NAG_ALLOC(n, double)) ||
      !(comm = NAG_ALLOC(lcomm, double)) ||
      !(eigv = NAG_ALLOC(ncv, double)) ||
      !(eigest = NAG_ALLOC(ncv, double)) ||
      !(resid = NAG_ALLOC(n, double)) ||
      !(v = NAG_ALLOC(n * ncv, double)) ||
      !(icomm = NAG_ALLOC(licomm, Integer)) ||
      !(ipiv = NAG_ALLOC(n, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  /* Initialise communication arrays for problem using
     nag_real_symm_sparse_eigensystem_init (f12fac). */
  nag_real_symm_sparse_eigensystem_init(n, nev, ncv, icomm, licomm, comm,
                                        lcomm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_real_symm_sparse_eigensystem_init "
             "(f12fac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* Select the required spectrum using
     nag_real_symm_sparse_eigensystem_option (f12fdc). */
  nag_real_symm_sparse_eigensystem_option("largest magnitude", icomm, comm,
                                          &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_real_symm_sparse_eigensystem_option "
             "(f12fdc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
```

```
    }
  /* Select the required mode */
  nag_real_symm_sparse_eigensystem_option("shifted inverse", icomm, comm,
                                          &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_real_symm_sparse_eigensystem_option "
             "(f12fdc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  h2 = 1.0 / (double)((n + 1) * (n + 1));
  sigma = 0.0;
  for (j = 0; j <= n-1; ++j)
    {
      dd[j] = 2.0 / h2 - sigma;
      dl[j] = -1.0 / h2;
      du[j] = dl[j];
    }
  my_dgttrf(n, dl, dd, du, du2, ipiv, &info);

  irevcm = 0;
 REVCOMLOOP:
  /* Repeated calls to reverse communication routine
     nag_real_symm_sparse_eigensystem_iter (f12fbc). */
  nag_real_symm_sparse_eigensystem_iter(&irevcm, resid, v, &x, &y, &mx,
                                        &nshift, comm, icomm, &fail);
  if (irevcm != 5)
    {
      if (irevcm == -1 || irevcm == 1)
        {
          /* Perform  y <--- OP*x = inv[A-SIGMA*I]*x. */
          /* Use my_dgttrs, a cut down C version of Lapack's dgttrs. */
          my_dgttrs(n, dl, dd, du, du2, ipiv, x, y);
        }
      else if (irevcm == 4 && imon == 1)
        {
          /* If imon=1, get monitoring information using
             nag_real_symm_sparse_eigensystem_monit (f12fec). */
          nag_real_symm_sparse_eigensystem_monit(&niter, &nconv, eigv, eigest,
                                                 icomm, comm);
          /* Compute 2-norm of Ritz estimates using
             nag_dge_norm (f16rac).*/
          nag_dge_norm(Nag_ColMajor, Nag_FrobeniusNorm, nev, 1, eigest, nev,
                       &estnrm, &fail);
          printf("Iteration %3"NAG_IFMT", ", niter);
          printf(" No. converged = %3"NAG_IFMT",", nconv);
          printf(" norm of estimates = %17.8e\n", estnrm);
        }
      goto REVCOMLOOP;
    }
  if (fail.code == NE_NOERROR)
    {
      /* Post-Process using nag_real_symm_sparse_eigensystem_sol
         (f12fcc) to compute eigenvalues/vectors. */
      nag_real_symm_sparse_eigensystem_sol(&nconv, eigv, v, sigma, resid, v,
                                           comm, icomm, &fail);
      printf("\n The %4"NAG_IFMT" Ritz values", nconv);
      printf(" closest to %8.4f are:\n\n", sigma);
      for (j = 0; j <= nconv-1; ++j)
        {
          printf("%8"NAG_IFMT"%5s%12.4f\n", j+1, "", eigv[j]);
        }
    }
  else
    {
      printf(" Error from "
             "nag_real_symm_sparse_eigensystem_iter (f12fbc).\n%s\n",
             fail.message);
      exit_status = 1;
```

```
      goto END;
    }
 END:
  NAG_FREE(dd);
  NAG_FREE(dl);
  NAG_FREE(du);
  NAG_FREE(du2);
  NAG_FREE(comm);
  NAG_FREE(eigv);
  NAG_FREE(eigest);
  NAG_FREE(resid);
  NAG_FREE(v);
  NAG_FREE(icomm);
  NAG_FREE(ipiv);

  return exit_status;
}

static void my_dgttrf(Integer n, double dl[], double d[],
                      double du[], double du2[], Integer ipiv[],
                      Integer *info)
{
  /* A simple C version of the Lapack routine dgttrf with argument
     checking removed */
  /* Scalars */
  double  temp, fact;
  Integer i;
  /* Function Body */
  *info = 0;
  for (i = 0; i < n; ++i)
    {
      ipiv[i] = i;
    }
  for (i = 0; i < n - 2; ++i)
    {
      du2[i] = 0.0;
    }
  for (i = 0; i < n - 2; i++)
    {
      if (fabs(d[i]) >= fabs(dl[i]))
        {
          /* No row interchange required, eliminate dl[i]. */
          if (d[i] != 0.0)
            {
              fact = dl[i] / d[i];
              dl[i] = fact;
              d[i+1] = d[i+1] - fact * du[i];
            }
        }
      else
        {
          /* Interchange rows I and I+1, eliminate dl[I] */
          fact = d[i] / dl[i];
          d[i] = dl[i];
          dl[i] = fact;
          temp = du[i];
          du[i] = d[i+1];
          d[i+1] = temp - fact*d[i+1];
          du2[i] = du[i+1];
          du[i+1] = -fact * du[i+1];
          ipiv[i] = i + 1;
        }
    }
  if (n > 1)
    {
      i = n - 2;
      if (fabs(d[i]) >= fabs(dl[i]))
        {
          if (d[i] != 0.0)
            {
              fact = dl[i] / d[i];
```

```
              dl[i] = fact;
              d[i+1] = d[i+1] - fact * du[i];
            }
        }
      else
        {
          fact = d[i] / dl[i];
          d[i] = dl[i];
          dl[i] = fact;
          temp = du[i];
          du[i] = d[i+1];
          d[i+1] = temp - fact * d[i+1];
          ipiv[i] = i + 1;
        }
    }
  /* Check for a zero on the diagonal of U. */
  for (i = 0; i < n; ++i)
    {
      if (d[i] == 0.0)
        {
          *info = i;
          goto END;
        }
    }
 END:
  return;
}

static void my_dgttrs(Integer n, double dl[], double d[],
                      double du[], double du2[], Integer ipiv[],
                      double b[], double y[])
{
  /* A simple C version of the Lapack routine dgttrs with argument
     checking removed, the number of right-hand-sides=1, Trans='N' */
  /* Scalars */
  Integer i, ip;
  double  temp;
  /* Solve L*x = b. */
  for (i = 0; i <= n - 1; ++i)
    {
      y[i] = b[i];
    }
  for (i = 0; i < n - 1; ++i)
    {
      ip = ipiv[i];
      temp = y[i+1-ip+i] - dl[i]*y[ip];
      y[i] = y[ip];
      y[i+1] = temp;
    }
  /* Solve U*x = b. */
  y[n-1] = y[n-1] / d[n-1];
  if (n > 1)
    {
      y[n-2] = (y[n-2] - du[n-2]*y[n-1])/d[n-2];
    }
  for (i = n - 3; i >= 0; --i)
    {
      y[i] = (y[i]-du[i]*y[i+1]-du2[i]*y[i+2])/d[i];
    }
  return;
}

int ex2(void)
{
  /* Constants */
  Integer  licomm = 140;
  /* Scalars */
  double   sigma = 0, axnorm;
  Integer  exit_status = 0, irevcm, j, lcomm, m, n, nconv, ncv;
  Integer  nev, nshift;
  NagError fail;
```

```
  /* Arrays */
  double   *comm = 0, *eigv = 0, *eigest = 0;
  double   *resid = 0, *v = 0, *ax = 0;
  Integer  *icomm = 0;
  /* Ponters */
  double   *mx = 0, *x = 0, *y = 0;

  INIT_FAIL(fail);

  printf("\nExample 2\n");
  /* Skip heading in data file. */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read values for m, n, nev and cnv from data file. */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"*[^\n] ",
         &m, &n, &nev, &ncv);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"*[^\n] ",
         &m, &n, &nev, &ncv);
#endif

  /* Allocate memory */
  lcomm = 3*n + ncv*ncv + 8*ncv + 60;
  if (!(comm = NAG_ALLOC(lcomm, double)) ||
      !(eigv = NAG_ALLOC(ncv, double)) ||
      !(eigest = NAG_ALLOC(ncv, double)) ||
      !(resid = NAG_ALLOC(n, double)) ||
      !(ax = NAG_ALLOC(m, double)) ||
      !(v = NAG_ALLOC(n * ncv, double)) ||
      !(icomm = NAG_ALLOC(licomm, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  /* Initialise communication arrays for problem using
     nag_real_symm_sparse_eigensystem_init (f12fac). */
  nag_real_symm_sparse_eigensystem_init(n, nev, ncv, icomm, licomm, comm,
                                        lcomm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_real_symm_sparse_eigensystem_init "
             "(f12fac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  irevcm = 0;
 REVCOMLOOP:
  /* Repeated calls to reverse communication routine
     nag_real_symm_sparse_eigensystem_iter (f12fbc). */
  nag_real_symm_sparse_eigensystem_iter(&irevcm, resid, v, &x, &y, &mx,
                                        &nshift, comm, icomm, &fail);
  if (irevcm != 5)
    {
      if (irevcm == -1 || irevcm == 1)
        {
          /* Perform matrix vector multiplication y <--- Op*x */
          av(m, n, x, ax);
          atv(m, n, ax, y);
        }
      goto REVCOMLOOP;
    }
  if (fail.code == NE_NOERROR)
    {
      /* Post-Process using nag_real_symm_sparse_eigensystem_sol
         (f12fcc) to compute singular values/vectors. */
```

```
      nag_real_symm_sparse_eigensystem_sol(&nconv, eigv, v, sigma, resid, v,
                                           comm, icomm, &fail);
      /* Singular values (squared) are returned in eigv and the
         corresponding right singular vectors are returned in the first
         nev n-length vectors in v. */
      printf("\n The %4"NAG_IFMT" leading Singular values and", nconv);
      printf(" direct residuals are:\n\n");
      for (j = 0; j <= nconv-1; ++j)
        {
          eigv[j] = sqrt(eigv[j]);
          /* Compute the left singular vectors from the formula
             u = Av/sigma
             u should have norm 1 so divide by norm(Av). */
          av(m, n, &v[j*n], ax);
          /* Compute 2-norm of Av using nag_dge_norm (f16rac).*/
          nag_dge_norm(Nag_ColMajor, Nag_FrobeniusNorm, m, 1, ax,
                       m, &axnorm, &fail);
          resid[j] = axnorm*fabs(1.0-eigv[j]/axnorm);

          printf("%8"NAG_IFMT"%5s%12.4f%5s%12.7f\n", j+1, "", eigv[j], "",
                 resid[j]);
        }
    }
  else
    {
      printf(" Error from "
              "nag_real_symm_sparse_eigensystem_iter (f12fbc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }
 END:
  NAG_FREE(comm);
  NAG_FREE(eigv);
  NAG_FREE(eigest);
  NAG_FREE(resid);
  NAG_FREE(v);
  NAG_FREE(ax);
  NAG_FREE(icomm);

  return exit_status;
}

static void av(Integer m, Integer n, double *x, double *w)
{
  /* Computes  w <- A*x. */
  /* Local Scalars */
  double  h, k, s, t;
  Integer i, j;
  h = 1.0/(double)(m+1);
  k = 1.0/(double)(n+1);
  for (i = 0; i < m; ++i)
    {
      w[i] = 0.0;
    }
  t = 0.0;

  for (j = 0; j < n; ++j)
    {
      t = t + k;
      s = 0.0;
      for (i = 0; i < j+1; i++)
        {
          s = s + h;
          w[i] = w[i] + k*s*(t-1.0)*x[j];
        }
      for (i = j+1; i < m; ++i)
        {
          s = s + h;
          w[i] = w[i] + k*t*(s-1.0)*x[j];
        }
```

```
    }
  return;
} /* av */

static void atv(Integer m, Integer n, double *x, double *y)
{
  /* Computes  y <- A'*w. */
  /* Local Scalars */
  double  h, k, s, t;
  Integer i, j;
  h = 1.0/(double)(m+1);
  k = 1.0/(double)(n+1);
  for (i = 0; i < n; ++i)
    {
      y[i] = 0.0;
    }
  t = 0.0;
  for (j = 0; j < n; ++j)
    {
      t = t + k;
      s = 0.0;
      for (i = 0; i < j+1; ++i)
        {
          s = s + h;
          y[j] = y[j] + k*s*(t-1.0)*x[i];
        }
      for (i = j+1; i < m; ++i)
        {
          s = s + h;
          y[j] = y[j] + k*t*(s-1.0)*x[i];
        }
    }
  return;
} /* atv */
```

10.2 Program Data

```
nag_real_symm_sparse_eigensystem_iter (f12fbc) Example Program Data
Example 1
 100  4  10 : Values for n, nev and ncv
Example 2
 500 100 4 10 : Values for m, n, nev and ncv
```

10.3 Program Results

```
nag_real_symm_sparse_eigensystem_iter (f12fbc) Example Program Results

Example 1

 The    4 Ritz values closest to   0.0000 are:

       1           9.8688
       2          39.4657
       3          88.7620
       4         157.7101

Example 2

 The    4 leading Singular values and direct residuals are:

       1           0.0410        0.0000000
       2           0.0605        0.0000000
       3           0.1178        0.0000000
       4           0.5572        0.0000000
```