NAG Library Function Document

nag_eigen_real_gen_sparse_arnoldi (f02ekc)

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, you need only read Sections 1 to 10 of this document. If, however, you wish to reset some or all of the settings this must be done by calling the option setting function nag_real_sparse_eigensystem_option (f12adc) from the user-supplied function* ***option****. Please refer to Section 11 for a detailed description of the specification of the optional arguments.*

1 Purpose

nag_eigen_real_gen_sparse_arnoldi (f02ekc) computes selected eigenvalues and eigenvectors of a real sparse general matrix.

2 Specification

```
#include <nag.h>
#include <nagf02.h>
void nag_eigen_real_gen_sparse_arnoldi (Integer n, Integer nnz, double a[],
     const Integer icolzp[], const Integer irowix[], Integer nev,
     Integer ncv, double sigma,
     void (*monit)(Integer ncv, Integer niter, Integer nconv,
          const Complex w[], const double rzest[], Integer *istat,
          Nag_Comm *comm),
     void (*option)(Integer icom[], double com[], Integer *istat,
          Nag_Comm *comm),
     Integer *nconv, Complex w[], double v[], Integer pdv, double resid[],
     Nag_Comm *comm, NagError *fail)
```

3 Description

nag_eigen_real_gen_sparse_arnoldi (f02ekc) computes selected eigenvalues and the corresponding right eigenvectors of a real sparse general matrix A:

$$Aw_i = \lambda_i w_i.$$

A specified number, n_{ev}, of eigenvalues λ_i, or the shifted inverses $\mu_i = 1/(\lambda_i - \sigma)$, may be selected either by largest or smallest modulus, largest or smallest real part, or, largest or smallest imaginary part. Convergence is generally faster when selecting larger eigenvalues, smaller eigenvalues can always be selected by choosing a zero inverse shift ($\sigma = 0.0$). When eigenvalues closest to a given real value are required then the shifted inverses of largest magnitude should be selected with shift equal to the required real value.

Note that even though A is real, λ_i and w_i may be complex. If w_i is an eigenvector corresponding to a complex eigenvalue λ_i, then the complex conjugate vector $\bar{w}_i$ is the eigenvector corresponding to the complex conjugate eigenvalue $\bar{\lambda}_i$. The eigenvalues in a complex conjugate pair λ_i and $\bar{\lambda}_i$ are either both selected or both not selected.

The sparse matrix A is stored in compressed column storage (CCS) format. See Section 2.1.3 in the f11 Chapter Introduction.

nag_eigen_real_gen_sparse_arnoldi (f02ekc) uses an implicitly restarted Arnoldi iterative method to converge approximations to a set of required eigenvalues and corresponding eigenvectors. Further algorithmic information is given in Section 9 while a fuller discussion is provided in the f12 Chapter Introduction. If shifts are to be performed then operations using shifted inverse matrices are performed using a direct sparse solver; further information on the solver used is provided in the f11 Chapter Introduction.

4 References

Golub G H and Van Loan C F (1996) *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

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

1: **n** – Integer *Input*

On entry: n, the order of the matrix A.

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

2: **nnz** – Integer *Input*

On entry: the dimension of the array **a**.The number of nonzero elements of the matrix A and, if a nonzero shifted inverse is to be applied, all diagonal elements. Each nonzero is counted once in the latter case.

Constraint: $0 \leq \mathbf{nnz} \leq \mathbf{n}^2$.

3: **a**[**nnz**] – double *Input/Output*

On entry: the array of nonzero elements (and diagonal elements if a nonzero inverse shift is to be applied) of the n by n general matrix A.

On exit: if a nonzero shifted inverse is to be applied then the diagonal elements of A have the shift value, as supplied in **sigma**, subtracted.

4: **icolzp**[**n** + **1**] – const Integer *Input*

On entry: **icolzp**$[i-1]$ contains the index in **a** of the start of column i, for $i = 1, 2, \ldots, n$; **icolzp**[**n**] must contain the value **nnz** + 1. Thus the number of nonzero elements in column i of A is **icolzp**$[i]$ − **icolzp**$[i-1]$; when shifts are applied this includes diagonal elements irrespective of value. See Section 2.1.3 in the f11 Chapter Introduction.

5: **irowix**[**nnz**] – const Integer *Input*

On entry: **irowix**$[i-1]$ contains the row index for each entry in **a**. See Section 2.1.3 in the f11 Chapter Introduction.

6: **nev** – Integer *Input*

On entry: the number of eigenvalues to be computed.

Constraint: $0 < \mathbf{nev} < \mathbf{n} - 1$.

7: **ncv** – Integer *Input*

On entry: the dimension of the array **w**. The number of Arnoldi basis vectors to use during the computation.

At present there is no *a priori* analysis to guide the selection of **ncv** relative to **nev**. However, it is recommended that $\mathbf{ncv} \geq 2 \times \mathbf{nev} + 1$. If many problems of the same type are to be solved, you should experiment with increasing **ncv** while keeping **nev** fixed for a given test problem. This will usually decrease the required number of matrix-vector operations but it also increases the work and storage required to maintain the orthogonal basis vectors. The optimal 'cross-over' with respect to CPU time is problem dependent and must be determined empirically.

Constraint: $\mathbf{nev} + 1 < \mathbf{ncv} \leq \mathbf{n}$.

8: **sigma** – double *Input*

On entry: if the **Shifted Inverse Real** mode has been selected then **sigma** contains the real shift used; otherwise **sigma** is not referenced. This mode can be selected by setting the appropriate options in the user-supplied function **option**.

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

monit is used to monitor the progress of nag_eigen_real_gen_sparse_arnoldi (f02ekc). **monit** may be specified as **NULLFN** if no monitoring is actually required. **monit** is called after the solution of each eigenvalue sub-problem and also just prior to return from nag_eigen_real_gen_sparse_arnoldi (f02ekc).

The specification of **monit** is:

```
void monit (Integer ncv, Integer niter, Integer nconv,
     const Complex w[], const double rzest[], Integer *istat,
     Nag_Comm *comm)
```

1: **ncv** – Integer *Input*

On entry: the dimension of the arrays **w** and **rzest**. The number of Arnoldi basis vectors used during the computation.

2: **niter** – Integer *Input*

On entry: the number of the current Arnoldi iteration.

3: **nconv** – Integer *Input*

On entry: the number of converged eigenvalues so far.

4: **w**[**ncv**] – const Complex *Input*

On entry: the first **nconv** elements of **w** contain the converged approximate eigenvalues.

5: **rzest**[**ncv**] – const double *Input*

On entry: the first **nconv** elements of **rzest** contain the Ritz estimates (error bounds) on the converged approximate eigenvalues.

6: **istat** – Integer * *Input/Output*

On entry: set to zero.

On exit: if set to a nonzero value nag_eigen_real_gen_sparse_arnoldi (f02ekc) returns immediately with **fail.code** = NE_USER_STOP.

7: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_eigen_real_gen_sparse_arnoldi (f02ekc) you may allocate memory and initialize these pointers with various quantities for use by **monit** when called from nag_eigen_real_gen_sparse_arnoldi (f02ekc) (see Section 3.2.1.1 in the Essential Introduction).

10: **option** – function, supplied by the user *External Function*

You can supply non-default options to the Arnoldi eigensolver by repeated calls to nag_real_sparse_eigensystem_option (f12adc) from within **option**. (Please note that it is only necessary to call nag_real_sparse_eigensystem_option (f12adc); no call to nag_real_sparse_eigensystem_init (f12aac) is required from within **option**.) For example, you can set the mode to **Shifted Inverse Real**, you can increase the **Iteration Limit** beyond its default and you can print varying levels of detail on the iterative process using **Print Level**.

If only the default options (including that the eigenvalues of largest magnitude are sought) are to be used then **option** may be specified as **NULLFN**. See Section 10 for an example of using **option** to set some non-default options.

The specification of **option** is:

```
void option (Integer icom[], double com[], Integer *istat,
     Nag_Comm *comm)
```

1: **icom**[**140**] – Integer *Communication Array*

On entry: contains details of the default option set. This array must be passed as argument **icomm** in any call to nag_real_sparse_eigensystem_option (f12adc).

On exit: contains data on the current options set which may be altered from the default set via calls to nag_real_sparse_eigensystem_option (f12adc).

2: **com**[**60**] – double *Communication Array*

On entry: contains details of the default option set. This array must be passed as argument **comm** in any call to nag_real_sparse_eigensystem_option (f12adc).

On exit: contains data on the current options set which may be altered from the default set via calls to nag_real_sparse_eigensystem_option (f12adc).

3: **istat** – Integer * *Input/Output*

On entry: set to zero.

On exit: if set to a nonzero value nag_eigen_real_gen_sparse_arnoldi (f02ekc) returns immediately with **fail**.**code** = NE_USER_STOP.

4: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_eigen_real_gen_sparse_arnoldi (f02ekc) you may allocate memory and initialize these pointers with various quantities for use by **option** when called from nag_eigen_real_gen_sparse_arnoldi (f02ekc) (see Section 3.2.1.1 in the Essential Introduction).

11: **nconv** – Integer * *Output*

On exit: the number of converged approximations to the selected eigenvalues. On successful exit, this will normally be either **nev** or **nev** + 1 depending on the number of complex conjugate pairs of eigenvalues returned.

12: **w**[**ncv**] – Complex *Output*

On exit: the first **nconv** elements contain the converged approximations to the selected eigenvalues. Since complex conjugate pairs of eigenvalues appear together, it is possible (given an

odd number of converged real eigenvalues) for nag_eigen_real_gen_sparse_arnoldi (f02ekc) to return one more eigenvalue than requested.

13: **v**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **v** must be at least **pdv** × **ncv**.

On exit: contains the eigenvectors associated with the eigenvalue λ_i, for $i = 1, 2, \ldots,$ **nconv** (stored in **w**). For a real eigenvalue, λ_j, the corresponding eigenvector is real and is stored in **v**$[(j-1) \times$ **pdv** $+ i - 1]$, for $i = 1, 2, \ldots, n$. For complex conjugate pairs of eigenvalues, $w_{j+1} = \bar{w}_j$, the real and imaginary parts of the corresponding eigenvectors are stored, respectively, in **v**$[(j-1) \times$ **pdv** $+ i - 1]$ and **v**$[j \times$ **pdv** $+ i - 1]$, for $i = 1, 2, \ldots, n$. The imaginary parts stored are for the first of the conjugate pair of eigenvectors; the other eigenvector in the pair is obtained by negating these imaginary parts.

14: **pdv** – Integer *Input*

On entry: the stride separating, in the array **v**, real and imaginary parts of elements of a conjugate pair of eigenvectors, or separating the elements of a real eigenvector from the corresponding real parts of the next eigenvector.

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

15: **resid**[**nev** + **1**] – double *Output*

On exit: the residual $\|Aw_i - \lambda_i w_i\|_2$ for the estimates to the eigenpair λ_i and w_i is returned in **resid**$[i-1]$, for $i = 1, 2, \ldots,$ **nconv**.

16: **comm** – Nag_Comm *

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

17: **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_DIAG_ELEMENTS

On entry, in shifted inverse mode, the jth diagonal element of A is not defined, for $j =$ ⟨*value*⟩.

NE_EIGENVALUES

The number of eigenvalues found to sufficient accuracy is zero.

NE_INT

On entry, **n** = ⟨*value*⟩.
Constraint: **n** > 0.

On entry, **nev** = ⟨*value*⟩.
Constraint: **nev** > 0.

On entry, **nnz** = ⟨*value*⟩.
Constraint: **nnz** > 0.

NE_INT_2

On entry, **ncv** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **ncv** ≤ **n**.

On entry, **ncv** = ⟨*value*⟩ and **nev** = ⟨*value*⟩.
Constraint: **ncv** > **nev** + 1.

On entry, **nnz** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **nnz** ≤ **n** × **n**.

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

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_EIGVEC_FAIL

In calculating eigenvectors, an internal call returned with an error.
Please contact NAG.

NE_INTERNAL_ERROR

An internal call to nag_real_sparse_eigensystem_iter (f12abc) returned with **fail.code** = NE_OPT_INCOMPAT.
This error should not occur. Please contact NAG.

An internal call to nag_real_sparse_eigensystem_sol (f12acc) returned with **fail.code** = NE_INVALID_OPTION.

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.

Internal inconsistency in the number of converged Ritz values. Number counted = ⟨*value*⟩, number expected = ⟨*value*⟩.

NE_INVALID_OPTION

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

NE_NO_ARNOLDI_FAC

Could not build an Arnoldi factorization. The size of the current Arnoldi factorization = ⟨*value*⟩.

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 Arnoldi iteration.

NE_SCHUR_EIG_FAIL

During calculation of a real Schur form, there was a failure to compute ⟨*value*⟩ eigenvalues in a total of ⟨*value*⟩ iterations.

NE_SCHUR_REORDER

The computed Schur form could not be reordered by an internal call.
This routine returned with **fail.code** = ⟨*value*⟩.
Please contact NAG.

NE_SINGULAR

On entry, the matrix $A - \sigma \times I$ is nearly numerically singular and could not be inverted. Try perturbing the value of σ. Norm of matrix = ⟨*value*⟩, Reciprocal condition number = ⟨*value*⟩.

On entry, the matrix $A - \sigma \times I$ is numerically singular and could not be inverted. Try perturbing the value of σ.

NE_SPARSE_COL

On entry, for $i = \langle value \rangle$, **icolzp**$[i-1] = \langle value \rangle$ and **icolzp**$[i] = \langle value \rangle$.
Constraint: **icolzp**$[i-1] <$ **icolzp**$[i]$.

On entry, **icolzp**$[0] = \langle value \rangle$.
Constraint: **icolzp**$[0] = 1$.

On entry, **icolzp**[**n**] = ⟨*value*⟩ and **nnz** = ⟨*value*⟩.
Constraint: **icolzp**[**n**] = **nnz** + 1.

NE_SPARSE_ROW

On entry, in specification of column ⟨*value*⟩, and for $j = \langle value \rangle$, **irowix**$[j-1] = \langle value \rangle$ and **irowix**$[j] = \langle value \rangle$.
Constraint: **irowix**$[j-1] <$ **irowix**$[j]$.

NE_TOO_MANY_ITER

The maximum number of iterations has been reached.
The maximum number of iterations = ⟨*value*⟩.
The number of converged eigenvalues = ⟨*value*⟩.
See the function document for further details.

NE_USER_STOP

User requested termination in **monit**, **istat** = ⟨*value*⟩.

User requested termination in **option**, **istat** = ⟨*value*⟩.

NE_ZERO_RESID

An internal call to nag_real_sparse_eigensystem_iter (f12abc) returned with **fail.code** = NE_ZERO_INIT_RESID.

7 Accuracy

The relative accuracy of a Ritz value (eigenvalue approximation), λ , is considered acceptable if its Ritz estimate $\leq$ **Tolerance** $\times \lambda$. The default value for **Tolerance** is the ***machine precision*** given by nag_machine_precision (X02AJC). The Ritz estimates are available via the **monit** function at each iteration in the Arnoldi process, or can be printed by setting option **Print Level** to a positive value.

8 Parallelism and Performance

nag_eigen_real_gen_sparse_arnoldi (f02ekc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_eigen_real_gen_sparse_arnoldi (f02ekc) 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

nag_eigen_real_gen_sparse_arnoldi (f02ekc) calls functions based on the ARPACK suite in Chapter f12. These functions use an implicitly restarted Arnoldi iterative method to converge to approximations to a set of required eigenvalues (see the f12 Chapter Introduction).

In the default **Regular** mode, only matrix-vector multiplications are performed using the sparse matrix A during the Arnoldi process. Each iteration is therefore cheap computationally, relative to the alternative, **Shifted Inverse Real**, mode described below. It is most efficient (i.e., the total number of iterations required is small) when the eigenvalues of largest magnitude are sought and these are distinct.

Although there is an option for returning the smallest eigenvalues using this mode (see **Smallest Magnitude** option), the number of iterations required for convergence will be far greater or the method may not converge at all. However, where convergence is achieved, **Regular** mode may still prove to be the most efficient since no inversions are required. Where smallest eigenvalues are sought and **Regular** mode is not suitable, or eigenvalues close to a given real value are sought, the **Shifted Inverse Real** mode should be used.

If the **Shifted Inverse Real** mode is used (via a call to nag_real_sparse_eigensystem_option (f12adc) in **option**) then the matrix $A - \sigma I$ is used in linear system solves by the Arnoldi process. This is first factorized internally using the direct LU factorization function nag_superlu_lu_factorize (f11mec). The condition number of $A - \sigma I$ is then calculated by a call to nag_superlu_condition_number_lu (f11mgc). If the condition number is too big then the matrix is considered to be nearly singular, i.e., σ is an approximate eigenvalue of A, and the function exits with an error. In this situation it is normally sufficient to perturb σ by a small amount and call nag_eigen_real_gen_sparse_arnoldi (f02ekc) again. After successful factorization, subsequent solves are performed by calls to nag_superlu_solve_lu (f11mfc).

Finally, nag_eigen_real_gen_sparse_arnoldi (f02ekc) transforms the eigenvectors. Each eigenvector w (real or complex) is normalized so that $\|w\|_2 = 1$, and the element of largest absolute value is real and positive.

The monitoring function **monit** provides some basic information on the convergence of the Arnoldi iterations. Much greater levels of detail on the Arnoldi process are available via option **Print Level**. If this is set to a positive value then information will be printed, by default, to standard output. The **Monitoring** option may be used to select a monitoring ***file*** by setting the option to a file identification (unit) number associated with **Monitoring** (see nag_open_file (x04acc)).

10 Example

This example computes the four eigenvalues of the matrix A which lie closest to the value $\sigma = 5.5$ on the real line, and their corresponding eigenvectors, where A is the tridiagonal matrix with elements

$$a_{ij} = \begin{cases} 2 + i, & j = i \\ 3, & j = i - 1 \\ -1 + \rho/(2n + 2), & j = i + 1 \text{ with } \rho = 10.0. \end{cases}$$

10.1 Program Text

```
/* nag_eigen_real_gen_sparse_arnoldi (f02ekc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 25, 2014.
 */

#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <naga02.h>
#include <nagf02.h>
#include <nagf12.h>
#include <nagx02.h>
#include <nagx04.h>

/* User-defined Functions */
#ifdef __cplusplus
extern "C" {
#endif
static void NAG_CALL myoption(Integer icomm[], double com[], Integer *istat,
                              Nag_Comm *comm);

static void NAG_CALL mymonit(Integer ncv, Integer niter, Integer nconv,
                             const Complex w[], const double rzest[],
                             Integer *istat, Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

int main(void)
{
#define V(I, J) v[(J-1)*tdv + I-1]

  /* Scalars */
  double   one = 1.0, two = 2.0, three = 3.0;
  double   h, rho, s, sigma;
  Integer  exit_status = 0;
  Integer  fileid, fmode, i, imon, k, maxit, mode;
  Integer  n, nconv, ncv, nev, nnz, nx, prtlvl, tdv;

  /* Local Arrays */
  Complex  *w = 0;
  double   *a = 0, *resid = 0, *v = 0;
  double   user[1];
  Integer  *icolzp = 0, *irowix = 0;
  Integer  iuser[5];
  const char *filename = "f02ekce.monit";

  /* Nag Types */
  Nag_Comm comm;
  NagError fail;

  INIT_FAIL(fail);

  comm.user = user;
  comm.iuser = iuser;
  user[0] = 0.0;
  iuser[0] = 0;

  /* Output preamble */
  printf("nag_eigen_real_gen_sparse_arnoldi (f02ekc) ");
  printf("Example Program Results\n\n");
  fflush(stdout);

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

```
  scanf("%*[^\n] ");
#endif

  /* Read in problem size and parameters */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]%"NAG_IFMT"", &nx, &nev, &ncv);
#else
  scanf("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]%"NAG_IFMT"", &nx, &nev, &ncv);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n]%lf%*[^\n]%lf%*[^\n]", &rho, &sigma);
#else
  scanf("%*[^\n]%lf%*[^\n]%lf%*[^\n]", &rho, &sigma);
#endif

  n = nx * nx;
  nnz = 3 * n - 2;
  tdv = n;

  if (!(resid  = NAG_ALLOC((ncv),   double)) ||
      !(a      = NAG_ALLOC((nnz),   double)) ||
      !(icolzp = NAG_ALLOC((n + 1), Integer)) ||
      !(irowix = NAG_ALLOC((nnz),   Integer)) ||
      !(w      = NAG_ALLOC((ncv),   Complex)) ||
      !(v      = NAG_ALLOC((tdv)*(ncv), double))
      )
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Construct  A in compressed column storage (CCS) format where:
   *   A_{i,i}   = 2 + i
   *   A_{i+1,i) = 3
   *   A_{i,i+1} = rho/(2n+2) - 1
   */
  h = one/(double) (n + 1);
  s = rho * h/two - one;
  a[0] = two + one;
  a[1] = three;
  icolzp[0] = 1;
  irowix[0] = 1;
  irowix[1] = 2;

  k = 3;
  for (i = 2; i <= n-1; i++) {
    icolzp[i - 1] = k;
    irowix[k - 1] = i - 1;
    irowix[k]     = i;
    irowix[k + 1] = i + 1;
    a[k - 1] = s;
    a[k]     = two + (double) (i);
    a[k + 1] = three;
    k = k + 3;
  }

  icolzp[n - 1] = k;
  icolzp[n]     = k + 2;
  irowix[k - 1] = n - 1;
  irowix[k]     = n;
  a[k - 1] = s;
  a[k]     = two + (double) (n);

  /* Set some options via iuser array and routine argument OPTION.
   * iuser[0] = print level, iuser[1] = iteration limit,
   * iuser[2]>0 means shifted-invert mode
   * iuser[3]>0 means print monitoring info.
   */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]", &prtlvl, &maxit);
```

```
#else
  scanf("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]", &prtlvl, &maxit);
#endif
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]", &mode, &imon);
#else
  scanf("%"NAG_IFMT"%*[^\n]%"NAG_IFMT"%*[^\n]", &mode, &imon);
#endif

  if (imon>0) {
    /* Open the monitoring file for writing using
     * nag_open_file (x04acc):
     *   open unit number for reading, writing or appending, and
     *   associate unit with named file
     */
    /* If prtlvl >=10 internal monitoring information in addition to whatever is
     * written to fileid using mymonit.
     */
    fmode = 1;
    nag_open_file(filename, fmode, &fileid, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_open_file (x04acc) %s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    iuser[4] = fileid;
  }

  iuser[0] = prtlvl;
  iuser[1] = maxit;
  iuser[2] = mode;
  iuser[3] = imon;

  /* Compute eigenvalues and eigenvectors using
   * nag_eigen_real_gen_sparse_arnoldi (f02ekc):
   *   selected eigenvalues of real general matrix driver.
   */
  nag_eigen_real_gen_sparse_arnoldi(n, nnz, a, icolzp, irowix, nev, ncv, sigma,
                                    mymonit, myoption, &nconv, w, v, tdv,
                                    resid, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_eigen_real_gen_sparse_arnoldi (f02ekc)\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  if (imon>0) {
    /* Close the monitoring file using
     * nag_close_file (x04adc):
     *   close file associated with given unit number
     */
    nag_close_file(fileid, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_close_file (x04adc) %s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  }

  printf(" The %4"NAG_IFMT" ", nconv);
  printf(" Ritz values of closest to %13.5e are \n", sigma);
  for (i = 1; i <= nconv; i++) {
    /* nag_machine_precision (x02ajc) */
    if (resid[i-1] > (double) (100 * n) * nag_machine_precision) {
      printf("%7"NAG_IFMT"  ( %13.5e, %13.5e) ", i, w[i-1].re, w[i-1].im);
      printf("%13.5e\n", resid[i-1]);
    } else {
      printf("%8"NAG_IFMT"  ( %13.5e, %13.5e) \n", i, w[i-1].re, w[i-1].im);
    }
  }
```

```
 END:

  NAG_FREE(w);
  NAG_FREE(a);
  NAG_FREE(v);
  NAG_FREE(resid);
  NAG_FREE(icolzp);
  NAG_FREE(irowix);
  return exit_status;
}

static void NAG_CALL myoption(Integer icomm[], double com[], Integer *istat,
                              Nag_Comm *comm)
{
  NagError fail1;
  char     rec[26];

  INIT_FAIL(fail1);

  if (comm->iuser[0] > 0) {
#ifdef _WIN32
    sprintf_s(rec, _countof(rec), "Print Level=%5"NAG_IFMT, comm->iuser[0]);
#else
    sprintf(rec, "Print Level=%5"NAG_IFMT, comm->iuser[0]);
#endif
    fail1.code = 1;
    /* Set print level using
     * nag_real_sparse_eigensystem_option (f12adc)
     * Set a single option from a string.
     */
    nag_real_sparse_eigensystem_option(rec, icomm, com, &fail1);
    *istat = MAX(*istat, fail1.code);
  }

  if (comm->iuser[1] > 100) {
#ifdef _WIN32
    sprintf_s(rec, _countof(rec), "Iteration Limit=%5"NAG_IFMT, comm->iuser[1]);
#else
    sprintf(rec, "Iteration Limit=%5"NAG_IFMT, comm->iuser[1]);
#endif
    fail1.code = 1;
    /* Set iteration limit using
     * nag_real_sparse_eigensystem_option (f12adc)
     * Set a single option from a string.
     */
    nag_real_sparse_eigensystem_option(rec, icomm, com, &fail1);
    *istat = MAX(*istat, fail1.code);
  }

  if (comm->iuser[2] > 0) {
    fail1.code = 1;
    /* Set computational mode to shifted inverse real. */
    nag_real_sparse_eigensystem_option("Shifted Inverse Real", icomm, com,
                                       &fail1);
    *istat = MAX(*istat, fail1.code);
  }

  if (comm->iuser[3] > 0) {
    fail1.code = 1;
    /* Switch monitoring on and use the fileid stored in iuser[4]. */
#ifdef _WIN32
    sprintf_s(rec, _countof(rec), "Monitoring=%5"NAG_IFMT, comm->iuser[4]);
#else
    sprintf(rec, "Monitoring=%5"NAG_IFMT, comm->iuser[4]);
#endif
    nag_real_sparse_eigensystem_option(rec, icomm, com, &fail1);
    *istat = MAX(*istat, fail1.code);
  }
}
```

```
static void NAG_CALL mymonit(Integer ncv, Integer niter, Integer nconv,
                             const Complex w[], const double rzest[],
                             Integer *istat, Nag_Comm *comm)
{
  Integer i;
  char    line[100];

  if (comm->iuser[3] > 0) {

    /* Write lines to the file we opened for monitoring using
     * nag_write_line (x04bac):
     *   write formatted record to external file.
     */

    if (niter == 1 && comm->iuser[2] > 0) {
#ifdef _WIN32
      sprintf_s(line, _countof(line), " Arnoldi basis vectors used: %4"NAG_IFMT
                "\n", ncv);
#else
      sprintf(line, " Arnoldi basis vectors used: %4"NAG_IFMT"\n", ncv);
#endif
      nag_write_line(comm->iuser[4], line);
#ifdef _WIN32
      sprintf_s(line, _countof(line), " The following Ritz values (mu) are "
                "related to the\n");
#else
      sprintf(line, " The following Ritz values (mu) are related to the\n");
#endif
      nag_write_line(comm->iuser[4], line);
#ifdef _WIN32
      sprintf_s(line, _countof(line), " true eigenvalues (lambda) by lambda = "
                "sigma + 1/mu\n");
#else
      sprintf(line, " true eigenvalues (lambda) by lambda = sigma + 1/mu\n");
#endif
      nag_write_line(comm->iuser[4], line);
    }

#ifdef _WIN32
    sprintf_s(line, _countof(line), "\n Iteration number %4"NAG_IFMT"\n",
              niter);
#else
    sprintf(line, "\n Iteration number %4"NAG_IFMT"\n", niter);
#endif
    nag_write_line(comm->iuser[4], line);
#ifdef _WIN32
    sprintf_s(line, _countof(line), " Ritz values converged so far "
              "(%4"NAG_IFMT") and their Ritz estimates:\n", nconv);
#else
    sprintf(line, " Ritz values converged so far (%4"NAG_IFMT") and their Ritz "
            "estimates:\n", nconv);
#endif
    nag_write_line(comm->iuser[4], line);

    for (i = 1; i <= nconv; i++) {
#ifdef _WIN32
      sprintf_s(line, _countof(line),
                "  %4"NAG_IFMT" (%13.5e,%13.5e) %13.5e\n",
                i, w[i-1].re, w[i-1].im, rzest[i-1]);
#else
      sprintf(line,
              "  %4"NAG_IFMT" (%13.5e,%13.5e) %13.5e\n",
              i, w[i-1].re, w[i-1].im, rzest[i-1]);
#endif
      nag_write_line(comm->iuser[4], line);
    }

#ifdef _WIN32
    sprintf_s(line, _countof(line), " Next (unconverged) Ritz value:\n");
#else
    sprintf(line, " Next (unconverged) Ritz value:\n");
```

```
#endif
    nag_write_line(comm->iuser[4], line);
#ifdef _WIN32
    sprintf_s(line, _countof(line),
              "  %4"NAG_IFMT" (%13.5e,%13.5e)\n",
              nconv + 1, w[nconv].re, w[nconv].im);
#else
    sprintf(line,
            "  %4"NAG_IFMT" (%13.5e,%13.5e)\n",
            nconv + 1, w[nconv].re, w[nconv].im);
#endif
    nag_write_line(comm->iuser[4], line);
  }
  *istat = 0;
}
```

10.2 Program Data

```
nag_eigen_real_gen_sparse_arnoldi (f02ekc) Example Program Data
 10      : nx, matrix order n = nx*nx
 4       : nev, number of eigenvalues requested
 20      : ncv, size of subspace
 10.0    : rho, parameter for determining A
 5.5     : sigma, shift (want eigenvalues close to sigma)
 0       : print level
 500     : maximum number of iterations
 1       : mode (0 = regular, 1 = shifted inverse)
 0       : imon (0 = no monitoring, 1 = monitoring on)
```

10.3 Program Results

```
nag_eigen_real_gen_sparse_arnoldi (f02ekc) Example Program Results

 The    5  Ritz values of closest to   5.50000e+00 are
       1  (   6.01781e+00,  -8.00277e-01)
       2  (   6.01781e+00,   8.00277e-01)
       3  (   4.34309e+00,  -1.94557e+00)
       4  (   4.34309e+00,   1.94557e+00)
       5  (   7.14535e+00,   0.00000e+00)
```

11 Optional Arguments

Internally nag_eigen_real_gen_sparse_arnoldi (f02ekc) calls functions from the suite nag_real_sparse_eigensystem_init (f12aac), nag_real_sparse_eigensystem_iter (f12abc), nag_real_sparse_eigensystem_sol (f12acc), nag_real_sparse_eigensystem_option (f12adc) and nag_real_sparse_eigensystem_monit (f12aec). Several optional arguments for these computational functions define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_eigen_real_gen_sparse_arnoldi (f02ekc) these optional arguments are also used here and have associated *default values* that are usually appropriate. Therefore, you need only specify those optional arguments whose values are to be different from their default values.

Optional arguments may be specified via the user-supplied function **option** in the call to nag_eigen_real_gen_sparse_arnoldi (f02ekc). **option** must be coded such that one call to nag_real_sparse_eigensystem_option (f12adc) is necessary to set each optional argument. All optional arguments you do not specify are set to their default values.

The remainder of this section can be skipped if you wish to use the default values for all optional arguments.

The following is a list of the optional arguments available. A full description of each optional argument is provided in Section 11.1.

Advisory

Defaults

Iteration Limit

Largest Imaginary
Largest Magnitude
Largest Real
List
Monitoring
Nolist
Print Level
Regular
Shifted Inverse Real
Smallest Imaginary
Smallest Magnitude
Smallest Real
Tolerance

11.1 Description of the Optional Arguments

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

The summary line contains:

the keywords, where the minimum abbreviation of each keyword is underlined;

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

the default value, where the symbol ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)).

Keywords and character values are case and white space insensitive.

Optional arguments used to specify files (e.g., **Advisory** and **Monitoring**) have type Integer. This Integer value corresponds with the Nag_FileID as returned by nag_open_file (x04acc). See Section 10 for an example of the use of this facility.

Advisory i Default $= 0$

If the optional argument **List** is set then optional argument specifications are listed in a **List** ***file*** by setting the option to a file identification (unit) number associated with **Advisory** messages (see nag_open_file (x04acc)).

Defaults

This special keyword may be used to reset all optional arguments to their default values.

Iteration Limit i Default $= 300$

The limit on the number of Arnoldi iterations that can be performed before nag_eigen_real_gen_sparse_arnoldi (f02ekc) exits with **fail.code** = NE_TOO_MANY_ITER.

Largest Magnitude Default
Largest Imaginary
Largest Real
Smallest Imaginary
Smallest Magnitude
Smallest Real

The Arnoldi iterative method converges on a number of eigenvalues with given properties. The default is to compute the eigenvalues of largest magnitude using **Largest Magnitude**. Alternatively, eigenvalues

may be chosen which have **Largest Real** part, **Largest Imaginary** part, **Smallest Magnitude**, **Smallest Real** part or **Smallest Imaginary** part.

Note that these options select the eigenvalue properties for eigenvalues of OP the linear operator determined by the computational mode and problem type.

Nolist Default
List

Normally each optional argument specification is not printed to **Advisory** as it is supplied. Optional argument **List** may be used to enable printing and optional argument **Nolist** may be used to suppress the printing.

Monitoring i Default $= -1$

Unless **Monitoring** is set to -1 (the default), monitoring information is output to the file associated with Nag_FileID i during the solution of each problem; this may be the same as **Advisory**. The type of information produced is dependent on the value of **Print Level**, see the description of the optional argument **Print Level** in this section for details of the information produced. Please see nag_open_file (x04acc) to associate a file with a given Nag_FileID (see Section 3.2.1.1 in the Essential Introduction).

Print Level i Default $= 0$

This controls the amount of printing produced by nag_eigen_real_gen_sparse_arnoldi (f02ekc) as follows.

$= 0$ No output except error messages.

> 0 The set of selected options.

$= 2$ Problem and timing statistics when all calls to nag_real_sparse_eigensystem_iter (f12abc) have been completed.

≥ 5 A single line of summary output at each Arnoldi iteration.

≥ 10 If **Monitoring** is set to a valid Nag_FileID then at each iteration, the length and additional steps of the current Arnoldi factorization and the number of converged Ritz values; during re-orthogonalization, the norm of initial/restarted starting vector.

≥ 20 Problem and timing statistics on final exit from nag_real_sparse_eigensystem_iter (f12abc). If **Monitoring** is set to a valid Nag_FileID then at each iteration, the number of shifts being applied, the eigenvalues and estimates of the Hessenberg matrix H, the size of the Arnoldi basis, the wanted Ritz values and associated Ritz estimates and the shifts applied; vector norms prior to and following re-orthogonalization.

≥ 30 If **Monitoring** is set to a valid Nag_FileID then on final iteration, the norm of the residual; when computing the Schur form, the eigenvalues and Ritz estimates both before and after sorting; for each iteration, the norm of residual for compressed factorization and the compressed upper Hessenberg matrix H; during re-orthogonalization, the initial/restarted starting vector; during the Arnoldi iteration loop, a restart is flagged and the number of the residual requiring iterative refinement; while applying shifts, the indices of the shifts being applied.

≥ 40 If **Monitoring** is set to a valid Nag_FileID then during the Arnoldi iteration loop, the Arnoldi vector number and norm of the current residual; while applying shifts, key measures of progress and the order of H; while computing eigenvalues of H, the last rows of the Schur and eigenvector matrices; when computing implicit shifts, the eigenvalues and Ritz estimates of H.

≥ 50 If **Monitoring** is set to a valid Nag_FileID then during Arnoldi iteration loop: norms of key components and the active column of H, norms of residuals during iterative refinement, the final upper Hessenberg matrix H; while applying shifts: number of shifts, shift values, block indices, updated matrix H; while computing eigenvalues of H: the matrix H, the computed eigenvalues and Ritz estimates.

Regular Default
Shifted Inverse Real

These options define the computational mode which in turn defines the form of operation OP(x) to be performed.

Given a standard eigenvalue problem in the form $Ax = \lambda x$ then the following modes are available with the appropriate operator OP(x).

Regular	OP $= A$
Shifted Inverse Real	OP $= (A - \sigma I)^{-1}$ where σ is real

Tolerance r Default $= \epsilon$

An approximate eigenvalue has deemed to have converged when the corresponding Ritz estimate is within **Tolerance** relative to the magnitude of the eigenvalue.
