NAG Library Function Document

nag_complex_sparse_eigensystem_iter (f12apc)

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_complex_sparse_eigensystem_option (f12arc) need not be called. If, however, you wish to reset some or all of the settings please refer to Section 11 in nag_complex_sparse_eigensystem_option (f12arc) for a detailed description of the specification of the optional arguments.*

1 Purpose

nag_complex_sparse_eigensystem_iter (f12apc) is an iterative solver in a suite of functions consisting of nag_complex_sparse_eigensystem_init (f12anc), nag_complex_sparse_eigensystem_iter (f12apc), nag_complex_sparse_eigensystem_sol (f12aqc), nag_complex_sparse_eigensystem_option (f12arc) and nag_complex_sparse_eigensystem_monit (f12asc). It is used to find some of the eigenvalues (and optionally the corresponding eigenvectors) of a standard or generalized eigenvalue problem defined by complex nonsymmetric matrices.

2 Specification

```
#include <nag.h>
#include <nagf12.h>
void nag_complex_sparse_eigensystem_iter (Integer *irevcm, Complex resid[],
     Complex v[], Complex **x, Complex **y, Complex **mx, Integer *nshift,
     Complex 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, complex and nonsymmetric. The suite can also be used to find selected eigenvalues/eigenvectors of smaller scale dense, complex and nonsymmetric problems.

nag_complex_sparse_eigensystem_iter (f12apc) is a **reverse communication** function, based on the ARPACK routine **znaupd**, using the Implicitly Restarted Arnoldi iteration 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 nonsymmetric matrices is provided in Lehoucq and Scott (1996). This suite of functions offers the same functionality as the ARPACK software for complex nonsymmetric problems, but the interface design is quite different in order to make the option setting clearer and to simplify the interface of nag_complex_sparse_eigensystem_iter (f12apc).

The setup function nag_complex_sparse_eigensystem_init (f12anc) must be called before nag_complex_sparse_eigensystem_iter (f12apc), the reverse communication iterative solver. Options may be set for nag_complex_sparse_eigensystem_iter (f12apc) by prior calls to the option setting function nag_complex_sparse_eigensystem_option (f12arc) and a post-processing function nag_complex_sparse_eigensystem_sol (f12aqc) must be called following a successful final exit from nag_complex_sparse_eigensystem_iter (f12apc). nag_complex_sparse_eigensystem_monit (f12asc) may be called following certain flagged intermediate exits from nag_complex_sparse_eigensystem_iter (f12apc) to provide additional monitoring information about the computation.

nag_complex_sparse_eigensystem_iter (f12apc) 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, shifted real or shifted imaginary) and other options can all be set using the option setting function nag_complex_sparse_eigensystem_option (f12arc) (see Section 11.1 in nag_complex_sparse_eigensystem_option (f12arc) for details on setting options and of the default settings).

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** complex shifts to be placed in the first **nshift** locations of the array **y**. This value of **irevcm** will only arise if the optional argument **Supplied Shifts** is set in a prior call to nag_complex_sparse_eigensystem_option (f12arc) which is intended for experienced users only; the default and recommended option is to use exact shifts (see Lehoucq *et al.* (1998) for details).

irevcm $= 4$
Monitoring step: a call to nag_complex_sparse_eigensystem_monit (f12asc) can now be made to return the number of Arnoldi iterations, the number of converged Ritz values, the array of converged values, and the corresponding Ritz estimates.

On final exit: **irevcm** $= 5$: nag_complex_sparse_eigensystem_iter (f12apc) has completed its tasks. The value of **fail.code** determines whether the iteration has been successfully completed, or whether errors have been detected. On successful completion nag_complex_sparse_eigensystem_sol (f12aqc) 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**$[dim]$ – Complex *Input/Output*

Note: the dimension, dim, of the array **resid** must be at least **n** (see nag_complex_sparse_eigensystem_init (f12anc)).

On initial entry: need not be set unless the option **Initial Residual** has been set in a prior call to nag_complex_sparse_eigensystem_option (f12arc) 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**$[\mathbf{n} \times \mathbf{ncv}]$ – Complex *Input/Output*

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

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** – Complex ** *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** – Complex ** *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 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** – Complex ** *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 $y = Bx$ when **irevcm** returns the value 2.

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] – Complex *Communication Array*

Note: the dimension, *dim*, of the array **comm** must be at least $\max(1, \mathbf{lcomm})$ (see nag_complex_sparse_eigensystem_init (f12anc)).

On initial entry: must remain unchanged following a call to the setup function nag_complex_sparse_eigensystem_init (f12anc).

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_complex_sparse_eigensystem_init (f12anc)).

On initial entry: must remain unchanged following a call to the setup function nag_complex_sparse_eigensystem_init (f12anc).

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_INITIALIZATION

Either the initialization function has not been called prior to the first call of this function or a communication array has become corrupted.

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 = ⟨*value*⟩. The number of converged eigenvalues = ⟨*value*⟩. The post-processing function nag_complex_sparse_eigensystem_sol (f12aqc) 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_complex_sparse_eigensystem_option (f12arc) 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_complex_sparse_eigensystem_init (f12anc)).

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_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_complex_sparse_eigensystem_iter (f12apc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_complex_sparse_eigensystem_iter (f12apc) 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 solves $Ax = \lambda x$ in shift-invert mode, where A is obtained from the standard central difference discretization of the convection-diffusion operator $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \rho\frac{\partial u}{\partial x}$ on the unit square, with zero Dirichlet boundary conditions. The shift used is a complex number.

10.1 Program Text

```
/* nag_complex_sparse_eigensystem_iter (f12apc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 8, 2005.
 */

#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nag_string.h>
#include <stdio.h>
#include <naga02.h>
#include <nagf12.h>
#include <nagf16.h>

static void my_zgttrf(Integer, Complex *, Complex *, Complex *,
                      Complex *, Integer *, Integer *);
static void my_zgttrs(Integer, Complex *, Complex *, Complex *,
                      Complex *, Integer *, Complex *, Complex *);

int main(void)
{
  /* Constants */
  Integer  licomm = 140, imon = 0;

  /* Scalars */
  Complex  rho, s1, s2, s3, sigma;
  double   estnrm, hr1, hr2, sr;
  Integer  exit_status, info, irevcm, j, lcomm, n, nconv;
  Integer  ncv, nev, niter, nshift, nx;
  /* Nag types */
  NagError fail;
  /* Arrays */
  Complex  *comm = 0, *eigv = 0, *eigest = 0, *dd = 0, *dl = 0, *du = 0;
  Complex  *du2 = 0, *resid = 0, *v = 0;
  Integer  *icomm = 0, *ipiv = 0;
  /* Ponters */
  Complex  *mx = 0, *x = 0, *y = 0;

  exit_status = 0;
  INIT_FAIL(fail);

  printf("nag_complex_sparse_eigensystem_iter (f12apc) 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"%"NAG_IFMT"%*[^\n] ", &nx, &nev, &ncv);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &nx, &nev, &ncv);
#endif
  n = nx * nx;
  lcomm = 3*n + 3*ncv*ncv + 5*ncv + 60;
  /* Allocate memory */
```

```
  if (!(comm = NAG_ALLOC(lcomm, Complex)) ||
      !(eigv = NAG_ALLOC(ncv, Complex)) ||
      !(eigest = NAG_ALLOC(ncv, Complex)) ||
      !(dd = NAG_ALLOC(n, Complex)) ||
      !(dl = NAG_ALLOC(n, Complex)) ||
      !(du = NAG_ALLOC(n, Complex)) ||
      !(du2 = NAG_ALLOC(n, Complex)) ||
      !(resid = NAG_ALLOC(n, Complex)) ||
      !(v = NAG_ALLOC(n * ncv, Complex)) ||
      !(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_complex_sparse_eigensystem_init (f12anc). */
  nag_complex_sparse_eigensystem_init(n, nev, ncv, icomm, licomm,
                                      comm, lcomm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf(
              "Error from nag_complex_sparse_eigensystem_init (f12anc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }
  /* Select the required mode using
     nag_complex_sparse_eigensystem_option (f12arc). */
  nag_complex_sparse_eigensystem_option("SHIFTED INVERSE", icomm,
                                        comm, &fail);
  /* Set values for sigma and rho */
  /* Assign to Complex type using nag_complex (a02bac) */
  sigma = nag_complex(0.0, 0.0);
  rho = nag_complex(10.0, 0.0);
  hr1 = (double)(n+1);                    /* 1/h */
  hr2 = hr1*hr1;                          /* 1/(h*h) */
  sr = 0.5*hr1*rho.re;                    /* s/h */
  /* Assign to Complex type using nag_complex (a02bac) */
  s1 = nag_complex(-hr2-sr, 0.0);         /* -1/(h*h) - s/h */
  s3 = nag_complex(-hr2+sr, 0.0);         /* -1/(h*h) + s/h */
  s2 = nag_complex(2.0*hr2, 0.0);
  /* Compute Complex subtraction using nag_complex_subtract
     (a02cbc). */
  s2 = nag_complex_subtract(s2, sigma);   /* two/(h*h) - sigma */

  for (j = 0; j <= n - 2; ++j)
    {
      dl[j] = s1;
      dd[j] = s2;
      du[j] = s3;
    }
  dd[n - 1] = s2;

  my_zgttrf(n, dl, dd, du, du2, ipiv, &info);
  irevcm = 0;
 REVCOMLOOP:
  /* repeated calls to reverse communication routine
     nag_complex_sparse_eigensystem_iter (f12apc). */
  nag_complex_sparse_eigensystem_iter(&irevcm, resid, v, &x, &y, &mx,
                                      &nshift, comm, icomm, &fail);
  if (irevcm != 5)
    {
      if (irevcm == -1 || irevcm == 1)
        {
          /* Perform  x <--- OP*x = inv[A-sigma*I]*x */
          my_zgttrs(n, dl, dd, du, du2, ipiv, x, y);
        }
      else if (irevcm == 4 && imon == 1)
        {
```

```
          /* If imon=1, get monitoring information using
             nag_complex_sparse_eigensystem_monit (f12asc). */
          nag_complex_sparse_eigensystem_monit(&niter, &nconv, eigv,
                                               eigest, icomm, comm);
          /* Compute 2-norm of Ritz estimates using
             nag_zge_norm (f16uac). */
          nag_zge_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\n", estnrm);
        }
      goto REVCOMLOOP;
    }
  if (fail.code == NE_NOERROR)
    {
      /* Post-Process using nag_complex_sparse_eigensystem_sol
         (f12aqc) to compute eigenvalues/vectors. */
      nag_complex_sparse_eigensystem_sol(&nconv, eigv, v, sigma,
                                         resid, v, comm, icomm,
                                         &fail);
      printf("\n");
      printf(" The %4"NAG_IFMT" Ritz values of smallest magnitude are:\n\n",
              nconv);
      for (j = 0; j <= nconv-1; ++j)
        {
          printf("%8"NAG_IFMT"%5s( %12.4f , %12.4f )\n", j+1, "",
                  eigv[j].re, eigv[j].im);
        }
    }
  else
    {
      printf(" Error from nag_complex_sparse_eigensystem_iter (f12apc)."
        "\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
 END:
  NAG_FREE(comm);
  NAG_FREE(eigv);
  NAG_FREE(eigest);
  NAG_FREE(dd);
  NAG_FREE(dl);
  NAG_FREE(du);
  NAG_FREE(du2);
  NAG_FREE(resid);
  NAG_FREE(v);
  NAG_FREE(icomm);
  NAG_FREE(ipiv);

  return exit_status;
}

static void my_zgttrf(Integer n, Complex dl[], Complex d[],
                      Complex du[], Complex du2[], Integer ipiv[],
                      Integer *info)
{
  /* A simple C version of the Lapack routine zgttrf with argument
     checking removed */
  /* Scalars */
  Complex temp, fact, z1;
  Integer i;
  /* Function Body */
  *info = 0;
  for (i = 0; i < n; ++i)
    {
      ipiv[i] = i;
    }
  for (i = 0; i < n - 2; ++i)
    {
      du2[i] = nag_complex(0.0, 0.0);
```

```
    }
  for (i = 0; i < n - 2; ++i)
    {
      if (fabs(d[i].re)+fabs(d[i].im) >= fabs(dl[i].re)+fabs(dl[i].im))
        {
          /* No row interchange required, eliminate dl[i]. */
          if (fabs(d[i].re)+fabs(d[i].im) != 0.0)
            {
              /* Compute Complex division using nag_complex_divide
                 (a02cdc). */
              fact = nag_complex_divide(dl[i], d[i]);
              dl[i] = fact;
              /* Compute Complex multiply using nag_complex_multiply
                 (a02ccc). */
              fact = nag_complex_multiply(fact, du[i]);
              /* Compute Complex subtraction using
                 nag_complex_subtract (a02cbc). */
              d[i+1] = nag_complex_subtract(d[i+1], fact);
            }
        }
      else
        {
          /* Interchange rows I and I+1, eliminate dl[I] */
          /* Compute Complex division using nag_complex_divide
             (a02cdc). */
          fact = nag_complex_divide(d[i], dl[i]);
          d[i] = dl[i];
          dl[i] = fact;
          temp = du[i];
          du[i] = d[i+1];
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          z1 = nag_complex_multiply(fact, d[i+1]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          d[i+1] = nag_complex_subtract(temp, z1);
          du2[i] = du[i+1];
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          du[i+1] = nag_complex_multiply(fact, du[i+1]);
          /* Perform Complex negation using nag_complex_negate
             (a02cec). */
          du[i+1] = nag_complex_negate(du[i+1]);
          ipiv[i] = i + 1;
        }
    }
  if (n > 1)
    {
      i = n - 2;
      if (fabs(d[i].re)+fabs(d[i].im) >= fabs(dl[i].re)+fabs(dl[i].im))
        {
          if (fabs(d[i].re)+fabs(d[i].im) != 0.0)
            {
              /* Compute Complex division using nag_complex_divide
                 (a02cdc). */
              fact = nag_complex_divide(dl[i], d[i]);
              dl[i] = fact;
              /* Compute Complex multiply using nag_complex_multiply
                 (a02ccc). */
              fact = nag_complex_multiply(fact, du[i]);
              /* Compute Complex subtraction using
                 nag_complex_subtract (a02cbc). */
              d[i+1] = nag_complex_subtract(d[i+1], fact);
            }
        }
      else
        {
          /* Compute Complex division using nag_complex_divide
             (a02cdc). */
          fact = nag_complex_divide(d[i], dl[i]);
          d[i] = dl[i];
```

```
          dl[i] = fact;
          temp = du[i];
          du[i] = d[i+1];
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          z1 = nag_complex_multiply(fact, d[i+1]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          d[i+1] = nag_complex_subtract(temp, z1);
          ipiv[i] = i + 1;
        }
    }
  /* Check for a zero on the diagonal of U. */
  for (i = 0; i < n; ++i)
    {
      if (fabs(d[i].re)+fabs(d[i].im) == 0.0)
        {
          *info = i;
          goto END;
        }
    }
 END:
  return;
}

static void my_zgttrs(Integer n, Complex dl[], Complex d[],
                      Complex du[], Complex du2[], Integer ipiv[],
                      Complex b[], Complex y[])
{
  /* A simple C version of the Lapack routine zgttrs with argument
     checking removed, the number of right-hand-sides=1, Trans='N' */
  /* Scalars */
  Complex temp, z1;
  Integer i;
  /* Solve L*x = b. */
  for (i = 0; i < n; ++i)
    {
      y[i] = b[i];
    }
  for (i = 0; i < n - 1; ++i)
    {
      if (ipiv[i] == i)
        {
          /* y[i+1] = y[i+1] - dl[i]*y[i] */
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          temp = nag_complex_multiply(dl[i], y[i]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          y[i+1] = nag_complex_subtract(y[i+1], temp);
        }
      else
        {
          temp = y[i];
          y[i] = y[i+1];
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          z1 = nag_complex_multiply(dl[i], y[i]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          y[i+1] = nag_complex_subtract(temp, z1);
        }
    }
  /* Solve U*x = b. */
  /* Compute Complex division using nag_complex_divide (a02cdc). */
  y[n-1] = nag_complex_divide(y[n-1], d[n-1]);
  if (n > 1)
    {
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      temp = nag_complex_multiply(du[n-2], y[n-1]);
```

```
      /* Compute Complex subtraction using nag_complex_subtract
         (a02cbc). */
      z1 = nag_complex_subtract(y[n-2], temp);
      /* Compute Complex division using nag_complex_divide (a02cdc). */
      y[n-2] = nag_complex_divide(z1, 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]; */
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      temp = nag_complex_multiply(du[i], y[i+1]);
      z1 = nag_complex_multiply(du2[i], y[i+2]);
      /* Compute Complex addition using nag_complex_add
         (a02cac). */
      temp = nag_complex_add(temp, z1);
      /* Compute Complex subtraction using nag_complex_subtract
         (a02cbc). */
      z1 = nag_complex_subtract(y[i], temp);
      /* Compute Complex division using nag_complex_divide
         (a02cdc). */
      y[i] = nag_complex_divide(z1, d[i]);
    }
  return;
}
```

10.2 Program Data

```
nag_complex_sparse_eigensystem_iter (f12apc) Example Program Data
 10  4  20 : Vaues for nx, nev and ncv
```

10.3 Program Results

```
nag_complex_sparse_eigensystem_iter (f12apc) Example Program Results

 The    4 Ritz values of smallest magnitude are:

       1     (      34.8720 ,      -0.0000 )
       2     (      64.4326 ,       0.0000 )
       3     (     113.6685 ,      -0.0000 )
       4     (     182.5320 ,       0.0000 )
```