NAG Library Function Document

nag_complex_sparse_eigensystem_option (f12arc)

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 this function need not be called. If, however, you wish to reset some or all of the settings please refer to Section 11 for a detailed description of the specification of the optional arguments*.

1 Purpose

nag_complex_sparse_eigensystem_option (f12arc) is an option setting function 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), for which it may be used to supply individual optional arguments to nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_sparse_eigensystem_sol (f12aqc). nag_complex_sparse_eigensystem_option (f12arc) is also an option setting function in a suite of functions consisting of nag_complex_sparse_eigensystem_init (f12anc), nag_complex_banded_eigensystem_init (f12atc) and nag_complex_banded_eigensystem_solve (f12auc) for which it may be used to supply individual optional arguments to nag_complex_banded_eigensystem_solve (f12auc).

The initialization function for the appropriate suite, nag_complex_sparse_eigensystem_init (f12anc) or nag_complex_banded_eigensystem_init (f12atc), **must** have been called prior to calling nag_complex_sparse_eigensystem_option (f12arc).

2 Specification

```
#include <nag.h>
#include <nagf12.h>
void nag_complex_sparse_eigensystem_option (const char *str,
     Integer icomm[], Complex comm[], NagError *fail)
```

3 Description

nag_complex_sparse_eigensystem_option (f12arc) may be used to supply values for optional arguments to nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_sparse_eigensystem_sol (f12aqc), or to nag_complex_banded_eigensystem_solve (f12auc). It is only necessary to call nag_complex_sparse_eigensystem_option (f12arc) for those arguments whose values are to be different from their default values. One call to nag_complex_sparse_eigensystem_option (f12arc) sets one argument value.

Each optional argument is defined by a single character string consisting of one or more items. The items associated with a given option must be separated by spaces, or equals signs [=]. Alphabetic characters may be upper or lower case. The string

```
'Iteration Limit = 500'
```

is an example of a string used to set an optional argument. For each option the string contains one or more of the following items:

– a mandatory keyword;

– a phrase that qualifies the keyword;

– a number that specifies an Integer or double value. Such numbers may be up to 16 contiguous characters in C's `d` or `g` format.

nag_complex_sparse_eigensystem_option (f12arc) does not have an equivalent function from the ARPACK package which passes options by directly setting values to scalar arguments or to specific

elements of array arguments. nag_complex_sparse_eigensystem_option (f12arc) is intended to make the passing of options more transparent and follows the same principle as the single option setting functions in Chapter e04 (see nag_opt_sparse_convex_qp_option_set_string (e04nsc) for an example).

The setup function nag_complex_sparse_eigensystem_init (f12anc) must be called prior to the first call to nag_complex_sparse_eigensystem_option (f12arc) or nag_complex_banded_eigensystem_init (f12atc), and all calls to nag_complex_sparse_eigensystem_option (f12arc) must precede the first call to nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_solve (f12auc).

A complete list of optional arguments, their abbreviations, synonyms and default values is given in Section 11.

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

1: **str** – const char * *Input*

On entry: a single valid option string (as described in Section 3 and Section 11).

2: **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 on the current options set.

3: **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 on the current options set.

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

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

6 Error Indicators and Warnings

NE_BAD_PARAM

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

NE_INITIALIZATION

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

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

NE_INVALID_OPTION

Ambiguous keyword: ⟨*value*⟩

Keyword not recognized: ⟨*value*⟩

Second keyword not recognized: ⟨*value*⟩

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

None.

10 Example

This example solves $Ax = \lambda Bx$ in shifted-inverse mode, where A and B are derived from the finite element discretization of the one-dimensional convection-diffusion operator $\frac{d^2u}{dx^2} + \rho\frac{du}{dx}$ on the interval $[0, 1]$, with zero Dirichlet boundary conditions.

10.1 Program Text

```
/* nag_complex_sparse_eigensystem_option (f12arc) Example Program.
 *
 * Copyright 2005 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>

/* Table of constant values */
static Complex four = { 4., 0. };

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

int main(void)
{
```

```
  /* Constants */
  Integer  licomm = 140, imon = 0;

  /* Scalars */
  Complex  rho, s1, s2, s3, sigma;
  double   estnrm, hr, hr1, sr, shs;
  Integer  exit_status, info, irevcm, j, lcomm, n, nconv, ncv;
  Integer  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_option (f12arc) Example "
         "Program Results\n");
  /* Skip heading in data file */
  scanf("%*[^\n] ");
  scanf("%ld%ld%ld%*[^\n] ", &nx, &nev, &ncv);
  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);
  /* Select the problem type using
     nag_complex_sparse_eigensystem_option (f12arc). */
  nag_complex_sparse_eigensystem_option("GENERALIZED", icomm,
                                        comm, &fail);
  /* Set values for sigma and rho */
  /* Assign to Complex type using nag_complex (a02bac) */
  sigma = nag_complex(500.0, 0.0);
  rho = nag_complex(10.0, 0.0);
  hr1 = (double)(n+1);                    /* one/h */
  hr = 1.0/hr1;                           /* h */
  sr = 0.5*rho.re;                        /* s */
```

```
  shs = sigma.re*hr/6.0;                  /* sigma*h/6 */
  /* Assign to Complex type using nag_complex (a02bac) */
  s1 = nag_complex(-hr1-sr-shs, 0.0);     /* -one/h - s -sigma*h/six */
  s3 = nag_complex(-hr1+sr-shs, 0.0);     /* -one/h + s -sigma*h/six */
  s2 = nag_complex(2.0*hr1-4.0*shs, 0.0); /* two/h - four*sigma*h/six */

  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)
        {
          /* Perform  x <--- OP*x = inv[A-SIGMA*M]*M*x */
          mv(nx, x, y);
          my_zgttrs(n, dl, dd, du, du2, ipiv, y);
        }
      else if (irevcm == 1)
        {
          /* Perform  x <--- OP*x = inv[A-SIGMA*M]*M*x, */
          /* MX stored in mx */
          for (j = 0; j < n; ++j)
            {
              y[j] = mx[j];
            }
          my_zgttrs(n, dl, dd, du, du2, ipiv, y);
        }
      else if (irevcm == 2)
        {
          /* Perform  y <--- M*x */
          mv(nx, 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 %3ld, ", niter);
          printf(" No. converged = %3ld,", nconv);
          printf(" norm of estimates = %17.8e\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 The %4ld generalized Ritz values closest to "
             "( %7.3f , %7.3f ) are:\n\n", nconv, sigma.re, sigma.im);
      for (j = 0; j <= nconv-1; ++j)
        {
          printf("%8ld%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 mv(Integer nx, Complex *v, Complex *y)
{
  /* Compute the out-of--place matrix vector multiplication Y<---M*X, */
  /* where M is mass matrix formed by using piecewise linear elements */
  /* on [0,1]. */

  /* Scalars */
  Complex hsix, z1;
  Integer j, n;
  /* Function Body */
  n = nx * nx;
  /* Assign to Complex type using nag_complex (a02bac) */
  hsix = nag_complex(1.0/(6.0*(double)(n+1)), 0.0);
  /* y[0] = (four*v[0]+v[1])*(h/six) */
  /* Compute Complex multiply using nag_complex_multiply
     (a02ccc). */
  z1 = nag_complex_multiply(four, v[0]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  z1 = nag_complex_add(z1, v[1]);
  y[0] = nag_complex_multiply(z1, hsix);
  for (j = 1; j <= n - 2; ++j)
    {
      /* y[j] = (v[j-1] + four*v[j] + V[j+1])*(h/six) */
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      z1 = nag_complex_multiply(four, v[j]);
      /* Compute Complex addition using nag_complex_add (a02cac). */
      z1 = nag_complex_add(v[j-1], z1);
      z1 = nag_complex_add(z1, v[j+1]);
      y[j] = nag_complex_multiply(z1, hsix);
    }
  /* y[n-1] = (v[n-2] + four*v[n-1])*(h/six) */
  /* Compute Complex multiply using nag_complex_multiply (a02ccc). */
  z1 = nag_complex_multiply(four, v[n-1]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  z1 = nag_complex_add(v[n-2], z1);
  y[n-1] = nag_complex_multiply(z1, hsix);
  return;
} /* mv */

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[])
{
  /* 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 - 1; ++i)
    {
      if (ipiv[i] == i)
        {
          /* b[i+1] = b[i+1] - dl[i]*b[i] */
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          temp = nag_complex_multiply(dl[i], b[i]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          b[i+1] = nag_complex_subtract(b[i+1], temp);
        }
      else
        {
          temp = b[i];
          b[i] = b[i+1];
          /* Compute Complex multiply using nag_complex_multiply
             (a02ccc). */
          z1 = nag_complex_multiply(dl[i], b[i]);
          /* Compute Complex subtraction using nag_complex_subtract
             (a02cbc). */
          b[i+1] = nag_complex_subtract(temp, z1);
        }
```

```
    }
  /* Solve U*x = b. */
  /* Compute Complex division using nag_complex_divide (a02cdc). */
  b[n-1] = nag_complex_divide(b[n-1], d[n-1]);
  if (n > 1)
    {
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      temp = nag_complex_multiply(du[n-2], b[n-1]);
      /* Compute Complex subtraction using nag_complex_subtract
         (a02cbc). */
      z1 = nag_complex_subtract(b[n-2], temp);
      /* Compute Complex division using nag_complex_divide (a02cdc). */
      b[n-2] = nag_complex_divide(z1, d[n-2]);
    }
  for (i = n - 3; i >= 0; --i)
    {
      /* b[i] = (b[i]-du[i]*b[i+1]-du2[i]*b[i+2])/d[i]; */
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      temp = nag_complex_multiply(du[i], b[i+1]);
      z1 = nag_complex_multiply(du2[i], b[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(b[i], temp);
      /* Compute Complex division using nag_complex_divide
         (a02cdc). */
      b[i] = nag_complex_divide(z1, d[i]);
    }
  return;
}
```

10.2 Program Data

```
nag_complex_sparse_eigensystem_option (f12arc) Example Program Data
 10  4  20 : Vaues for nx, nev and ncv
```

10.3 Program Results

```
nag_complex_sparse_eigensystem_option (f12arc) Example Program Results

 The    4 generalized Ritz values closest to ( 500.000 ,   0.000 ) are:

       1     (     509.9390 ,       0.0000 )
       2     (     380.9092 ,       0.0000 )
       3     (     659.1558 ,      -0.0000 )
       4     (     271.9412 ,      -0.0000 )
```

11 Optional Arguments

Several optional arguments for the computational suite functions nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_sparse_eigensystem_sol (f12aqc), and for the banded driver nag_complex_banded_eigensystem_solve (f12auc), define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_complex_sparse_eigensystem_iter (f12apc), nag_complex_sparse_eigensystem_sol (f12aqc) and nag_complex_banded_eigensystem_solve (f12auc) these optional arguments have associated *default values* that are appropriate for most problems. Therefore, you need only specify those optional arguments whose values are to be different from 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
Exact Shifts
Generalized
Initial Residual
Iteration Limit
Largest Imaginary
Largest Magnitude
Largest Real
List
Monitoring
Nolist
Print Level
Random Residual
Regular
Regular Inverse
Shifted Inverse
Smallest Imaginary
Smallest Magnitude
Smallest Real
Standard
Supplied Shifts
Tolerance
Vectors

Optional parameters may be specified by calling nag_complex_sparse_eigensystem_option (f12arc) before a call to nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_init (f12atc), but after a corresponding call to nag_complex_sparse_eigensystem_init (f12anc) or nag_complex_banded_eigensystem_solve (f12auc). One call is necessary for each optional argument. Any optional arguments you do not specify are set to their default values. Optional arguments you do specify are unaltered by nag_complex_sparse_eigensystem_iter (f12apc), nag_complex_sparse_eigensystem_sol (f12aqc) and nag_complex_banded_eigensystem_solve (f12auc) (unless they define invalid values) and so remain in effect for subsequent calls unless you alter them.

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 Nag_FileID. This ID value must either be set to 0 (the default value) in which case there will be no output, or will be as returned by a call of nag_open_file (x04acc).

Advisory Default $= 0$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

Advisory messages are output to Nag_FileID **Advisory** during the solution of the problem.

Defaults

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

Exact Shifts Default
Supplied Shifts

During the Arnoldi iterative process, shifts are applied as part of the implicit restarting scheme. The shift strategy used by default and selected by the optional argument **Exact Shifts** is strongly recommended over the alternative **Supplied Shifts** and will always be used by nag_complex_banded_eigensystem_solve (f12auc).

If **Exact Shifts** are used then these are computed internally by the algorithm in the implicit restarting scheme. This strategy is generally effective and cheaper to apply in terms of number of operations than using explicit shifts.

If **Supplied Shifts** are used then, during the Arnoldi iterative process, you must supply shifts through array arguments of nag_complex_sparse_eigensystem_iter (f12apc) when nag_complex_sparse_eigensystem_iter (f12apc) returns with **irevcm** $= 3$; the complex shifts are supplied in **y**. This option should only be used if you are an experienced user since this requires some algorithmic knowledge and because more operations are usually required than for the implicit shift scheme. Details on the use of explicit shifts and further references on shift strategies are available in Lehoucq *et al.* (1998).

Iteration Limit i Default $= 300$

The limit on the number of Arnoldi iterations that can be performed before nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_solve (f12auc) exits. If not all requested eigenvalues have converged to within **Tolerance** and the number of Arnoldi iterations has reached this limit then nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_solve (f12auc) exits with an error; nag_complex_banded_eigensystem_solve (f12auc) returns the number of converged eigenvalues, the converged eigenvalues and, if requested, the corresponding eigenvectors, while nag_complex_sparse_eigensystem_sol (f12aqc) can be called subsequent to nag_complex_sparse_eigensystem_iter (f12apc) to do the same.

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 for nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_solve (f12auc) 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 (and B for **Generalized** problems), 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 Default $= -1$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

Unless **Monitoring** is set to -1 (the default), monitoring information is output to Nag_FileID **Monitoring** 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.

Print Level i Default $= 0$

This controls the amount of printing produced by nag_complex_sparse_eigensystem_option (f12arc) as follows.

$= 0$ No output except error messages.

> 0 The set of selected options.

$= 2$ Problem and timing statistics on final exit from nag_complex_sparse_eigensystem_iter (f12apc) or nag_complex_banded_eigensystem_solve (f12auc).

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

≥ 10 If **Monitoring** is set, 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_complex_sparse_eigensystem_iter (f12apc). If **Monitoring** is set, 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, 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, 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, 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.

Random Residual Default
Initial Residual

To begin the Arnoldi iterative process, nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_banded_eigensystem_solve (f12auc) requires an initial residual vector. By default nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_banded_eigensystem_solve (f12auc)

provides its own random initial residual vector; this option can also be set using optional argument **Random Residual**. Alternatively, you can supply an initial residual vector (perhaps from a previous computation) to nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_banded_eigensystem_solve (f12auc) through the array argument **resid**; this option can be set using optional argument **Initial Residual**.

Regular Default
Regular Inverse
Shifted Inverse

These options define the computational mode which in turn defines the form of operation OP(x) to be performed by nag_complex_banded_eigensystem_solve (f12auc) or when nag_complex_sparse_eigensystem_iter (f12apc) returns with **irevcm** $= -1$ or 1 and the matrix-vector product Bx when nag_complex_sparse_eigensystem_iter (f12apc) returns with **irevcm** $= -2$.

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	OP $= (A - \sigma I)^{-1}$

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

Regular Inverse	OP $= B^{-1}A$
Shifted Inverse	OP $= (A - \sigma B)^{-1}B$

Standard Default
Generalized

The problem to be solved is either a standard eigenvalue problem, $Ax = \lambda x$, or a generalized eigenvalue problem, $Ax = \lambda Bx$. The optional argument **Standard** should be used when a standard eigenvalue problem is being solved and the optional argument **Generalized** should be used when a generalized eigenvalue problem is being solved.

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.

Vectors Default = 'RITZ'

The function nag_complex_sparse_eigensystem_sol (f12aqc) or nag_complex_banded_eigensystem_solve (f12auc) can optionally compute the Schur vectors and/or the eigenvectors corresponding to the converged eigenvalues. To turn off computation of any vectors the option **Vectors** = 'NONE' should be set. To compute only the Schur vectors (at very little extra cost), the option **Vectors** = 'SCHUR' should be set and these will be returned in the array argument **v** of nag_complex_sparse_eigensystem_sol (f12aqc) or nag_complex_banded_eigensystem_solve (f12auc). To compute the eigenvectors (Ritz vectors) corresponding to the eigenvalue estimates, the option **Vectors** = 'RITZ' should be set and these will be returned in the array argument **z** of nag_complex_sparse_eigensystem_sol (f12aqc) or nag_complex_banded_eigensystem_solve (f12auc), if **z** is set equal to **v** (as in Section 10) then the Schur vectors in **v** are overwritten by the eigenvectors computed by nag_complex_sparse_eigensystem_sol (f12aqc) or nag_complex_banded_eigensystem_solve (f12auc).