

NAG Library Function Document

nag_complex_sparse_eigensystem_monit (f12asc)

Note: this function uses **optional parameters** to define choices in the problem specification. If you wish to use default settings for all of the optional parameters, 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 parameters.

1 Purpose

`nag_complex_sparse_eigensystem_monit (f12asc)` can be used to return additional monitoring information during computation. It is 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)`.

2 Specification

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

void nag_complex_sparse_eigensystem_monit (Integer *niter, Integer *nconv,
      Complex ritz[], Complex rzest[], const Integer icomm[],
      const Complex comm[])
```

3 Description

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

On an intermediate exit from `nag_complex_sparse_eigensystem_iter (f12apc)` with `irevcm = 4`, `nag_complex_sparse_eigensystem_monit (f12asc)` may be called to return monitoring information on the progress of the Arnoldi iterative process. The information returned by `nag_complex_sparse_eigensystem_monit (f12asc)` is:

- the number of the current Arnoldi iteration;
- the number of converged eigenvalues at this point;
- the converged eigenvalues;
- the error bounds on the converged eigenvalues.

`nag_complex_sparse_eigensystem_monit (f12asc)` does not have an equivalent function from the ARPACK package which prints various levels of detail of monitoring information through an output channel controlled via a argument value (see Lehoucq *et al.* (1998) for details of ARPACK routines). `nag_complex_sparse_eigensystem_monit (f12asc)` should not be called at any time other than immediately following an `irevcm = 4` return from `nag_complex_sparse_eigensystem_iter (f12apc)`.

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, Philadelphia

5 Arguments

- 1: **niter** – Integer * *Output*
On exit: the number of the current Arnoldi iteration.
- 2: **nconv** – Integer * *Output*
On exit: the number of converged eigenvalues so far.
- 3: **ritz**[*dim*] – Complex *Output*
Note: the dimension, *dim*, of the array **ritz** must be at least **ncv** (see nag_complex_sparse_eigensystem_init (f12anc)).
On exit: the first **nconv** locations of the array **ritz** contain the converged approximate eigenvalues.
- 4: **rzest**[*dim*] – Complex *Output*
Note: the dimension, *dim*, of the array **rzest** must be at least **ncv** (see nag_complex_sparse_eigensystem_init (f12anc)).
On exit: the first **nconv** locations of the array **rzest** contain the complex Ritz estimates on the converged approximate eigenvalues.
- 5: **icomm**[*dim*] – const Integer *Communication Array*
Note: the dimension, *dim*, of the array **icomm** must be at least $\max(1, \mathbf{licomm})$, where **licomm** is passed to the setup function (see nag_complex_sparse_eigensystem_init (f12anc)).
On entry: the array **icomm** output by the preceding call to nag_complex_sparse_eigensystem_iter (f12apc).
- 6: **comm**[*dim*] – const Complex *Communication Array*
Note: the dimension, *dim*, of the array **comm** must be at least $\max(1, \mathbf{licomm})$, where **licomm** is passed to the setup function (see nag_complex_sparse_eigensystem_init (f12anc)).
On entry: the array **comm** output by the preceding call to nag_complex_sparse_eigensystem_iter (f12apc).

6 Error Indicators and Warnings

None.

7 Accuracy

A Ritz value, λ , is deemed to have converged if the magnitude of its Ritz estimate $\leq \mathbf{Tolerance} \times |\lambda|$. The default **Tolerance** used is the *machine precision* given by nag_machine_precision (X02AJC).

8 Parallelism and Performance

nag_complex_sparse_eigensystem_monit (f12asc) is not threaded in any implementation.

9 Further Comments

None.

10 Example

This example solves $Ax = \lambda Bx$ in shifted-inverse mode, where A and B are obtained from the standard central difference discretization of the one-dimensional convection-diffusion operator $\frac{d^2u}{dx^2} + \rho \frac{du}{dx}$ on $[0, 1]$, with zero Dirichlet boundary conditions. The shift, σ , is a complex number, and the operator used in the shifted-inverse iterative process is $OP = \text{inv}(A - \sigma B) \times B$.

10.1 Program Text

```

/* nag_complex_sparse_eigensystem_monit (f12asc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#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_zgtrrf(Integer, Complex *, Complex *, Complex *,
                    Complex *, Integer *, Integer *);
static void my_zgtrrs(Integer, Complex *, Complex *, Complex *,
                    Complex *, Integer *, Complex *);

int main(void)
{
    /* Constants */
    Integer licomm = 140, imon = 1;

    /* Scalars */
    Complex rho, s1, s2, s3, sigma;
    double estnrm, hr, hrl, 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_monit (f12asc) 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;
    }
/* Initialize 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 (f12adc). */
    nag_complex_sparse_eigensystem_option("SHIFTED INVERSE", icomm, comm,
        &fail);
/* Select the problem type using
    nag_complex_sparse_eigensystem_option (f12adc). */
    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(5000.0, 0.0);
    rho = nag_complex(10.0, 0.0);
    hr1 = (double) (n + 1);
    hr = 1.0 / hr1;
    sr = 0.5 * rho.re;
    shs = sigma.re * hr / 6.0;
/* Assign to Complex type using nag_complex (a02bac) */
    s1 = nag_complex(-hr1 - sr - shs, 0.0);
    s3 = nag_complex(-hr1 + sr - shs, 0.0);
    s2 = nag_complex(2.0 * hr1 - 4.0 * shs, 0.0);

    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  $x \leftarrow OP*x = inv[A-SIGMA*M]*M*x$  */
    mv(nx, x, y);
    my_zgttrs(n, dl, dd, du, du2, ipiv, y);
  }
  else if (irevcm == 1) {
    /* Perform  $x \leftarrow OP*x = inv[A-SIGMA*M]*M*x$ , */
    /* 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  $y \leftarrow M*x$  */
    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 %3" NAG_IFMT " ", niter);
    printf(" No. converged = %3" NAG_IFMT " ", nconv);
    printf(" norm of estimates = %17.8e\n", estnrm);
  }
  goto REVCOMLOOP;
}
if (fail.code == NE_NOERROR) {
  /* Post-Process using nag_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 " generalized Ritz values closest", nconv);
  printf(" to ( %7.3f , %7.3f ) are:\n\n", sigma.re, sigma.im);
  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 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_zgtrrf(Integer n, Complex dl[], Complex d[],
                    Complex du[], Complex du2[], Integer ipiv[],
                    Integer *info)
{
    /* A simple C version of the Lapack routine zgtrrf 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_monit (f12asc) Example Program Data
 16 4 10 : Vaues for nx, nev and ncv

10.3 Program Results

```
nag_complex_sparse_eigensystem_monit (f12asc) Example Program Results
Iteration 1, No. converged = 0, norm of estimates = 5.94672394e-07
Iteration 2, No. converged = 1, norm of estimates = 1.47822750e-09
Iteration 3, No. converged = 2, norm of estimates = 3.29270191e-11
Iteration 4, No. converged = 2, norm of estimates = 5.94072850e-14
Iteration 5, No. converged = 2, norm of estimates = 8.40812032e-16
Iteration 6, No. converged = 3, norm of estimates = 8.13435708e-18
```

The 4 generalized Ritz values closest to (5000.000 , 0.000) are:

```
1 ( 4829.8497 , -0.0000 )
2 ( 5279.5223 , -0.0000 )
3 ( 4400.6310 , 0.0000 )
4 ( 5749.7160 , -0.0000 )
```
