

NAG C Library Function Document

nag_real_sparse_eigensystem_monit (f12aec)

1 Purpose

nag_real_sparse_eigensystem_monit (f12aec) can be used to return additional monitoring information during computation. It is in a suite of functions consisting of nag_real_sparse_eigensystem_monit (f12aec), nag_real_sparse_eigensystem_init (f12aac), nag_real_sparse_eigensystem_iter (f12abc), nag_real_sparse_eigensystem_sol (f12acc) and nag_real_sparse_eigensystem_option (f12adc).

2 Specification

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

void nag_real_sparse_eigensystem_monit (Integer *niter, Integer *nconv,
    double ritzr[], double ritzi[], double rzest[], Integer icomm[], double 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 eigenvalue problem $Ax = \lambda x$, or of a generalized eigenvalue problem $Ax = \lambda Bx$ of order n , where n is large and the coefficient matrices A and B are sparse, real and non-symmetric. The suite can also be used to find selected eigenvalues/eigenvectors of smaller scale dense, real and non-symmetric problems.

On an intermediate exit from nag_real_sparse_eigensystem_iter (f12abc) with **irevcn** = 4, nag_real_sparse_eigensystem_monit (f12aec) may be called to return monitoring information on the progress of the Arnoldi iterative process. The information returned by nag_real_sparse_eigensystem_monit (f12aec) is:

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

nag_real_sparse_eigensystem_monit (f12aec) 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_real_sparse_eigensystem_monit (f12aec) should not be called at any time other than immediately following an **irevcn** = 4 return from nag_real_sparse_eigensystem_iter (f12abc).

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: **ritzr**[*dim*] – double *Output*
Note: the dimension, *dim*, of the array **ritzr** must be at least **nev** (see nag_real_sparse_eigensystem_init (f12aac)).
On exit: the first **nconv** locations of the array **ritzr** contain the real parts of the converged approximate eigenvalues.
- 4: **ritzi**[*dim*] – double *Output*
Note: the dimension, *dim*, of the array **ritzi** must be at least **nev** (see nag_real_sparse_eigensystem_init (f12aac)).
On exit: the first **nconv** locations of the array **ritzi** contain the imaginary parts of the converged approximate eigenvalues.
- 5: **rzest**[*dim*] – double *Output*
Note: the dimension, *dim*, of the array **rzest** must be at least **nev** (see nag_real_sparse_eigensystem_init (f12aac)).
On exit: the first **nconv** locations of the array **rzest** contain the Ritz estimates (error bounds) on the converged approximate eigenvalues.
- 6: **icomm**[*dim*] – Integer *Communication Array*
Note: the dimension, *dim*, of the array **icomm** must be at least $\max(1, \mathbf{lcomm})$, where **lcomm** is passed to the setup function nag_real_sparse_eigensystem_init (f12aac) (see nag_real_sparse_eigensystem_init (f12aac)).
icomm must remain unchanged.
- 7: **comm**[*dim*] – double *Communication Array*
Note: the dimension, *dim*, of the array **comm** must be at least $\max(1, \mathbf{lcomm})$, where **lcomm** is passed to the setup function nag_real_sparse_eigensystem_init (f12aac) (see nag_real_sparse_eigensystem_init (f12aac)).
comm must remain unchanged.

6 Error Indicators and Warnings

None.

7 Accuracy

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

8 Further Comments

None.

9 Example

This example solves $Ax = \lambda Bx$ in shifted-real mode, where A is the tridiagonal matrix with 2 on the diagonal, -2 on the subdiagonal and 3 on the superdiagonal. The matrix B is the tridiagonal matrix with 4 on the diagonal and 1 on the off-diagonals. The shift sigma, σ , is a complex number, and the operator used in the shifted-real iterative process is $OP = \text{real}((A - \sigma B)_{-1}B)$.

9.1 Program Text

```

/* nag_real_sparse_eigensystem_monit (f12aec) Example Program.
 *
 * Copyright 2005 Numerical Algorithms Group.
 *
 * Mark 8, 2005.
 */

#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <stdio.h>
#include <naga02.h>
#include <nagf12.h>
#include <nagf16.h>
static void mv(Integer, double *, double *);
static void av(Integer, double *, double *);
static int ytax(Integer, double *, double *, double *);
static int ytmx(Integer, double *, double *, double *);
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=1;
    /* Scalars */
    Complex c1, c2, c3, eigv, num, den;
    double estnrm, deni, denr, i2, numi, numr, r2;
    double sigmai, sigmar;
    Integer exit_status, info, irevcm, j, k, lcomm, n;
    Integer nconv, ncv, nev, niter, nshift;
    /* Nag types */
    Nag_Boolean first;
    NagError fail;

    /* Arrays */
    Complex *cdd=0, *cdl=0, *cdu=0, *cdu2=0, *ctemp=0;
    double *comm=0, *eigvr=0, *eigvi=0, *eigest=0;
    double *resid=0, *v=0;
    Integer *icomm=0, *ipiv=0;
    /* Pointers */
    double *mx=0, *x=0, *y=0;

    exit_status = 0;
    INIT_FAIL(fail);

    Vprintf("nag_real_sparse_eigensystem_monit (f12aec) Example Program Results"
           "\n");
    /* Skip heading in data file */
    Vscanf("%*[\n] ");

    /* Read problem parameter values from data file. */
    Vscanf("%ld%ld%ld%lf%lf%*[\n] ", &n, &nev, &ncv,
           &sigmar, &sigmai);
    /* Allocate memory */
    lcomm = 3*n + 3*ncv*ncv + 6*ncv + 60;
    if ( !(cdd = NAG_ALLOC(n, Complex)) ||
        !(cdl = NAG_ALLOC(n, Complex)) ||

```

```

!(cdu = NAG_ALLOC(n, Complex)) ||
!(cdu2 = NAG_ALLOC(n, Complex)) ||
!(ctemp = NAG_ALLOC(n, Complex)) ||
!(comm = NAG_ALLOC(lcomm, double)) ||
!(eigvr = NAG_ALLOC(ncv, double)) ||
!(eigvi = NAG_ALLOC(ncv, double)) ||
!(eigest = NAG_ALLOC(ncv, double)) ||
!(resid = NAG_ALLOC(n, double)) ||
!(v = NAG_ALLOC(n * ncv, double)) ||
!(icomm = NAG_ALLOC(licomm, Integer)) ||
!(ipiv = NAG_ALLOC(n, Integer)) )
{
Vprintf("Allocation failure\n");
exit_status = -1;
goto END;
}

/* Initialise communication arrays for problem using
nag_real_sparse_eigensystem_init (f12aac). */
nag_real_sparse_eigensystem_init(n, nev, ncv, icomm, licomm, comm,
                                lcomm, &fail);
/* Select the required spectrum using
nag_real_sparse_eigensystem_option (f12adc). */
nag_real_sparse_eigensystem_option("SHIFTED REAL", icomm, comm,
                                   &fail);
/* Select the problem type using
nag_real_sparse_eigensystem_option (f12adc). */
nag_real_sparse_eigensystem_option("GENERALIZED", icomm, comm,
                                   &fail);
/* Solve A*x = lambda*B*x in shift-invert mode. */
/* The shift, sigma, is a complex number (sigmar, sigmai). */
/* OP = Real_Part{inv[A-(sigmar,sigmai)*M]*M and B = M. */
c1 = nag_complex(-2. - sigmar, -sigmai);
c2 = nag_complex(2. - sigmar * 4., sigmai * -4.);
c3 = nag_complex(3. - sigmar, -sigmai);

for (j = 0; j <= n - 2; ++j)
{
    cd1[j] = c1;
    cd2[j] = c2;
    cd3[j] = c3;
}
cd3[n-1] = c2;

my_zgtrf(n, cd1, cd2, cd3, ipiv, &info);

irevcn = 0;
REVCNLOOP:
/* repeated calls to reverse communication routine
nag_real_sparse_eigensystem_iter (f12abc). */
nag_real_sparse_eigensystem_iter(&irevcn, resid, v, &x, &y, &mx,
                                &nshift, comm, icomm, &fail);

if (irevcn != 5)
{
    if (irevcn == -1)
    {
        /* Perform x <--- OP*x = inv[A-SIGMA*M]*M*x */
        mv(n, x, y);
        for (j = 0; j <= n-1; ++j)
        {
            ctemp[j].re = y[j], ctemp[j].im = 0.;
        }
        my_zgtrs(n, cd1, cd2, cd3, ipiv, ctemp);
        for (j = 0; j <= n-1; ++j)
        {
            y[j] = ctemp[j].re;
        }
    }
    else if (irevcn == 1)
    {
        /* Perform x <--- OP*x = inv[A-SIGMA*M]*M*x, */

```

```

    /* M*X stored in MX. */
    for (j = 0; j <= n-1; ++j)
    {
        ctemp[j].re = mx[j], ctemp[j].im = 0.;
    }
    my_zgtrrs(n, cdl, cdd, cdu, cdu2, ipiv, ctemp);
    for (j = 0; j <= n-1; ++j)
    {
        y[j] = ctemp[j].re;
    }
}
else if (irevcm == 2)
{
    /* Perform y <--- M*x */
    mv(n, x, y);
}
else if (irevcm == 4 && imon == 1)
{
    /* If imon=1, get monitoring information using
       nag_real_sparse_eigensystem_monit (f12aec). */
    nag_real_sparse_eigensystem_monit(&niter, &nconv, eigvr,
                                       eigvi, eigest, icomm, comm);
    /* Compute 2-norm of Ritz estimates using
       nag_dge_norm (f16rac).*/
    nag_dge_norm(Nag_ColMajor, Nag_FrobeniusNorm, nev, 1, eigest,
                 nev, &estnrm, &fail);
    Vprintf("Iteration %3ld, ", niter);
    Vprintf(" No. converged = %3ld,", nconv);
    Vprintf(" norm of estimates = %16.8e\n", estnrm);
}
goto REVCOMLOOP;
}
if (fail.code == NE_NOERROR)
{
    /* Post-Process using nag_real_sparse_eigensystem_sol
       (f12acc) to compute eigenvalues/vectors. */
    nag_real_sparse_eigensystem_sol(&nconv, eigvr, eigvi, v, sigmar,
                                    sigmai, resid, v, comm, icomm,
                                    &fail);

    first = Nag_TRUE;
    k = 0;
    for (j = 0; j <= nconv-1; ++j)
    {
        /* Use Rayleigh Quotient to recover eigenvalues of the */
        /* original problem. */
        if (eigvi[j] == 0.)
        {
            /* Ritz value is real. */
            /* Numerator = Vj . AVj where Vj is jth Ritz vector */
            if (ytax(n, &v[k], &v[k], &numr) ) { goto END;}
            /* Denominator = Vj . MVj */
            if (ytmx(n, &v[k], &v[k], &denr) ) { goto END;}
            eigvr[j] = numr / denr;
        }
        else if (first)
        {
            /* Ritz value is complex: (x,y). */
            /* Compute x'(Ax) and y'(Ax). */
            if (ytax(n, &v[k], &v[k], &numr) ) { goto END;}
            if (ytax(n, &v[k], &v[k+n], &numi) ) { goto END;}
            /* Compute y'(Ay) and x'(Ay). */
            if (ytax(n, &v[k+n], &v[k+n], &r2) ) { goto END;}
            if (ytax(n, &v[k+n], &v[k], &i2) ) { goto END;}
            numr += r2;
            numi = i2 - numi;
            /* Assign to Complex type using nag_complex (a02bac). */
            num = nag_complex(numr, numi);
            /* Compute x'(Mx) and y'(Mx). */
            if (ytmx(n, &v[k], &v[k], &denr) ) { goto END;}
            if (ytmx(n, &v[k], &v[k+n], &deni) ) { goto END;}
            /* Compute y'(Ay) and x'(Ay). */

```

```

        if (ytmx(n,&v[k+n],&v[k+n],&r2) ) { goto END;}
        if (ytmx(n,&v[k+n],&v[k], &i2) ) { goto END;}
        denr += r2;
        deni = i2 - deni;
        /* Assign to Complex type using nag_complex (a02bac). */
        den = nag_complex(denr,deni);
        /* eigv = x'(Ax)/x'(Mx) */
        /* Compute Complex division using nag_complex_divide
           (a02cdc). */
        eigv = nag_complex_divide(num,den);
        eigvr[j] = eigv.re;
        eigvi[j] = eigv.im;
        first = Nag_FALSE;
    }
    else
    {
        /* Second of complex conjugate pair. */
        eigvr[j] = eigvr[j-1];
        eigvi[j] = -eigvi[j-1];
        first = Nag_TRUE;
    }
    k = k + n;
}
/* Print computed eigenvalues. */
Vprintf("\n The %4ld generalized Ritz values closest", nconv);
Vprintf(" to ( %8.4f , %8.4f ) are:\n\n", sigmar, sigmai);
for (j = 0; j <= nconv-1; ++j)
{
    Vprintf("%8ld%5s( %7.4f, %7.4f )\n", j+1, "",
            eigvr[j],eigvi[j]);
}
}
else
{
    Vprintf(" Error from nag_real_sparse_eigensystem_iter (f12abc).\n%s\n",
            fail.message);
    exit_status = 1;
    goto END;
}
END:
if (cdd) NAG_FREE(cdd);
if (cdl) NAG_FREE(cdl);
if (cdu) NAG_FREE(cdu);
if (cdu2) NAG_FREE(cdu2);
if (ctemp) NAG_FREE(ctemp);
if (comm) NAG_FREE(comm);
if (eigvr) NAG_FREE(eigvr);
if (eigvi) NAG_FREE(eigvi);
if (eigest) NAG_FREE(eigest);
if (resid) NAG_FREE(resid);
if (v) NAG_FREE(v);
if (icomm) NAG_FREE(icomm);
if (ipiv) NAG_FREE(ipiv);

return exit_status;
}

static void mv(Integer n, double *v, double *y)
{
    /* Compute the matrix vector multiplication y<---M*x, */
    /* where M is mass matrix formed by using piecewise linear elements */
    /* on [0,1]. */

    /* Scalars */
    Integer j;

    /* Function Body */
    y[0] = v[0] * 4. + v[1];
    for (j = 1; j <= n - 2; ++j)
    {

```

```

        y[j] = v[j-1] + v[j] * 4. + v[j+1];
    }
    y[n-1] = v[n-2] + v[n-1] * 4.;
    return;
} /* mv */

static void av(Integer n, double *v, double *w)
{
    /* Scalars */
    Integer j;

    /* Function Body */
    w[0] = v[0] * 2. + v[1] * 3.;
    for (j = 1; j <= n - 2; ++j)
    {
        w[j] = v[j-1] * -2. + v[j] * 2. + v[j+1] * 3.;
    }
    w[n-1] = v[n-2] * -2. + v[n-1] * 2.;
    return;
} /* av */

static int ytax(Integer n, double x[], double y[], double *r)
{
    /* Given the vectors x and y, Performs the operation */
    /* y'Ax and returns the scalar value. */

    /* Scalars */
    Integer exit_status, j;
    /* Arrays */
    double *ax=0;

    /* Function Body */
    exit_status = 0;
    /* Allocate memory */
    if ( !(ax = NAG_ALLOC(n, double)) )
    {
        Vprintf("Allocation failure\n");
        exit_status = -1;
        goto YTAXEND;
    }
    av(n, x, ax);
    *r = 0.0;
    for (j = 0; j <= n - 1; ++j)
    {
        *r = *r + y[j] * ax[j];
    }
YTAXEND:
    if (ax) NAG_FREE(ax);
    return exit_status;
} /* ytax */

static int ytmx(Integer n, double x[], double y[], double *r)
{
    /* Given the vectors x and y, Performs the operation */
    /* y'Mx and returns the scalar value. */

    /* Scalars */
    Integer exit_status, j;
    /* Arrays */
    double *mx=0;

    /* Function Body */
    exit_status = 0;
    /* Allocate memory */
    if ( !(mx = NAG_ALLOC(n, double)) )
    {
        Vprintf("Allocation failure\n");
        exit_status = -1;
        goto YTMXEND;
    }

```

```

mv(n, x, mx);
*r = 0.0;
for (j = 0; j <= n - 1; ++j)
    {
        *r = *r + y[j] * mx[j];
    }
YTMXEND:
    if (mx) NAG_FREE(mx);
    return exit_status;
} /* ytmx */

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;
}

```

9.2 Program Data

nag_real_sparse_eigensystem_monit (f12aec) Example Program Data
 100 4 20 4.0e-1 6.0e-1 : Values for nx, nev, ncv, sigmar, sigmai

9.3 Program Results

nag_real_sparse_eigensystem_monit (f12aec) Example Program Results

Iteration	1,	No. converged =	0,	norm of estimates =	1.05198320e-01
Iteration	2,	No. converged =	0,	norm of estimates =	1.18821782e-03
Iteration	3,	No. converged =	0,	norm of estimates =	1.38923424e-06
Iteration	4,	No. converged =	0,	norm of estimates =	3.93878037e-09
Iteration	5,	No. converged =	0,	norm of estimates =	1.15839744e-11
Iteration	6,	No. converged =	0,	norm of estimates =	5.22183096e-14

The 4 generalized Ritz values closest to (0.4000 , 0.6000) are:

1	(0.5000, -0.5958)
2	(0.5000, 0.5958)
3	(0.5000, -0.6331)
4	(0.5000, 0.6331)

