NAG C Library Function Document

nag_complex_sparse_eigensystem_sol (f12aqc)

1 Purpose

nag_complex_sparse_eigensystem_sol (f12aqc) is a post-processing function in a suite of functions consisting of nag_complex_sparse_eigensystem_sol (f12aqc), nag_complex_sparse_eigensystem_init (f12anc), nag_complex_sparse_eigensystem_iter (f12apc), nag_complex_sparse_eigensystem_option (f12arc) and nag_complex_sparse_eigensystem_monit (f12asc), that must be called following a final exit from nag_complex_sparse_eigensystem_sol (f12aqc).

2 Specification

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

void nag_complex_sparse_eigensystem_sol (Integer *nconv, Complex d[],
     Complex z[], Complex sigma, const Complex resid[], Complex v[],
     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 non-symmetric. The suite can also be used to find selected eigenvalues/eigenvectors of smaller scale dense, complex and non-symmetric problems.

Following a call to nag_complex_sparse_eigensystem_iter (f12apc), nag_complex_sparse_eigensystem_sol (f12aqc) returns the converged approximations to eigenvalues and (optionally) the corresponding approximate eigenvectors and/or an orthonormal basis for the associated approximate invariant subspace. The eigenvalues (and eigenvectors) are selected from those of a standard or generalized eigenvalue problem defined by complex non-symmetric matrices. There is negligible additional cost to obtain eigenvectors; an orthonormal basis is always computed, but there is an additional storage cost if both are requested.

nag_complex_sparse_eigensystem_sol (f12aqc) is based on the function **zneupd** from the ARPACK package, which uses 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 non-symmetric matrices is provided in Lehoucq and Scott (1996). This suite of functions offers the same functionality as the ARPACK software for complex non-symmetric problems, but the interface design is quite different in order to make the option setting clearer to you and to simplify some of the interfaces.

nag_complex_sparse_eigensystem_sol (f12aqc), is a post-processing function that must be called following a successful final exit from nag_complex_sparse_eigensystem_iter (f12apc). nag_complex_sparse_eigensystem_sol (f12aqc) uses data returned from nag_complex_sparse_eigensystem_iter (f12apc) and options, set either by default or explicitly by calling nag_complex_sparse_eigensystem_option (f12arc), to return the converged approximations to selected eigenvalues and (optionally):

– the corresponding approximate eigenvectors;

– an orthonormal basis for the associated approximate invariant subspace;

– both.

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: **nconv** – Integer * *Output*

On exit: the number of converged eigenvalues as found by nag_complex_sparse_eigensystem_option (f12arc).

2: **d**[dim] – Complex *Output*

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

On exit: the first **nconv** locations of the array **d** contain the converged approximate eigenvalues.

3: **z**[dim] – Complex *Output*

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

On exit: if the default option **Vectors** = Ritz (see nag_real_sparse_eigensystem_option (f12adc)) has been selected then **z** contains the final set of eigenvectors corresponding to the eigenvalues held in **d**. The complex eigenvector associated with an eigenvalue is stored in the corresponding column of **z**.

4: **sigma** – Complex *Input*

On entry: if one of the **Shifted** modes (see nag_complex_sparse_eigensystem_option (f12arc)) has been selected then **sigma** contains the shift used; otherwise **sigma** is not referenced.

5: **resid**[dim] – const Complex *Input*

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

On entry: must not be modified following a call to nag_complex_sparse_eigensystem_iter (f12apc) since it contains data required by nag_complex_sparse_eigensystem_sol (f12aqc).

6: **v**[dim] – Complex *Input/Output*

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

The ith element of the jth basis vector is stored in location $\mathbf{v}[j \times \mathbf{n} + i]$, for $i = 0, 1, \ldots, \mathbf{n} - 1$ and $j = 0, 1, \ldots, \mathbf{ncv} - 1$.

On entry: the **ncv** sections of **v**, of length n, contain the Arnoldi basis vectors for OP as constructed by nag_complex_sparse_eigensystem_iter (f12apc).

On exit: if the option **Vectors** = Schur or Ritz has been set and a separate array **z** has been passed (i.e., **z** does not equal **v**), then the first **nconv** sections of **v**, of length n, will contain approximate Schur vectors that span the desired invariant subspace.

7: **comm**[dim] – Complex *Communication Array*

Note: the dimension, *dim*, of the array **comm** must be at least $\max(1, 3 \times \mathbf{n} + 3 \times \mathbf{ncv} \times \mathbf{ncv} + 5 \times \mathbf{ncv} + 60)$ (see nag_complex_sparse_eigensystem_init (f12anc)).

On initial entry: must remain unchanged from the prior call to nag_complex_sparse_eigensystem_init (f12anc).

On exit: contains data on the current state of the solution.

8: **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 from the prior call to nag_complex_sparse_eigensystem_init (f12anc).

On exit: contains data on the current state of the solution.

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

The NAG error argument (see Section 2.6 of the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Error: unable to allocate requested internal workspace.

NE_BAD_PARAM

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

NE_INTERNAL_EIGVEC_FAIL

In calculating eigenvectors, an internal call returned with an error. The function returned with **fail.code** = ⟨*value*⟩. 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 consult NAG for assistance.

NE_INVALID_OPTION

On entry, **Vectors** = Select, but this is not yet implemented.

NE_RITZ_COUNT

Got a different count of the number of converged Ritz values than the value passed to it through the argument **icomm**: number counted = ⟨*value*⟩, number expected = ⟨*value*⟩. This usually indicates that a communication array has been altered or has become corrupted between calls to nag_complex_sparse_eigensystem_iter (f12apc) and nag_complex_sparse_eigensystem_sol (f12aqc).

NE_SCHUR_EIG_FAIL

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

NE_SCHUR_REORDER

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

NE_ZERO_EIGS_FOUND

The number of eigenvalues found to sufficient accuracy, as communicated through the argument **icomm**, is zero. You should experiment with different values of **nev** and **ncv**, or select a different computational mode or increase the maximum number of iterations prior to calling nag_complex_sparse_eigensystem_iter (f12apc).

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 Further Comments

None.

9 Example

The example solves $Ax = \lambda Bx$ in regular-invert mode, where A and B are derived 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.

9.1 Program Text

```
/* nag_complex_sparse_eigensystem_sol (f12aqc) 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 rho = {10.,0.};
static void av(Integer, Complex *, Complex *);
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 h, h4, sigma;
  double estnrm, hr;
  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);

  Vprintf("nag_complex_sparse_eigensystem_sol (f12aqc) Example Program "
          "Results\n");
  /* Skip heading in data file */
  Vscanf("%*[^\n] ");

  Vscanf("%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)) )
    {
      Vprintf("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);
  /* Select the required mode using
     nag_complex_sparse_eigensystem_option (f12arc). */
  nag_complex_sparse_eigensystem_option("REGULAR INVERSE",
                                        icomm, comm, &fail);
  /* Select the problem type using
     nag_complex_sparse_eigensystem_option (f12arc). */
  nag_complex_sparse_eigensystem_option("GENERALIZED", icomm,
                                        comm, &fail);
  hr = 1.0/(double) (n+1);
  /* Assign to Complex type using nag_complex (a02bac) */
  h = nag_complex(hr, 0.0);
  h4 = nag_complex(4.0 * hr, 0.0);

  for (j = 0; j <= n - 2; ++j)
    {
      dl[j] = h;
      dd[j] = h4;
      du[j] = h;
    }
  dd[n - 1] = h4;

  my_zgttrf(n, dl, dd, du, du2, ipiv, &info);
  if (fail.code != NE_NOERROR)
    {
      Vprintf(" Error from nag_zgttrf.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  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  y <--- OP*x = inv[M]*A*x      | */
          av(nx, x, y);
          my_zgttrs(n, dl, dd, du, du2, ipiv, y);
          if (fail.code != NE_NOERROR)
            {
              Vprintf(" Error from nag_zgttrs.\n%s\n", fail.message);
              exit_status = 1;
              goto END;
            }
        }
      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);
          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_complex_sparse_eigensystem_sol
         (f12aqc) to compute eigenvalues. */
      nag_complex_sparse_eigensystem_sol(&nconv, eigv, v, sigma,
                                         resid, v, comm, icomm,
                                         &fail);

      Vprintf("\n");
      Vprintf(" The %4ld", nconv);
      Vprintf(" Ritz values of largest magnitude are:\n\n");
      for (j = 0; j <= nconv-1; ++j)
        {
          Vprintf("%8ld%5s( %12.4f , %12.4f )\n", j+1, "",
                  eigv[j].re , eigv[j].im);
        }
    }
  else
    {
      Vprintf(" Error from nag_complex_sparse_eigensystem_iter (f12apc)."
              "\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
 END:
  if (comm) NAG_FREE(comm);
  if (eigv) NAG_FREE(eigv);
  if (eigest) NAG_FREE(eigest);
  if (dd) NAG_FREE(dd);
  if (dl) NAG_FREE(dl);
  if (du) NAG_FREE(du);
  if (du2) NAG_FREE(du2);
  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 av(Integer nx, Complex *v, Complex *y)
{
  /* Scalars */
  Complex dd, dl, du, z1, z2, z3;
  double hr1, sr;
  Integer j, n;

  /* Function Body */
  n = nx * nx;
  hr1 = (double) (n+1);
  sr = 0.5*rho.re;
  /* Assign to Complex type using nag_complex (a02bac) */
  dd = nag_complex(2.0*hr1,0.0); /* dd = 2.0/h */
  dl = nag_complex(-hr1-sr,0.0); /* dl = -1.0/h - rho/2 */
  du = nag_complex(-hr1+sr,0.0); /* du = -1.0/h + rho/2 */
  /* w[0] = dd*v[0] + du*v[1] */
  /* Compute Complex multiply using nag_complex_multiply (a02ccc). */
  z1 = nag_complex_multiply(dd, v[0]);
  z2 = nag_complex_multiply(du, v[1]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  y[0] = nag_complex_add(z1, z2);
  for (j = 1; j <= n - 2; ++j)
    {
      /* y[j] = dl*V[j-1] + dd*V[j] + du*v[j+1] */
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      z1 = nag_complex_multiply(dl, v[j-1]);
      z2 = nag_complex_multiply(dd, v[j]);
      z3 = nag_complex_multiply(du, v[j+1]);
      /* Compute Complex addition using nag_complex_add
         (a02cac). */
      z1 = nag_complex_add(z1, z2);
      y[j] = nag_complex_add(z1, z3);
    }
  /* y[n-1] = dl*v[n-2] + dd*v[n-1] */
  /* Compute Complex multiply using nag_complex_multiply (a02ccc). */
  z1 = nag_complex_multiply(dl, v[n-2]);
  z2 = nag_complex_multiply(dd, v[n-1]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  y[n-1] = nag_complex_add(z1, z2);
  return;
} /* av */


static void mv(Integer nx, Complex *v, Complex *y)
{
  /* Scalars */
  Complex oneh, fourh, z1, z2;
  double hr;
  Integer j, n;

  /* Function Body */
  n = nx * nx;
  hr = 1.0/(double)(n+1);
  /* Assign to Complex type using nag_complex (a02bac) */
  oneh = nag_complex(hr, 0.0);
  fourh = nag_complex(4.0*hr, 0.0);
  /* y[0] = h*(four*v[0] + one*v[1]) */
  /* Compute Complex multiply using nag_complex_multiply
     (a02ccc). */
  z1 = nag_complex_multiply(fourh, v[0]);
  z2 = nag_complex_multiply(oneh, v[1]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  y[0] = nag_complex_add(z1, z2);
  for (j = 1; j <= n - 2; ++j)
    {
      /* y[j] = h*(one*v[j-1] + four*v[j] + one*v[j+1]) */
      /* Compute Complex multiply using nag_complex_multiply
         (a02ccc). */
      z1 = nag_complex_multiply(fourh, v[j]);
      /* Compute Complex addition using nag_complex_add
         (a02cac). */
      z2 = nag_complex_add(v[j-1], v[j+1]);
      z2 = nag_complex_multiply(oneh, z2);
      y[j] = nag_complex_add(z1, z2);
    }
  /* y[n-1] = h*(one*v[n-2] + four*v[n-1]) */
  /* Compute Complex multiply using nag_complex_multiply
     (a02ccc). */
  z1 = nag_complex_multiply(fourh, v[n-1]);
  z2 = nag_complex_multiply(oneh, v[n-2]);
  /* Compute Complex addition using nag_complex_add (a02cac). */
  y[n-1] = nag_complex_add(z1,z2);
  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;
}
```

9.2 Program Data

```
nag_complex_sparse_eigensystem_sol (f12aqc) Example Program Data
 10  4  20 : Vaues for nx, nev and ncv
```

9.3 Program Results

```
nag_complex_sparse_eigensystem_sol (f12aqc) Example Program Results

 The    4 Ritz values of largest magnitude are:

      1     (   20383.0384 ,      -0.0000 )
      2     (   20338.7563 ,       0.0000 )
      3     (   20265.2844 ,      -0.0000 )
      4     (   20163.1142 ,       0.0000 )
```