NAG Library Function Document

nag_dtgevc (f08ykc)

1 Purpose

nag_dtgevc (f08ykc) computes some or all of the right and/or left generalized eigenvectors of a pair of real matrices (A, B) which are in generalized real Schur form.

2 Specification

```
#include <nag.h>
#include <nagf08.h>
void nag_dtgevc (Nag_OrderType order, Nag_SideType side,
     Nag_HowManyType how_many, const Nag_Boolean select[], Integer n,
     const double a[], Integer pda, const double b[], Integer pdb,
     double vl[], Integer pdvl, double vr[], Integer pdvr, Integer mm,
     Integer *m, NagError *fail)
```

3 Description

nag_dtgevc (f08ykc) computes some or all of the right and/or left generalized eigenvectors of the matrix pair (A, B) which is assumed to be in generalized upper Schur form. If the matrix pair (A, B) is not in the generalized upper Schur form, then nag_dhgeqz (f08xec) should be called before invoking nag_dtgevc (f08ykc).

The right generalized eigenvector x and the left generalized eigenvector y of (A, B) corresponding to a generalized eigenvalue λ are defined by

$$(A - \lambda B)x = 0$$

and

$$y^{\mathrm{H}}(A - \lambda B) = 0.$$

If a generalized eigenvalue is determined as $0/0$, which is due to zero diagonal elements at the same locations in both A and B, a unit vector is returned as the corresponding eigenvector.

Note that the generalized eigenvalues are computed using nag_dhgeqz (f08xec) but nag_dtgevc (f08ykc) does not explicitly require the generalized eigenvalues to compute eigenvectors. The ordering of the eigenvectors is based on the ordering of the eigenvalues as computed by nag_dtgevc (f08ykc).

If all eigenvectors are requested, the function may either return the matrices X and/or Y of right or left eigenvectors of (A, B), or the products ZX and/or QY, where Z and Q are two matrices supplied by you. Usually, Q and Z are chosen as the orthogonal matrices returned by nag_dhgeqz (f08xec). Equivalently, Q and Z are the left and right Schur vectors of the matrix pair supplied to nag_dhgeqz (f08xec). In that case, QY and ZX are the left and right generalized eigenvectors, respectively, of the matrix pair supplied to nag_dhgeqz (f08xec).

A must be block upper triangular; with 1 by 1 and 2 by 2 diagonal blocks. Corresponding to each 2 by 2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part. Each 1 by 1 block gives a real generalized eigenvalue and a corresponding eigenvector.

4 References

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide* (3rd Edition) SIAM, Philadelphia

Golub G H and Van Loan C F (1996) *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

Moler C B and Stewart G W (1973) An algorithm for generalized matrix eigenproblems *SIAM J. Numer. Anal.* **10** 241–256

Stewart G W and Sun J-G (1990) *Matrix Perturbation Theory* Academic Press, London

5 Arguments

1: **order** – Nag_OrderType *Input*

On entry: the **order** argument specifies the two-dimensional storage scheme being used, i.e., row-major ordering or column-major ordering. C language defined storage is specified by **order** = Nag_RowMajor. See Section 3.2.1.3 in the Essential Introduction for a more detailed explanation of the use of this argument.

Constraint: **order** = Nag_RowMajor or Nag_ColMajor.

2: **side** – Nag_SideType *Input*

On entry: specifies the required sets of generalized eigenvectors.

side = Nag_RightSide
Only right eigenvectors are computed.

side = Nag_LeftSide
Only left eigenvectors are computed.

side = Nag_BothSides
Both left and right eigenvectors are computed.

Constraint: **side** = Nag_BothSides, Nag_LeftSide or Nag_RightSide.

3: **how_many** – Nag_HowManyType *Input*

On entry: specifies further details of the required generalized eigenvectors.

how_many = Nag_ComputeAll
All right and/or left eigenvectors are computed.

how_many = Nag_BackTransform
All right and/or left eigenvectors are computed; they are backtransformed using the input matrices supplied in arrays **vr** and/or **vl**.

how_many = Nag_ComputeSelected
Selected right and/or left eigenvectors, defined by the array **select**, are computed.

Constraint: **how_many** = Nag_ComputeAll, Nag_BackTransform or Nag_ComputeSelected.

4: **select**[*dim*] – const Nag_Boolean *Input*

Note: the dimension, *dim*, of the array **select** must be at least

n when **how_many** = Nag_ComputeSelected;
otherwise **select** may be **NULL**.

On entry: specifies the eigenvectors to be computed if **how_many** = Nag_ComputeSelected. To select the generalized eigenvector corresponding to the jth generalized eigenvalue, the jth element of **select** should be set to Nag_TRUE; if the eigenvalue corresponds to a complex conjugate pair,

then real and imaginary parts of eigenvectors corresponding to the complex conjugate eigenvalue pair will be computed.

If **how_many** = Nag_ComputeAll or Nag_BackTransform, **select** is not referenced and may be **NULL**.

Constraint: if **how_many** = Nag_ComputeSelected, **select**$[j]$ = Nag_TRUE or Nag_FALSE, for $j = 0, 1, \ldots, n-1$.

5: **n** – Integer *Input*

On entry: n, the order of the matrices A and B.

Constraint: **n** ≥ 0.

6: **a**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **a** must be at least **pda** $\times$ **n**.

The (i,j)th element of the matrix A is stored in

a$[(j-1) \times$ **pda** $+ i - 1]$ when **order** = Nag_ColMajor;
a$[(i-1) \times$ **pda** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: the matrix pair (A, B) must be in the generalized Schur form. Usually, this is the matrix A returned by nag_dhgeqz (f08xec).

7: **pda** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **a**.

Constraint: **pda** $\geq \max(1,$ **n**$)$.

8: **b**$[dim]$ – const double *Input*

Note: the dimension, *dim*, of the array **b** must be at least **pdb** $\times$ **n**.

The (i,j)th element of the matrix B is stored in

b$[(j-1) \times$ **pdb** $+ i - 1]$ when **order** = Nag_ColMajor;
b$[(i-1) \times$ **pdb** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: the matrix pair (A, B) must be in the generalized Schur form. If A has a 2 by 2 diagonal block then the corresponding 2 by 2 block of B must be diagonal with positive elements. Usually, this is the matrix B returned by nag_dhgeqz (f08xec).

9: **pdb** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **b**.

Constraint: **pdb** $\geq \max(1,$ **n**$)$.

10: **vl**$[dim]$ – double *Input/Output*

Note: the dimension, *dim*, of the array **vl** must be at least

pdvl $\times$ **mm** when **side** = Nag_LeftSide or Nag_BothSides and **order** = Nag_ColMajor;
n $\times$ **pdvl** when **side** = Nag_LeftSide or Nag_BothSides and **order** = Nag_RowMajor;
otherwise **vl** may be **NULL**.

The (i,j)th element of the matrix is stored in

vl$[(j-1) \times$ **pdvl** $+ i - 1]$ when **order** = Nag_ColMajor;
vl$[(i-1) \times$ **pdvl** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: if **how_many** = Nag_BackTransform and **side** = Nag_LeftSide or Nag_BothSides, **vl** must be initialized to an n by n matrix Q. Usually, this is the orthogonal matrix Q of left Schur vectors returned by nag_dhgeqz (f08xec).

On exit: if **side** = Nag_LeftSide or Nag_BothSides, **vl** contains:

if **how_many** = Nag_ComputeAll, the matrix Y of left eigenvectors of (A, B);

if **how_many** = Nag_BackTransform, the matrix QY;

if **how_many** = Nag_ComputeSelected, the left eigenvectors of (A, B) specified by **select**, stored consecutively in the rows or columns (depending on the value of **order**) of the array **vl**, in the same order as their corresponding eigenvalues.

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive rows or columns, the first holding the real part, and the second the imaginary part.

If **side** = Nag_RightSide, **vl** is not referenced and may be **NULL**.

11: **pdvl** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **vl**.

Constraints:

if **order** = Nag_ColMajor,

if **side** = Nag_LeftSide or Nag_BothSides, **pdvl** $\geq$ **n**;
if **side** = Nag_RightSide, **vl** may be **NULL**.;

if **order** = Nag_RowMajor,

if **side** = Nag_LeftSide or Nag_BothSides, **pdvl** $\geq$ **mm**;
if **side** = Nag_RightSide, **vl** may be **NULL**..

12: **vr**[*dim*] – double *Input/Output*

Note: the dimension, *dim*, of the array **vr** must be at least

pdvr $\times$ **mm** when **side** = Nag_RightSide or Nag_BothSides and **order** = Nag_ColMajor;
n $\times$ **pdvr** when **side** = Nag_RightSide or Nag_BothSides and **order** = Nag_RowMajor;
otherwise **vr** may be **NULL**.

The (i, j)th element of the matrix is stored in

vr$[(j-1) \times$ **pdvr** $+ i - 1]$ when **order** = Nag_ColMajor;
vr$[(i-1) \times$ **pdvr** $+ j - 1]$ when **order** = Nag_RowMajor.

On entry: if **how_many** = Nag_BackTransform and **side** = Nag_RightSide or Nag_BothSides, **vr** must be initialized to an n by n matrix Z. Usually, this is the orthogonal matrix Z of right Schur vectors returned by nag_dhgeqz (f08xec).

On exit: if **side** = Nag_RightSide or Nag_BothSides, **vr** contains:

if **how_many** = Nag_ComputeAll, the matrix X of right eigenvectors of (A, B);

if **how_many** = Nag_BackTransform, the matrix ZX;

if **how_many** = Nag_ComputeSelected, the right eigenvectors of (A, B) specified by **select**, stored consecutively in the rows or columns (depending on the value of **order**) of the array **vr**, in the same order as their corresponding eigenvalues.

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive rows or columns, the first holding the real part, and the second the imaginary part.

If **side** = Nag_LeftSide, **vr** is not referenced and may be **NULL**.

13: **pdvr** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **vr**.

Constraints:

if **order** = Nag_ColMajor,

if **side** = Nag_RightSide or Nag_BothSides, **pdvr** $\geq$ **n**;
if **side** = Nag_LeftSide, **vr** may be **NULL**.;
if **order** = Nag_RowMajor,

if **side** = Nag_RightSide or Nag_BothSides, **pdvr** $\geq$ **mm**;
if **side** = Nag_LeftSide, **vr** may be **NULL**..

14: **mm** – Integer *Input*

On entry: the number of columns in the arrays **vl** and/or **vr**.

Constraints:

if **how_many** = Nag_ComputeAll or Nag_BackTransform, **mm** $\geq$ **n**;
if **how_many** = Nag_ComputeSelected, **mm** must not be less than the number of requested eigenvectors.

15: **m** – Integer * *Output*

On exit: the number of columns in the arrays **vl** and/or **vr** actually used to store the eigenvectors. If **how_many** = Nag_ComputeAll or Nag_BackTransform, **m** is set to **n**. Each selected real eigenvector occupies one row or column and each selected complex eigenvector occupies two rows or columns.

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

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

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

NE_CONSTRAINT

On entry, **select**[j] = ⟨*value*⟩ and **how_many** = ⟨*value*⟩.
Constraint: if **how_many** = Nag_ComputeSelected, **select**[j] = Nag_TRUE or Nag_FALSE, for $j = 0, 1, \ldots, n-1$.

NE_ENUM_INT_2

On entry, **how_many** = ⟨*value*⟩, **n** = ⟨*value*⟩ and **mm** = ⟨*value*⟩.
Constraint: if **how_many** = Nag_ComputeAll or Nag_BackTransform, **mm** $\geq$ **n**;
if **how_many** = Nag_ComputeSelected, **mm** must not be less than the number of requested eigenvectors.

On entry, **side** = ⟨*value*⟩, **pdvl** = ⟨*value*⟩, **mm** = ⟨*value*⟩.
Constraint: if **side** = Nag_LeftSide or Nag_BothSides, **pdvl** $\geq$ **mm**.

On entry, **side** = ⟨*value*⟩, **pdvl** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: if **side** = Nag_LeftSide or Nag_BothSides, **pdvl** $\geq$ **n**.

On entry, **side** $= \langle value \rangle$, **pdvr** $= \langle value \rangle$, **mm** $= \langle value \rangle$.
Constraint: if **side** = Nag_RightSide or Nag_BothSides, **pdvr** $\geq$ **mm**.

On entry, **side** $= \langle value \rangle$, **pdvr** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **side** = Nag_RightSide or Nag_BothSides, **pdvr** $\geq$ **n**.

NE_INT

On entry, **n** $= \langle value \rangle$.
Constraint: **n** ≥ 0.

On entry, **pda** $= \langle value \rangle$.
Constraint: **pda** > 0.

On entry, **pdb** $= \langle value \rangle$.
Constraint: **pdb** > 0.

On entry, **pdvl** $= \langle value \rangle$.
Constraint: **pdvl** > 0.

On entry, **pdvr** $= \langle value \rangle$.
Constraint: **pdvr** > 0.

NE_INT_2

On entry, **pda** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: **pda** $\geq \max(1, \mathbf{n})$.

On entry, **pdb** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: **pdb** $\geq \max(1, \mathbf{n})$.

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_NOT_COMPLEX

The 2 by 2 block $(\langle value \rangle : \langle value \rangle + 1)$ does not have complex eigenvalues.

7 Accuracy

It is beyond the scope of this manual to summarise the accuracy of the solution of the generalized eigenvalue problem. Interested readers should consult Section 4.11 of the LAPACK Users' Guide (see Anderson *et al.* (1999)) and Chapter 6 of Stewart and Sun (1990).

8 Parallelism and Performance

nag_dtgevc (f08ykc) is not threaded by NAG in any implementation.

nag_dtgevc (f08ykc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

nag_dtgevc (f08ykc) is the sixth step in the solution of the real generalized eigenvalue problem and is called after nag_dhgeqz (f08xec).

The complex analogue of this function is nag_ztgevc (f08yxc).

10 Example

This example computes the α and β arguments, which defines the generalized eigenvalues and the corresponding left and right eigenvectors, of the matrix pair (A, B) given by

$$A = \begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 & 16.0 & 32.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 243.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \\ 5.0 & 25.0 & 125.0 & 625.0 & 3125.0 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ 1.0 & 4.0 & 9.0 & 16.0 & 25.0 \\ 1.0 & 8.0 & 27.0 & 64.0 & 125.0 \\ 1.0 & 16.0 & 81.0 & 256.0 & 625.0 \\ 1.0 & 32.0 & 243.0 & 1024.0 & 3125.0 \end{pmatrix}.$$

To compute generalized eigenvalues, it is required to call five functions: nag_dggbal (f08whc) to balance the matrix, nag_dgeqrf (f08aec) to perform the QR factorization of B, nag_dormqr (f08agc) to apply Q to A, nag_dgghrd (f08wec) to reduce the matrix pair to the generalized Hessenberg form and nag_dhgeqz (f08xec) to compute the eigenvalues via the QZ algorithm.

The computation of generalized eigenvectors is done by calling nag_dtgevc (f08ykc) to compute the eigenvectors of the balanced matrix pair. The function nag_dggbak (f08wjc) is called to backward transform the eigenvectors to the user-supplied matrix pair. If both left and right eigenvectors are required then nag_dggbak (f08wjc) must be called twice.

10.1 Program Text

```
/* nag_dtgevc (f08ykc) Example Program.
 *
 * Copyright 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 */

#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf06.h>
#include <nagf08.h>
#include <nagf16.h>
#include <nagx04.h>

static Integer normalize_vectors(Nag_OrderType order, Integer n, double qz[],
                                 double alphai[], const char* title);

int main(void)
{
  /* Scalars */
  Integer       i, icols, ihi, ilo, irows, j, m, n, pda, pdb, pdq, pdz;
  Integer       exit_status = 0;
  Nag_Boolean   ileft, iright;

  NagError      fail;
  Nag_OrderType order;
  /* Arrays */
  double        *a = 0, *alphai = 0, *alphar = 0, *b = 0, *beta = 0;
  double        *lscale = 0, *q = 0, *rscale = 0, *tau = 0, *z = 0;

#ifdef NAG_COLUMN_MAJOR
#define A(I, J) a[(J-1)*pda + I - 1]
#define B(I, J) b[(J-1)*pdb + I - 1]
#define Q(I, J) q[(J-1)*pdq + I - 1]
#define Z(I, J) z[(J-1)*pdz + I - 1]
  order = Nag_ColMajor;
#else
#define A(I, J) a[(I-1)*pda + J - 1]
#define B(I, J) b[(I-1)*pdb + J - 1]
#define Q(I, J) q[(I-1)*pdq + J - 1]
#define Z(I, J) z[(I-1)*pdz + J - 1]
  order = Nag_RowMajor;
#endif
```

```
  INIT_FAIL(fail);

  printf("nag_dtgevc (f08ykc) Example Program Results\n\n");

  /* ileft  is Nag_TRUE if left  eigenvectors are required */
  /* iright is Nag_TRUE if right eigenvectors are required */
  ileft = Nag_TRUE;
  iright = Nag_TRUE;

  /* Skip heading in data file */
  scanf("%*[^\n] ");
  scanf("%ld %*[^\n] ", &n);

  pda = n;
  pdb = n;
  pdq = n;
  pdz = n;

  /* Allocate memory */
  if (
      !(a      = NAG_ALLOC(n * n, double)) ||
      !(b      = NAG_ALLOC(n * n, double)) ||
      !(q      = NAG_ALLOC(n * n, double)) ||
      !(z      = NAG_ALLOC(n * n, double)) ||
      !(alphai = NAG_ALLOC(n, double)) ||
      !(alphar = NAG_ALLOC(n, double)) ||
      !(beta   = NAG_ALLOC(n, double)) ||
      !(lscale = NAG_ALLOC(n, double)) ||
      !(rscale = NAG_ALLOC(n, double)) ||
      !(tau    = NAG_ALLOC(n, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* READ matrix A from data file */
  for (i = 1; i <= n; ++i)
    for (j = 1; j <= n; ++j)
      scanf("%lf", &A(i, j));
  scanf("%*[^\n] ");

  /* READ matrix B from data file */
  for (i = 1; i <= n; ++i)
    for (j = 1; j <= n; ++j)
      scanf("%lf", &B(i, j));
  scanf("%*[^\n] ");

  /* Balance the real general matrix pair (A,B) using nag_dggbal (f08whc). */
  nag_dggbal(order, Nag_DoBoth, n, a, pda, b, pdb, &ilo, &ihi, lscale,
             rscale, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dggbal (f08whc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Print matrices A and B after balancing using
   *  nag_gen_real_mat_print (x04cac).
   */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n, a,
                         pda, "Matrix A after balancing", 0, &fail);
  if (fail.code == NE_NOERROR) {
    fflush(stdout);
    nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n, b,
                           pdb, "Matrix B after balancing", 0, &fail);
  }
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 2;
```

```
    goto END;
  }
  printf("\n");

  /* Reduce B to triangular form using QR and multipling both sides by Q^T */
  irows = ihi + 1 - ilo;
  icols = n + 1 - ilo;
  /* nag_dgeqrf (f08aec).
   * QR factorization of real general rectangular matrix
   */
  nag_dgeqrf(order, irows, icols, &B(ilo, ilo), pdb, tau, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgeqrf (f08aec).\n%s\n", fail.message);
    exit_status = 3;
    goto END;
  }

  /* Apply the Q to matrix A -  nag_dormqr (f08agc)
   * as determined by nag_dgeqrf (f08aec).
   */
  nag_dormqr(order, Nag_LeftSide, Nag_Trans, irows, icols, irows,
             &B(ilo, ilo), pdb, tau, &A(ilo, ilo), pda, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dormqr (f08agc).\n%s\n", fail.message);
    exit_status = 4;
    goto END;
  }

  /* Initialize Q (if left eigenvectors are required) */
  if (ileft) {
    /* Q = I. */
    nag_dge_load(order, n, n, 0.0, 1.0, q, pdq, &fail);
      /* Copy B to Q using nag_dge_copy (f16qfc). */
    nag_dge_copy(order, Nag_NoTrans, irows-1, irows-1, &B(ilo+1,ilo), pdb,
                 &Q(ilo+1,ilo), pdq, &fail);
    /* nag_dorgqr (f08afc).
     * Form all or part of orthogonal Q from QR factorization
     * determined by nag_dgeqrf (f08aec) or nag_dgeqpf (f08bec)
     */
    nag_dorgqr(order, irows, irows, irows, &Q(ilo, ilo), pdq, tau, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_dorgqr (f08afc).\n%s\n", fail.message);
      exit_status = 5;
      goto END;
    }
  }

  /* Initialize Z (if right eigenvectors are required) */
  if (iright) {
    /* Z = I. */
    nag_dge_load(order, n, n, 0.0, 1.0, z, pdz, &fail);
  }

  /* Compute the generalized Hessenberg form of (A,B) */
  /* nag_dgghrd (f08wec).
   * Orthogonal reduction of a pair of real general matrices
   * to generalized upper Hessenberg form
   */
  nag_dgghrd(order, Nag_UpdateSchur, Nag_UpdateZ, n, ilo, ihi, a, pda,
             b, pdb, q, pdq, z, pdz, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgghrd (f08wec).\n%s\n", fail.message);
    exit_status = 6;
    goto END;
  }

  /* Matrix A in generalized Hessenberg form */
  /* nag_gen_real_mat_print (x04cac), see above. */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n, a,
                         pda, "Matrix A in Hessenberg form", 0, &fail);
```

```
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 7;
    goto END;
  }
  printf("\n");

  /* Matrix B in generalized Hessenberg form */
  /* nag_gen_real_mat_print (x04cac), see above. */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n, b,
                         pdb, "Matrix B in Hessenberg form", 0, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_dhgeqz (f08xec).
   * Eigenvalues and generalized Schur factorization of real
   * generalized upper Hessenberg form reduced from a pair of
   * real general matrices.
   */
  nag_dhgeqz(order, Nag_Schur, Nag_AccumulateQ, Nag_AccumulateZ, n, ilo, ihi,
             a, pda, b, pdb, alphar, alphai, beta, q, pdq, z, pdz, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dhgeqz (f08xec).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Print the generalized eigenvalue parameters */
  printf("\n Generalized eigenvalues\n");
  for (i = 0; i < n; ++i) {
    if (beta[i] != 0.0) {
      printf(" %4ld     (%7.3f,%7.3f)\n", i+1, alphar[i]/beta[i],
             alphai[i]/beta[i]);
    }
    else
      printf(" %4ldEigenvalue is infinite\n", i+1);
  }
  printf("\n");

  /* Compute left and right generalized eigenvectors
   * of the balanced matrix - nag_dtgevc (f08ykc).
   */
  nag_dtgevc(order, Nag_BothSides, Nag_BackTransform, NULL, n, a, pda,
             b, pdb, q, pdq, z, pdz, n, &m, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dtgevc (f08ykc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  if (iright) {
    /* Compute right eigenvectors of the original matrix pair
     * supplied tonag_dggbal (f08whc) using nag_dggbak (f08wjc).
     */
    nag_dggbak(order, Nag_DoBoth, Nag_RightSide, n, ilo, ihi, lscale,
               rscale, n, z, pdz, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_dggbak (f08wjc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    /* Normalize and print the right eigenvectors */
    exit_status = normalize_vectors(order, n, z, alphai, "Right eigenvectors");
  }
  printf("\n");

  /* Compute left eigenvectors of the original matrix */
  if (ileft) {
```

```
    /* nag_dggbak (f08wjc), see above. */
    nag_dggbak(order, Nag_DoBoth, Nag_LeftSide, n, ilo, ihi, lscale, rscale, n,
               q, pdq, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_dggbak (f08wjc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    /* Normalize the left eigenvectors */
    exit_status = normalize_vectors(order, n, q, alphai, "Left eigenvectors");
  }
 END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(q);
  NAG_FREE(z);
  NAG_FREE(alphai);
  NAG_FREE(alphar);
  NAG_FREE(beta);
  NAG_FREE(lscale);
  NAG_FREE(rscale);
  NAG_FREE(tau);

  return exit_status;
}

static Integer normalize_vectors(Nag_OrderType order, Integer n, double qz[],
                                 double alphai[], const char* title)
{
  /* Real eigenvectors are scaled so that the maximum value of elements is 1.0;
   * each complex eigenvector z[] is normalized so that the element of largest
   * magnitude is scaled to be (1.0,0.0).
   */

  double        a, b, u, v, r, ri;
  Integer       colinc, rowinc, i, j, k, indqz, errors=0;
  NagError      fail;

  INIT_FAIL(fail);

  if (order==Nag_ColMajor) {
    rowinc = 1;
    colinc = n;
  } else {
    rowinc = n;
    colinc = 1;
  }
  indqz = 0;
  for (j=0; j<n; j++) {
    if (alphai[j]>=0.0) {
      if (alphai[j]==0.0) {
        /* Find element of eigenvector with largest absolute value using
         * nag_damax_val (f16jqc).
         */
        nag_damax_val(n, &qz[indqz], rowinc, &k, &r, &fail);
        if (fail.code != NE_NOERROR) {
          printf("Error from nag_damax_val (f16jqc).\n%s\n", fail.message);
          errors = 1;
          goto END;
        }
        r = qz[indqz+k];
        for (i=0; i<n*rowinc; i+=rowinc) {
          qz[indqz+i] = qz[indqz+i]/r;
        }
      } else {
        /* norm of j-th complex eigenvector using nag_dge_norm (f16rac),
         * stored as two arrays of length n.
         */
        k = 0;
        r = -1.0;
        for (i=0; i<n*rowinc;i+=rowinc) {
```

```
          ri = abs(qz[indqz+i])+abs(qz[indqz+colinc+i]);
          if (ri>r) {
            k = i;
            r = ri;
          }
        }
        a = qz[indqz+k];
        b = qz[indqz+colinc+k];
        r = a*a + b*b;

        for (i=0; i<n*rowinc; i+=rowinc) {
          u = qz[indqz+i];
          v = qz[indqz+colinc+i];
          qz[indqz+i] = (u*a + v*b)/r;
          qz[indqz+colinc+i] = (v*a - u*b)/r;
        }
        indqz += colinc;
      }
      indqz += colinc;
    }
  }
  /* Print the normalized eigenvectors using
   * nag_gen_real_mat_print (x04cac)
   */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
                         qz, n, title, 0, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    errors = 1;
  }
 END:
  return errors;
}
```

10.2 Program Data

```
nag_dtgevc (f08ykc) Example Program Data
   5                                                          :Value of N
1.00        1.00        1.00        1.00        1.00
2.00        4.00        8.00       16.00       32.00
3.00        9.00       27.00       81.00      243.00
4.00       16.00       64.00      256.00     1024.00
5.00       25.00      125.00      625.00     3125.00          :End of matrix A
1.00        2.00        3.00        4.00        5.00
1.00        4.00        9.00       16.00       25.00
1.00        8.00       27.00       64.00      125.00
1.00       16.00       81.00      256.00      625.00
1.00       32.00      243.00     1024.00     3125.00          :End of matrix B
```

10.3 Program Results

```
nag_dtgevc (f08ykc) Example Program Results

 Matrix A after balancing
           1          2          3          4          5
 1      1.0000     1.0000     0.1000     0.1000     0.1000
 2      2.0000     4.0000     0.8000     1.6000     3.2000
 3      0.3000     0.9000     0.2700     0.8100     2.4300
 4      0.4000     1.6000     0.6400     2.5600    10.2400
 5      0.5000     2.5000     1.2500     6.2500    31.2500
 Matrix B after balancing
           1          2          3          4          5
 1      1.0000     2.0000     0.3000     0.4000     0.5000
 2      1.0000     4.0000     0.9000     1.6000     2.5000
 3      0.1000     0.8000     0.2700     0.6400     1.2500
 4      0.1000     1.6000     0.8100     2.5600     6.2500
 5      0.1000     3.2000     2.4300    10.2400    31.2500
```

```
Matrix A in Hessenberg form
           1          2          3          4          5
1     -2.1898    -0.3181     2.0547     4.7371    -4.6249
2     -0.8395    -0.0426     1.7132     7.5194   -17.1850
3      0.0000    -0.2846    -1.0101    -7.5927    26.4499
4      0.0000     0.0000     0.0376     1.4070    -3.3643
5      0.0000     0.0000     0.0000     0.3813    -0.9937

Matrix B in Hessenberg form
           1          2          3          4          5
1     -1.4248    -0.3476     2.1175     5.5813    -3.9269
2      0.0000    -0.0782     0.1189     8.0940   -15.2928
3      0.0000     0.0000     1.0021   -10.9356    26.5971
4      0.0000     0.0000     0.0000     0.5820    -0.0730
5      0.0000     0.0000     0.0000     0.0000     0.5321

Generalized eigenvalues
   1     ( -2.437,  0.000)
   2     (  0.607,  0.795)
   3     (  0.607, -0.795)
   4     (  1.000,  0.000)
   5     ( -0.410,  0.000)

Right eigenvectors
        1       2       3       4       5
1  -0.4655  1.0000  0.0000 -0.5469 -0.5106
2   1.0000 -0.7945 -0.5235  1.0000  1.0000
3  -0.9428  0.2277  0.2509 -0.7383 -0.7350
4   0.4126 -0.0300 -0.0700  0.1953  0.2343
5  -0.0662  0.0014  0.0102 -0.0273 -0.0261

Left eigenvectors
        1       2       3       4       5
1  -0.5106  1.0000  0.0000 -0.5469 -0.4655
2   1.0000 -0.7945 -0.5235  1.0000  1.0000
3  -0.7350  0.2277  0.2509 -0.7383 -0.9428
4   0.2343 -0.0300 -0.0700  0.1953  0.4126
5  -0.0261  0.0014  0.0102 -0.0273 -0.0662
```