NAG Library Function Document

nag_ztgevc (f08yxc)

1 Purpose

nag_ztgevc (f08yxc) computes some or all of the right and/or left generalized eigenvectors of a pair of complex upper triangular matrices (A, B).

2 Specification

```
#include <nag.h>
#include <nagf08.h>

void nag_ztgevc (Nag_OrderType order, Nag_SideType side,
     Nag_HowManyType how_many, const Nag_Boolean select[], Integer n,
     const Complex a[], Integer pda, const Complex b[], Integer pdb,
     Complex vl[], Integer pdvl, Complex vr[], Integer pdvr, Integer mm,
     Integer *m, NagError *fail)
```

3 Description

nag_ztgevc (f08yxc) computes some or all of the right and/or left generalized eigenvectors of the matrix pair (A, B) which is assumed to be in upper triangular form. If the matrix pair (A, B) is not upper triangular then the function nag_zhgeqz (f08xsc) should be called before invoking nag_ztgevc (f08yxc).

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_zhgeqz (f08xsc) but nag_ztgevc (f08yxc) 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_ztgevc (f08yxc).

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 unitary matrices returned by nag_zhgeqz (f08xsc). Equivalently, Q and Z are the left and right Schur vectors of the matrix pair supplied to nag_zhgeqz (f08xsc). In that case, QY and ZX are the left and right generalized eigenvectors, respectively, of the matrix pair supplied to nag_zhgeqz (f08xsc).

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 2.3.1.3 in How to Use the NAG Library and its Documentation 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.

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 Complex *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 A must be in upper triangular form. Usually, this is the matrix A returned by nag_zhgeqz (f08xsc).

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 Complex *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 B must be in upper triangular form with non-negative real diagonal elements. Usually, this is the matrix B returned by nag_zhgeqz (f08xsc).

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*] – Complex *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 ith element of the jth vector 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 unitary matrix Q of left Schur vectors returned by nag_zhgeqz (f08xsc).

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.

11: **pdvl** – Integer *Input*

On entry: the stride used 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] – Complex *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 ith element of the jth vector 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 unitary 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.

13: **pdvr** – Integer *Input*

On entry: the stride used 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 eigenvector occupies one column.

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

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

NE_CONSTRAINT

On entry, **how_many** = ⟨*value*⟩ and **select**[j] = ⟨*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** = ⟨*value*⟩, **pdvr** = ⟨*value*⟩, **mm** = ⟨*value*⟩.
Constraint: if **side** = Nag_RightSide or Nag_BothSides, **pdvr** $\geq$ **mm**.

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

NE_INT

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 0.

On entry, **pda** = ⟨*value*⟩.
Constraint: **pda** > 0.

On entry, **pdb** = ⟨*value*⟩.
Constraint: **pdb** > 0.

On entry, **pdvl** = ⟨*value*⟩.
Constraint: **pdvl** > 0.

On entry, **pdvr** = ⟨*value*⟩.
Constraint: **pdvr** > 0.

NE_INT_2

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

On entry, **pdb** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

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_ztgevc (f08yxc) 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 x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

nag_ztgevc (f08yxc) is the sixth step in the solution of the complex generalized eigenvalue problem and is usually called after nag_zhgeqz (f08xsc).

The real analogue of this function is nag_dtgevc (f08ykc).

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

$$A = \begin{pmatrix} 1.0+3.0i & 1.0+4.0i & 1.0+5.0i & 1.0+6.0i \\ 2.0+2.0i & 4.0+3.0i & 8.0+4.0i & 16.0+5.0i \\ 3.0+1.0i & 9.0+2.0i & 27.0+3.0i & 81.0+4.0i \\ 4.0+0.0i & 16.0+1.0i & 64.0+2.0i & 256.0+3.0i \end{pmatrix}$$

and

$$B = \begin{pmatrix} 1.0+0.0i & 2.0+1.0i & 3.0+2.0i & 4.0+3.0i \\ 1.0+1.0i & 4.0+2.0i & 9.0+3.0i & 16.0+4.0i \\ 1.0+2.0i & 8.0+3.0i & 27.0+4.0i & 64.0+5.0i \\ 1.0+3.0i & 16.0+4.0i & 81.0+5.0i & 256.0+6.0i \end{pmatrix}.$$

To compute generalized eigenvalues, it is required to call five functions: nag_zggbal (f08wvc) to balance the matrix, nag_zgeqrf (f08asc) to perform the QR factorization of B, nag_zunmqr (f08auc) to apply Q to A, nag_zgghrd (f08wsc) to reduce the matrix pair to the generalized Hessenberg form and nag_zhgeqz (f08xsc) to compute the eigenvalues via the QZ algorithm.

The computation of generalized eigenvectors is done by calling nag_ztgevc (f08yxc) to compute the eigenvectors of the balanced matrix pair. The function nag_zggbak (f08wwc) is called to backward transform the eigenvectors to the user-supplied matrix pair. If both left and right eigenvectors are required then nag_zggbak (f08wwc) must be called twice.

10.1 Program Text

```
/* nag_ztgevc (f08yxc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <naga02.h>
#include <nagf06.h>
#include <nagf08.h>
#include <nagf16.h>
#include <nagm01.h>
#include <nagx04.h>
#include <naga02.h>

#ifdef __cplusplus
extern "C"
{
#endif
  static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b);
  static Integer normalize_vectors(Nag_OrderType order, Integer n, Complex qz[],
                                   Complex tz[], double temp[], size_t rank[],
                                   const char *title);
  static Integer sort_values (Integer n, Complex alpha[], Complex beta[],
                              size_t rank[], double temp[]);
#ifdef __cplusplus
}
#endif

int main(void)
{
  /* Scalars */
  Integer       i, icols, ihi, ilo, irows, j, m, n, pda, pdb, pdq, pdz;
  Integer       exit_status = 0, prbal = 0, prhess = 0;
  Complex       one, zero;
  /* Arrays */
  Complex       *a = 0, *alpha = 0, *b = 0, *beta = 0, *q = 0, *tau = 0;
  Complex       *z = 0;
  double        *lscale = 0, *rscale = 0, *temp = 0;
  size_t        *rank = 0;
  /* Nag Types */
  Nag_Boolean   ileft, iright;
  NagError      fail;
  Nag_OrderType order;

#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]
  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]
  order = Nag_RowMajor;
#endif

  INIT_FAIL(fail);

  printf("nag_ztgevc (f08yxc) Example Program Results\n\n");

  /* ileft  is true if left  eigenvectors are required;
   * iright is true if right eigenvectors are required.
```

```
    */
   ileft = Nag_TRUE;
   iright = Nag_TRUE;
   zero = nag_complex(0.0, 0.0);
   one = nag_complex(1.0, 0.0);

   /* Skip heading in data file and read matrix size. */
#ifdef _WIN32
   scanf_s("%*[^\n] ");
#else
   scanf("%*[^\n] ");
#endif
#ifdef _WIN32
   scanf_s("%" NAG_IFMT "%*[^\n] ", &n);
#else
   scanf("%" NAG_IFMT "%*[^\n] ", &n);
#endif

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

   /* Allocate memory */
   if (!(a = NAG_ALLOC(n * n, Complex)) ||
       !(b = NAG_ALLOC(n * n, Complex)) ||
       !(q = NAG_ALLOC(n * n, Complex)) ||
       !(z = NAG_ALLOC(n * n, Complex)) ||
       !(alpha = NAG_ALLOC(n, Complex)) ||
       !(beta = NAG_ALLOC(n, Complex)) ||
       !(tau = NAG_ALLOC(n, Complex)) ||
       !(temp = NAG_ALLOC(n, double)) ||
       !(rank = NAG_ALLOC(n, size_t)) ||
       !(lscale = NAG_ALLOC(n, double)) || !(rscale = 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)
#ifdef _WIN32
       scanf_s(" ( %lf , %lf )", &A(i, j).re, &A(i, j).im);
#else
       scanf(" ( %lf , %lf )", &A(i, j).re, &A(i, j).im);
#endif
#ifdef _WIN32
   scanf_s("%*[^\n] ");
#else
   scanf("%*[^\n] ");
#endif

   /* READ matrix B from data file */
   for (i = 1; i <= n; ++i)
     for (j = 1; j <= n; ++j)
#ifdef _WIN32
       scanf_s(" ( %lf , %lf )", &B(i, j).re, &B(i, j).im);
#else
       scanf(" ( %lf , %lf )", &B(i, j).re, &B(i, j).im);
#endif
#ifdef _WIN32
   scanf_s("%*[^\n] ");
#else
   scanf("%*[^\n] ");
#endif

   /* Balance pair (A,B) of complex general matrices using
    * nag_zggbal (f08wvc).
    */
```

```
  nag_zggbal(order, Nag_DoBoth, n, a, pda, b, pdb, &ilo, &ihi, lscale,
             rscale, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zggbal (f08wvc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  if (prbal) {
    /* Print complex general matrices A and B after balancing using
     * nag_gen_complx_mat_print_comp (x04dbc).
     */
    fflush(stdout);
    nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n,
                                  n, a, pda, Nag_BracketForm, "%7.4f",
                                  "Matrix A after balancing",
                                  Nag_IntegerLabels, 0, Nag_IntegerLabels, 0,
                                  80, 0, 0, &fail);
    if (fail.code == NE_NOERROR) {
      fflush(stdout);
      nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag,
                                    n, n, b, pdb, Nag_BracketForm, "%7.4f",
                                    "Matrix B after balancing",
                                    Nag_IntegerLabels, 0, Nag_IntegerLabels, 0,
                                    80, 0, 0, &fail);
    }
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }
    printf("\n");
  }

  /* Reduce B to triangular form using QR and premultiply A by Q^H. */
  irows = ihi + 1 - ilo;
  icols = n + 1 - ilo;
  /* nag_zgeqrf (f08asc).
   * QR factorization of complex general rectangular matrix B.
   */
  nag_zgeqrf(order, irows, icols, &B(ilo, ilo), pdb, tau, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zgeqrf (f08asc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Apply the orthogonal transformation Q^H to matrix A using
   * nag_zunmqr (f08auc).
   */
  nag_zunmqr(order, Nag_LeftSide, Nag_ConjTrans, irows, icols, irows,
             &B(ilo, ilo), pdb, tau, &A(ilo, ilo), pda, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zunmqr (f08auc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Initialize Q (if left eigenvectors are required) */
  if (ileft) {
    /* Q = I */
    nag_zge_load(order, n, n, zero, one, q, pdq, &fail);
    /* Q = B using nag_zge_copy (f16tfc). */
    nag_zge_copy(order, Nag_NoTrans, irows - 1, irows - 1, &B(ilo + 1, ilo),
                 pdb, &Q(ilo + 1, ilo), pdq, &fail);
    /* Form Q from QR factorization using nag_zungqr (f08atc). */
    nag_zungqr(order, irows, irows, irows, &Q(ilo, ilo), pdq, tau, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_zungqr (f08atc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  }

  if (iright) {
    /* Z = I. */
    nag_zge_load(order, n, n, zero, one, z, pdz, &fail);
  }

  /* Compute the generalized Hessenberg form of (A,B) by Unitary reduction
   * using nag_zgghrd (f08wsc).
   */
  nag_zgghrd(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_zgghrd (f08wsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  if (prhess) {
    /* Print generalized Hessenberg form of (A,B) using
     * nag_gen_complx_mat_print_comp (x04dbc).
     */
    fflush(stdout);
    nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n,
                                  n, a, pda, Nag_BracketForm, "%7.3f",
                                  "Matrix A in Hessenberg form",
                                  Nag_IntegerLabels, 0, Nag_IntegerLabels, 0,
                                  80, 0, 0, &fail);
    if (fail.code == NE_NOERROR) {
      printf("\n");

      fflush(stdout);
      nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag,
                                    n, n, b, pdb, Nag_BracketForm, "%7.3f",
                                    "Matrix B in Hessenberg form",
                                    Nag_IntegerLabels, 0, Nag_IntegerLabels, 0,
                                    80, 0, 0, &fail);
    }
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }
  }

  /* Compute the generalized Schur form - nag_zhgeqz (f08xsc).
   * Eigenvalues and generalized Schur factorization of
   * complex generalized upper Hessenberg form reduced from a
   * pair of complex general matrices
   */
  nag_zhgeqz(order, Nag_Schur, Nag_AccumulateQ, Nag_AccumulateZ, n, ilo, ihi,
             a, pda, b, pdb, alpha, beta, q, pdq, z, pdz, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zhgeqz (f08xsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Print the ordered generalized eigenvalues if finite */
  exit_status = sort_values(n, alpha, beta, rank, temp);
  if (exit_status) {
    goto END;
  }

  /* nag_ztgevc (f08yxc).
   * Left and right eigenvectors of a pair of complex upper
   * triangular matrices
   */
  nag_ztgevc(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_ztgevc (f08yxc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  if (iright) {
    /* nag_zggbak (f08wwc).
     * Transform eigenvectors of a pair of complex balanced
     * matrices to those of original matrix pair supplied to
     * nag_zggbal (f08wvc)
     */
    nag_zggbak(order, Nag_DoBoth, Nag_RightSide, n, ilo, ihi, lscale,
               rscale, n, z, pdz, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_zggbak (f08wwc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

    /* Normalize and print the right eigenvectors */
    exit_status = normalize_vectors(order, n, z, a, temp, rank,
                                    "Right eigenvectors");
    printf("\n");
  }

  /* Compute left eigenvectors of the original matrix */
  if (ileft) {
    /* nag_zggbak (f08wwc), see above. */
    nag_zggbak(order, Nag_DoBoth, Nag_LeftSide, n, ilo, ihi, lscale,
               rscale, n, q, pdq, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_zggbak (f08wwc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

    /* Normalize and print the left eigenvectors */
    exit_status = normalize_vectors(order, n, q, a, temp, rank,
                                    "Left eigenvectors");
  }
END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(q);
  NAG_FREE(z);
  NAG_FREE(alpha);
  NAG_FREE(beta);
  NAG_FREE(tau);
  NAG_FREE(temp);
  NAG_FREE(rank);
  NAG_FREE(lscale);
  NAG_FREE(rscale);

  return exit_status;
}

static Integer normalize_vectors(Nag_OrderType order, Integer n, Complex qz[],
                                 Complex tz[], double temp[], size_t rank[],
                                 const char *title)
{
  /* Each complex eigenvector is normalized so that the element of largest
   * magnitude is scaled to be real and positive.
   */

  Complex   scal;
  double    norm, r;
  Integer   i, j, k, errors = 0;
  NagError  fail;

  INIT_FAIL(fail);
```

```
#ifdef NAG_COLUMN_MAJOR
#define Z(I, J) qz[(J-1)*n + I - 1]
#define T(I, J) tz[(J-1)*n + I - 1]
  order = Nag_ColMajor;
#else
#define Z(I, J) qz[(I-1)*n + J - 1]
#define T(I, J) tz[(I-1)*n + J - 1]
  order = Nag_RowMajor;
#endif

  for (j = 1; j <=n; j++) {
    /* Find element of eigenvector with largest absolute value using
     * nag_complex_abs (a02dbc) and nag_dmax_val (f16jnc).
     */
    norm = 0.0;
    for (i = 1; i <= n; i++) {
      temp[i-1] = nag_complex_abs(Z(i,j));
      norm = norm + temp[i-1]*temp[i-1];
    }
    norm = sqrt(norm);
    nag_dmax_val(n, temp, 1, &k, &r, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_dmax_val (f16jnc).\n%s\n", fail.message);
      errors = 1;
      goto END;
    }
    k = k + 1;
    scal.re = Z(k,j).re/r/norm;
    scal.im = -Z(k,j).im/r/norm;
    for (i = 1; i <= n; i++)
      Z(i, j) = nag_complex_multiply(Z(i, j), scal);
    Z(k, j).im = 0.0;

  }
  for (j = 1; j <=n; j++) {
    k = (Integer) rank[j-1];
    for (i = 1; i <= n; i++) {
      T(i,j) = Z(i,k+1);
    }
  }
  /* Print the normalized eigenvectors using
   * nag_gen_complx_mat_print_comp (x04dbc)
   */
  fflush(stdout);
  nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag,
                                n, n, tz, n, Nag_BracketForm, "%7.4f",
                                title, Nag_IntegerLabels, 0,
                                Nag_IntegerLabels, 0, 80, 0, 0, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n",
           fail.message);
    errors = 3;
  }
END:
  return errors;
}

static Integer sort_values (Integer n, Complex alpha[], Complex beta[],
                            size_t rank[], double temp[])
{
  Integer  i, errors = 0, isinf = 0;
  Complex  e;
  NagError fail;

  INIT_FAIL(fail);

  /* Accumulate eigenvalue modulii in temp. */
  for (i = 0; i < n; ++i) {
    if (beta[i].re != 0.0) {
      /* nag_complex_divide (a02cdc) - Quotient of two complex numbers. */
      e = nag_complex_divide(alpha[i], beta[i]);
      temp[i] = nag_complex_abs(e);
      alpha[i] = e;
    } else {
      isinf = i;
      printf(" %4" NAG_IFMT "Eigenvalue is infinite\n", isinf + 1);
      goto END;
    }
  }
  /* Rank sort eigenvalues by absolute values using nag_rank_sort (m01dsc). */
  nag_rank_sort((Pointer) temp, (size_t) n, (ptrdiff_t) (sizeof(double)),
                compare, Nag_Descending, rank, &fail);
  /* Turn ranks into indices using nag_make_indices (m01zac). */
  nag_make_indices(rank, (size_t) n, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_make_indices (m01zac).\n%s\n", fail.message);
    errors = 1;
    goto END;
  }
  /* Sort eigenvalues using nag_reorder_vector (m01esc). */
  nag_reorder_vector((Pointer) alpha, (size_t) n, sizeof(Complex),
                     (ptrdiff_t) sizeof(Complex), rank, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_reorder_vector (m01esc).\n%s\n", fail.message);
    errors = 2;
    goto END;
  }
  printf("\n Generalized eigenvalues\n");
  for (i = 0; i < n; ++i) {
    e = alpha[i];
    printf(" %4" NAG_IFMT "     (%7.3f,%7.3f)\n", i + 1, e.re, e.im);
  }
  printf("\n");
 END:
  return errors;
}

static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b)
{
  double x = *((const double *) a) - *((const double *) b);
  return (x < 0.0 ? -1 : (x == 0.0 ? 0 : 1));
}
```

10.2 Program Data

```
nag_ztgevc (f08yxc) Example Program Data
   4                                                          :Value of N
(   1.00, 3.00)  (  1.00, 4.00)  (  1.00, 5.00)  (  1.00, 6.00)
(   2.00, 2.00)  (  4.00, 3.00)  (  8.00, 4.00)  ( 16.00, 5.00)
(   3.00, 1.00)  (  9.00, 2.00)  ( 27.00, 3.00)  ( 81.00, 4.00)
(   4.00, 0.00)  ( 16.00, 1.00)  ( 64.00, 2.00)  (256.00, 3.00)  :End of matrix A
(   1.00, 0.00)  (  2.00, 1.00)  (  3.00, 2.00)  (  4.00, 3.00)
(   1.00, 1.00)  (  4.00, 2.00)  (  9.00, 3.00)  ( 16.00, 4.00)
(   1.00, 2.00)  (  8.00, 3.00)  ( 27.00, 4.00)  ( 64.00, 5.00)
(   1.00, 3.00)  ( 16.00, 4.00)  ( 81.00, 5.00)  (256.00, 6.00)  :End of matrix B
```

10.3 Program Results

```
nag_ztgevc (f08yxc) Example Program Results


 Generalized eigenvalues
    1     ( -0.635,  1.653)
    2     (  0.493,  0.910)
    3     (  0.458, -0.843)
    4     (  0.674, -0.050)

 Right eigenvectors
                   1                  2                  3                  4
 1  ( 0.7186, 0.0000)  (-0.3946, 0.0246)  (-0.4649, 0.0156)  (-0.6788,-0.1233)
 2  (-0.6208,-0.2009)  ( 0.7921, 0.0000)  ( 0.7652, 0.0000)  ( 0.7184, 0.0000)
 3  ( 0.2251, 0.0762)  (-0.4554, 0.0334)  (-0.4275,-0.0912)  (-0.0886,-0.0067)
 4  (-0.0372,-0.0088)  ( 0.0824,-0.0322)  ( 0.0707, 0.0442)  (-0.0048, 0.0006)

 Left eigenvectors
                    1                  2                  3                  4
 1  ( 0.7397, 0.0000)  (-0.3240,-0.1559)  (-0.3722,-0.0016)  (-0.4118,-0.2276)
 2  (-0.5812, 0.2589)  ( 0.8063, 0.0000)  ( 0.8003, 0.0000)  ( 0.8681, 0.0000)
 3  ( 0.1875,-0.1097)  (-0.4523, 0.0903)  (-0.4606,-0.0279)  (-0.1564, 0.0136)
 4  (-0.0219, 0.0195)  ( 0.0755,-0.0453)  ( 0.0839, 0.0311)  ( 0.0206,-0.0038)
```