

NAG Library Function Document

nag_dggevx (f08wbc)

1 Purpose

nag_dggevx (f08wbc) computes for a pair of n by n real nonsymmetric matrices (A, B) the generalized eigenvalues and, optionally, the left and/or right generalized eigenvectors using the QZ algorithm.

Optionally it also computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors, reciprocal condition numbers for the eigenvalues, and reciprocal condition numbers for the right eigenvectors.

2 Specification

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

void nag_dggevx (Nag_OrderType order, Nag_BalanceType balanc,
                Nag_LeftVecsType jobvl, Nag_RightVecsType jobvr, Nag_RCondType sense,
                Integer n, double a[], Integer pda, double b[], Integer pdb,
                double alphas[], double alphas_i[], double betas[], double vl[],
                Integer pdvl, double vr[], Integer pdvr, Integer *ilo, Integer *ihi,
                double lscale[], double rscale[], double *abnorm, double *bbnorm,
                double rconde[], double rcondv[], NagError *fail)
```

3 Description

A generalized eigenvalue for a pair of matrices (A, B) is a scalar λ or a ratio $\alpha/\beta = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta = 0$, and even for both being zero.

The right eigenvector v_j corresponding to the eigenvalue λ_j of (A, B) satisfies

$$Av_j = \lambda_j Bv_j.$$

The left eigenvector u_j corresponding to the eigenvalue λ_j of (A, B) satisfies

$$u_j^H A = \lambda_j u_j^H B,$$

where u_j^H is the conjugate-transpose of u_j .

All the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem $Ax = \lambda Bx$, where A and B are real, square matrices, are determined using the QZ algorithm. The QZ algorithm consists of four stages:

1. A is reduced to upper Hessenberg form and at the same time B is reduced to upper triangular form.
2. A is further reduced to quasi-triangular form while the triangular form of B is maintained. This is the real generalized Schur form of the pair (A, B) .
3. The quasi-triangular form of A is reduced to triangular form and the eigenvalues extracted. This function does not actually produce the eigenvalues λ_j , but instead returns α_j and β_j such that

$$\lambda_j = \alpha_j / \beta_j, \quad j = 1, 2, \dots, n.$$

The division by β_j becomes your responsibility, since β_j may be zero, indicating an infinite eigenvalue. Pairs of complex eigenvalues occur with α_j / β_j and $\alpha_{j+1} / \beta_{j+1}$ complex conjugates, even though α_j and α_{j+1} are not conjugate.

4. If the eigenvectors are required they are obtained from the triangular matrices and then transformed back into the original coordinate system.

For details of the balancing option, see Section 3 in nag_dggbal (f08whc).

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 <http://www.netlib.org/lapack/lug>

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

Wilkinson J H (1979) Kronecker's canonical form and the *QZ* algorithm *Linear Algebra Appl.* **28** 285–303

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: **balanc** – Nag_BalanceType *Input*

On entry: specifies the balance option to be performed.

balanc = Nag_NoBalancing

Do not diagonally scale or permute.

balanc = Nag_BalancePermute

Permute only.

balanc = Nag_BalanceScale

Scale only.

balanc = Nag_BalanceBoth

Both permute and scale.

Computed reciprocal condition numbers will be for the matrices after permuting and/or balancing. Permuting does not change condition numbers (in exact arithmetic), but balancing does. In the absence of other information, **balanc** = Nag_BalanceBoth is recommended.

Constraint: **balanc** = Nag_NoBalancing, Nag_BalancePermute, Nag_BalanceScale or Nag_BalanceBoth.

3: **jobvl** – Nag_LeftVecsType *Input*

On entry: if **jobvl** = Nag_NotLeftVecs, do not compute the left generalized eigenvectors.

If **jobvl** = Nag_LeftVecs, compute the left generalized eigenvectors.

Constraint: **jobvl** = Nag_NotLeftVecs or Nag_LeftVecs.

4: **jobvr** – Nag_RightVecsType *Input*

On entry: if **jobvr** = Nag_NotRightVecs, do not compute the right generalized eigenvectors.

If **jobvr** = Nag_RightVecs, compute the right generalized eigenvectors.

Constraint: **jobvr** = Nag_NotRightVecs or Nag_RightVecs.

- 5: **sense** – Nag_RCondType *Input*
On entry: determines which reciprocal condition numbers are computed.
sense = Nag_NotRCond
 None are computed.
sense = Nag_RCondEigVals
 Computed for eigenvalues only.
sense = Nag_RCondEigVecs
 Computed for eigenvectors only.
sense = Nag_RCondBoth
 Computed for eigenvalues and eigenvectors.
Constraint: **sense** = Nag_NotRCond, Nag_RCondEigVals, Nag_RCondEigVecs or Nag_RCondBoth.
- 6: **n** – Integer *Input*
On entry: n , the order of the matrices A and B .
Constraint: $n \geq 0$.
- 7: **a**[dim] – double *Input/Output*
Note: the dimension, dim , of the array **a** must be at least $\max(1, pda \times n)$.
 Where $\mathbf{A}(i, j)$ appears in this document, it refers to the array element

$$\mathbf{a}[(j-1) \times pda + i - 1] \text{ when } \mathbf{order} = \text{Nag_ColMajor};$$

$$\mathbf{a}[(i-1) \times pda + j - 1] \text{ when } \mathbf{order} = \text{Nag_RowMajor}.$$
On entry: the matrix A in the pair (A, B) .
On exit: **a** has been overwritten. If **jobvl** = Nag_LeftVecs or **jobvr** = Nag_RightVecs or both, then A contains the first part of the real Schur form of the ‘balanced’ versions of the input A and B .
- 8: **pda** – Integer *Input*
On entry: the stride separating row or column elements (depending on the value of **order**) in the array **a**.
Constraint: $pda \geq \max(1, n)$.
- 9: **b**[dim] – double *Input/Output*
Note: the dimension, dim , of the array **b** must be at least $\max(1, pdb \times n)$.
 Where $\mathbf{B}(i, j)$ appears in this document, it refers to the array element

$$\mathbf{b}[(j-1) \times pdb + i - 1] \text{ when } \mathbf{order} = \text{Nag_ColMajor};$$

$$\mathbf{b}[(i-1) \times pdb + j - 1] \text{ when } \mathbf{order} = \text{Nag_RowMajor}.$$
On entry: the matrix B in the pair (A, B) .
On exit: **b** has been overwritten.
- 10: **pdb** – Integer *Input*
On entry: the stride separating row or column elements (depending on the value of **order**) in the array **b**.
Constraint: $pdb \geq \max(1, n)$.
- 11: **alphar**[n] – double *Output*
On exit: the element **alphar**[$j - 1$] contains the real part of α_j .

- 12: **alphai[n]** – double *Output*
On exit: the element **alphai**[$j - 1$] contains the imaginary part of α_j .
- 13: **beta[n]** – double *Output*
On exit: (**alpha**[$j - 1$] + **alphai**[$j - 1$] $\times i$)/**beta**[$j - 1$], for $j = 1, 2, \dots, \mathbf{n}$, will be the generalized eigenvalues.
 If **alphai**[$j - 1$] is zero, then the j th eigenvalue is real; if positive, then the j th and ($j + 1$)st eigenvalues are a complex conjugate pair, with **alphai**[j] negative.
Note: the quotients **alpha**[$j - 1$]/**beta**[$j - 1$] and **alphai**[$j - 1$]/**beta**[$j - 1$] may easily overflow or underflow, and **beta**[$j - 1$] may even be zero. Thus, you should avoid naively computing the ratio α_j/β_j . However, $\max(|\alpha_j|)$ will always be less than and usually comparable with $\|\mathbf{a}\|_2$ in magnitude, and $\max(|\beta_j|)$ will always be less than and usually comparable with $\|\mathbf{b}\|_2$.
- 14: **vl[dim]** – double *Output*
Note: the dimension, *dim*, of the array **vl** must be at least
 $\max(1, \mathbf{pdvl} \times \mathbf{n})$ when **jobvl** = Nag_LeftVecs;
 1 otherwise.
 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 exit: if **jobvl** = Nag_LeftVecs, the left generalized eigenvectors u_j are stored one after another in the columns of **vl**, in the same order as the corresponding eigenvalues. Each eigenvector will be scaled so the largest component will have |real part| + |imag. part| = 1.
 If **jobvl** = Nag_NotLeftVecs, **vl** is not referenced.
- 15: **pdvl** – Integer *Input*
On entry: the stride separating row or column elements (depending on the value of **order**) in the array **vl**.
Constraints:
 if **jobvl** = Nag_LeftVecs, **pdvl** \geq $\max(1, \mathbf{n})$;
 otherwise **pdvl** \geq 1.
- 16: **vr[dim]** – double *Output*
Note: the dimension, *dim*, of the array **vr** must be at least
 $\max(1, \mathbf{pdvr} \times \mathbf{n})$ when **jobvr** = Nag_RightVecs;
 1 otherwise.
 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 exit: if **jobvr** = Nag_RightVecs, the right generalized eigenvectors v_j are stored one after another in the columns of **vr**, in the same order as the corresponding eigenvalues. Each eigenvector will be scaled so the largest component will have |real part| + |imag. part| = 1.
 If **jobvr** = Nag_NotRightVecs, **vr** is not referenced.
- 17: **pdvr** – Integer *Input*
On entry: the stride separating row or column elements (depending on the value of **order**) in the array **vr**.

Constraints:

if **jobvr** = Nag_RightVecs, **pdvr** \geq max(1, **n**);
otherwise **pdvr** \geq 1.

- 18: **ilo** – Integer * *Output*
19: **ihi** – Integer * *Output*

On exit: **ilo** and **ihi** are integer values such that $\mathbf{A}(i, j) = 0$ and $\mathbf{B}(i, j) = 0$ if $i > j$ and $j = 1, 2, \dots, \mathbf{ilo} - 1$ or $i = \mathbf{ihi} + 1, \dots, \mathbf{n}$.

If **balanc** = Nag_NoBalancing or Nag_BalanceScale, **ilo** = 1 and **ihi** = **n**.

- 20: **lscale**[**n**] – double *Output*

On exit: details of the permutations and scaling factors applied to the left side of *A* and *B*.

If pl_j is the index of the row interchanged with row j , and dl_j is the scaling factor applied to row j , then:

lscale[$j - 1$] = pl_j , for $j = 1, 2, \dots, \mathbf{ilo} - 1$;

lscale = dl_j , for $j = \mathbf{ilo}, \dots, \mathbf{ihi}$;

lscale = pl_j , for $j = \mathbf{ihi} + 1, \dots, \mathbf{n}$.

The order in which the interchanges are made is **n** to **ihi** + 1, then 1 to **ilo** - 1.

- 21: **rscale**[**n**] – double *Output*

On exit: details of the permutations and scaling factors applied to the right side of *A* and *B*.

If pr_j is the index of the column interchanged with column j , and dr_j is the scaling factor applied to column j , then:

rscale[$j - 1$] = pr_j , for $j = 1, 2, \dots, \mathbf{ilo} - 1$;

if **rscale** = dr_j , for $j = \mathbf{ilo}, \dots, \mathbf{ihi}$;

if **rscale** = pr_j , for $j = \mathbf{ihi} + 1, \dots, \mathbf{n}$.

The order in which the interchanges are made is **n** to **ihi** + 1, then 1 to **ilo** - 1.

- 22: **abnorm** – double * *Output*

On exit: the 1-norm of the balanced matrix *A*.

- 23: **bbnorm** – double * *Output*

On exit: the 1-norm of the balanced matrix *B*.

- 24: **rconde**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **rconde** must be at least max(1, **n**).

On exit: if **sense** = Nag_RCondEigVals or Nag_RCondBoth, the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of **rconde** are set to the same value. Thus **rconde**[$j - 1$], **rcondv**[$j - 1$], and the j th columns of **vl** and **vr** all correspond to the j th eigenpair.

If **sense** = Nag_RCondEigVecs, **rconde** is not referenced.

- 25: **rcondv**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **rcondv** must be at least max(1, **n**).

On exit: if **sense** = Nag_RCondEigVecs or Nag_RCondBoth, the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of **rcondv** are set to the same value.

If **sense** = Nag_RCondEigVals, **rcondv** is not referenced.

- 26: **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.
 See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ had an illegal value.

NE_EIGENVECTORS

A failure occurred in nag_dtgevc (f08yk) while computing generalized eigenvectors.

NE_ENUM_INT_2

On entry, **jobvl** = $\langle value \rangle$, **pdvl** = $\langle value \rangle$ and **n** = $\langle value \rangle$.
 Constraint: if **jobvl** = Nag_LeftVecs, **pdvl** $\geq \max(1, \mathbf{n})$;
 otherwise **pdvl** ≥ 1 .

On entry, **jobvr** = $\langle value \rangle$, **pdvr** = $\langle value \rangle$ and **n** = $\langle value \rangle$.
 Constraint: if **jobvr** = Nag_RightVecs, **pdvr** $\geq \max(1, \mathbf{n})$;
 otherwise **pdvr** ≥ 1 .

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_ITERATION_QZ

The QZ iteration failed. No eigenvectors have been calculated but **alphar**[j], **alphai**[j] and **beta**[j] should be correct from element $\langle value \rangle$.

The QZ iteration failed with an unexpected error, please contact NAG.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

7 Accuracy

The computed eigenvalues and eigenvectors are exact for a nearby matrices $(A + E)$ and $(B + F)$, where

$$\|(E, F)\|_F = O(\epsilon)\|(A, B)\|_F,$$

and ϵ is the *machine precision*.

An approximate error bound on the chordal distance between the i th computed generalized eigenvalue w and the corresponding exact eigenvalue λ is

$$\epsilon \times \|\mathbf{abnrm}, \mathbf{bbnrm}\|_2 / \mathbf{rconde}[i - 1].$$

An approximate error bound for the angle between the i th computed eigenvector u_j or v_j is given by

$$\epsilon \times \|\mathbf{abnrm}, \mathbf{bbnrm}\|_2 / \mathbf{rcondv}[i - 1].$$

For further explanation of the reciprocal condition numbers **rconde** and **rcondv**, see Section 4.11 of Anderson *et al.* (1999).

Note: interpretation of results obtained with the QZ algorithm often requires a clear understanding of the effects of small changes in the original data. These effects are reviewed in Wilkinson (1979), in relation to the significance of small values of α_j and β_j . It should be noted that if α_j and β_j are **both** small for any j , it may be that no reliance can be placed on **any** of the computed eigenvalues $\lambda_i = \alpha_i / \beta_i$. You are recommended to study Wilkinson (1979) and, if in difficulty, to seek expert advice on determining the sensitivity of the eigenvalues to perturbations in the data.

8 Parallelism and Performance

nag_dggevx (f08wbc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_dggevx (f08wbc) 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

The total number of floating-point operations is proportional to n^3 .

The complex analogue of this function is nag_zggevx (f08wpc).

10 Example

This example finds all the eigenvalues and right eigenvectors of the matrix pair (A, B) , where

$$A = \begin{pmatrix} 3.9 & 12.5 & -34.5 & -0.5 \\ 4.3 & 21.5 & -47.5 & 7.5 \\ 4.3 & 21.5 & -43.5 & 3.5 \\ 4.4 & 26.0 & -46.0 & 6.0 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1.0 & 2.0 & -3.0 & 1.0 \\ 1.0 & 3.0 & -5.0 & 4.0 \\ 1.0 & 3.0 & -4.0 & 3.0 \\ 1.0 & 3.0 & -4.0 & 4.0 \end{pmatrix},$$

together with estimates of the condition number and forward error bounds for each eigenvalue and eigenvector. The option to balance the matrix pair is used.

10.1 Program Text

```

/* nag_dggevx (f08wbc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

#include <math.h>
#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf08.h>
#include <nagx02.h>
#include <naga02.h>
int main(void)
{
    /* Scalars */
    Complex      eig, eigl, eigr;
    double       abnorm, abnorm, bbnrm, eps, sign, small, tol;
    Integer      i, ihi, ilo, j, k, n, pda, pdb, pdvl, pdvr;
    Integer      exit_status = 0;

    /* Arrays */
    double       *a = 0, *alpha_i = 0, *alpha_r = 0, *b = 0, *beta = 0;
    double       *lscale = 0, *rconde = 0, *rcondv = 0, *rscale = 0;
    double       *vl = 0, *vr = 0;
    char         nag_enum_arg[40];

    /* Nag Types */
    NagError     fail;
    Nag_OrderType order;
    Nag_LeftVecsType jobvl;
    Nag_RightVecsType jobvr;
    Nag_RCondType sense;

#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 VL(I, J) vl[(J-1)*pdvl + I - 1]
#define VR(I, J) vr[(J-1)*pdvr + 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 VL(I, J) vl[(I-1)*pdvl + J - 1]
#define VR(I, J) vr[(I-1)*pdvr + J - 1]
    order = Nag_RowMajor;
#endif

    INIT_FAIL(fail);

    printf("nag_dggevx (f08wbc) Example Program Results\n");

    /* Skip heading in data file */

```



```

#ifdef _WIN32
    scanf_s("%*[\n]");
#else
    scanf("%*[\n]");
#endif
#ifdef _WIN32
    scanf_s("%"NAG_IFMT"%*[\n]", &n);
#else
    scanf("%"NAG_IFMT"%*[\n]", &n);
#endif
    if (n < 0)
    {
        printf("Invalid n\n");
        exit_status = 1;
        goto END;
    }
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, _countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    /* nag_enum_name_to_value (x04nac).
     * Converts NAG enum member name to value
     */
    jobvl = (Nag_LeftVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, _countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    jobvr = (Nag_RightVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, _countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    sense = (Nag_RCondType) nag_enum_name_to_value(nag_enum_arg);

    pda = n;
    pdb = n;
    pdvl = (jobvl==Nag_LeftVecs?n:1);
    pdvr = (jobvr==Nag_RightVecs?n:1);

    /* Allocate memory */
    if (!(a      = NAG_ALLOC(n*n, double)) ||
        !(alphai = NAG_ALLOC(n, double)) ||
        !(alphar = NAG_ALLOC(n, double)) ||
        !(b      = NAG_ALLOC(n*n, double)) ||
        !(beta   = NAG_ALLOC(n, double)) ||
        !(lscale = NAG_ALLOC(n, double)) ||
        !(rconde = NAG_ALLOC(n, double)) ||
        !(rcondv = NAG_ALLOC(n, double)) ||
        !(rscale = NAG_ALLOC(n, double)) ||
        !(vl     = NAG_ALLOC(pdvl*pdvl, double)) ||
        !(vr     = NAG_ALLOC(pdvr*pdvr, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }

    /* Read in the matrices A and B */
    for (i = 1; i <= n; ++i)
#ifdef _WIN32
        for (j = 1; j <= n; ++j) scanf_s("%lf", &A(i, j));
#else
        for (j = 1; j <= n; ++j) scanf("%lf", &A(i, j));
#endif
    #endif
    scanf_s("%*[\n]");
    #else

```

```

    scanf("%*[\n]");
#endif
    for (i = 1; i <= n; ++i)
#ifdef _WIN32
        for (j = 1; j <= n; ++j) scanf_s("%lf", &B(i, j));
#else
        for (j = 1; j <= n; ++j) scanf("%lf", &B(i, j));
#endif
#ifdef _WIN32
    scanf_s("%*[\n]");
#else
    scanf("%*[\n]");
#endif

/* Solve the generalized eigenvalue problem using nag_dggevx (f08wbc). */
nag_dggevx(order, Nag_BalanceBoth, jobvl, jobvr, sense, n, a, pda, b, pdb,
           alphas, alphas_i, beta, vl, pdvl, vr, pdvr, &ilo, &ihi, lscale,
           rscale, &abnorm, &bbnorm, rconde, rcondv, &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_dggevx (f08wbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* nag_real_safe_small_number (x02amc), nag_machine_precision (x02ajc) */
eps = nag_machine_precision;
small = nag_real_safe_small_number;
if (abnorm == 0.0)
    abnorm = ABS(bbnorm);
else if (bbnorm == 0.0)
    abnorm = ABS(abnorm);
else if (ABS(abnorm) >= ABS(bbnorm))
    abnorm = ABS(abnorm)*sqrt(1.0+(bbnorm/abnorm)*(bbnorm/abnorm));
else
    abnorm = ABS(bbnorm)*sqrt(1.0+(abnorm/bbnorm)*(abnorm/bbnorm));

tol = eps * abnorm;

/* Print out eigenvalues and vectors and associated condition
 * number and bounds.
 */
for (j = 0; j < n; ++j)
{
    /* Print out information on the jth eigenvalue */
    printf("\n");
    if ((fabs(alphas[j]) + fabs(alphas_i[j])) * small >= fabs(beta[j]))
    {
        printf("Eigenvalue %2"NAG_IFMT" is numerically infinite or "
              "undetermined\n", j+1);
        printf("alpha = (%13.4e, %13.4e), beta = %13.4e\n", alphas[j],
              alphas_i[j], beta[j]);
    }
    else if (alphas_i[j] == 0.0)
    {
        printf("Eigenvalue %2"NAG_IFMT" = %13.4e\n", j+1, alphas[j]/beta[j]);
    }
    else
    {
        eig.re = alphas[j]/beta[j], eig.im = alphas_i[j]/beta[j];
        printf("Eigenvalue %2"NAG_IFMT" = (%13.4e, %13.4e)\n", j+1, eig.re,
              eig.im);
    }
    if (sense==Nag_RCondEigVals || sense==Nag_RCondBoth) {
        printf("\n Reciprocal condition number = %10.1e\n", rconde[j]);

        if ( rconde[j] > 0.0)
            printf(" Error bound = %10.1e\n", tol/rconde[j]);
        else
            printf(" Error bound is infinite\n");
    }
}

```

```

printf("\n\n");
/* Normalize and print out information on the jth eigenvector(s) */
if (jobvl==Nag_LeftVecs) printf("%21s%8s", "Left Eigenvector", "");
if (jobvr==Nag_RightVecs) printf("%21s", "Right Eigenvector");
printf(" %2"NAG_IFMT"\n", j+1);
if (alpha[i] == 0.0)
  for (i = 1; i <= n; ++i) {
    if (jobvl==Nag_LeftVecs)
      printf("%7s%13.4e%12s", "", VL(i, j+1)/VL(n, j+1), "");
    if (jobvr==Nag_RightVecs)
      printf("%7s%13.4e", "", VR(i, j+1)/VR(n, j+1));
    printf("\n");
  }
else
  {
    k = (alpha[i]>0.0?j+1:j);
    sign = (alpha[i]>0.0?1.0:-1.0);
    if (jobvl==Nag_LeftVecs) eigl = nag_complex(VL(n,k), VL(n,k+1));
    if (jobvr==Nag_RightVecs) eigr = nag_complex(VR(n,k), VR(n,k+1));
    for (i = 1; i <= n; ++i)
      {
        if (jobvl==Nag_LeftVecs) {
          eig = nag_complex_divide(nag_complex(VL(i,k), VL(i,k+1)),
                                   eigl);
          printf(" (%13.4e,%13.4e) ", eig.re, sign*eig.im);
        }
        if (jobvr==Nag_RightVecs) {
          eig = nag_complex_divide(nag_complex(VR(i,k), VR(i,k+1)),
                                   eigr);
          printf(" (%13.4e,%13.4e)", eig.re, sign*eig.im);
        }
        printf("\n");
      }
  }

if (sense==Nag_RCondEigVecs || sense==Nag_RCondBoth) {
  printf("\n Reciprocal condition number = %10.1e\n", rcondv[j]);

  if (rcondv[j] > 0.0)
    printf(" Error bound                = %10.1e\n\n", tol/rcondv[j]);
  else
    printf(" Error bound is infinite\n\n");
}
}
}

END:
NAG_FREE(a);
NAG_FREE(alpha);
NAG_FREE(alphar);
NAG_FREE(b);
NAG_FREE(beta);
NAG_FREE(lscale);
NAG_FREE(rconde);
NAG_FREE(rcondv);
NAG_FREE(rscale);
NAG_FREE(vl);
NAG_FREE(vr);

return exit_status;
}

```

10.2 Program Data

nag_dggevx (f08wbc) Example Program Data

```

4                : n

Nag_NotLeftVecs  : jobvl
Nag_RightVecs   : jobvr
Nag_RCondBoth    : sense

```

```

3.9  12.5 -34.5 -0.5
4.3  21.5 -47.5  7.5
4.3  21.5 -43.5  3.5
4.4  26.0 -46.0  6.0 : matrix A

1.0  2.0 -3.0  1.0
1.0  3.0 -5.0  4.0
1.0  3.0 -4.0  3.0
1.0  3.0 -4.0  4.0 : matrix B

```

10.3 Program Results

nag_dggevx (f08wbc) Example Program Results

```

Eigenvalue 1 =      2.0000e+00

Reciprocal condition number =      9.5e-02
Error bound                  =      2.5e-14

Right Eigenvector 1
  1.5909e+01
  9.0909e-02
  1.0000e+00
  1.0000e+00

Reciprocal condition number =      1.3e-01
Error bound                  =      1.9e-14

Eigenvalue 2 = (      3.0000e+00,      4.0000e+00)

Reciprocal condition number =      1.7e-01
Error bound                  =      1.4e-14

Right Eigenvector 2
(      3.0000e+00,      4.0000e+00)
(      6.0000e-01,      8.0000e-01)
(      1.0000e+00,      3.9126e-16)
(      1.0000e+00,     -0.0000e+00)

Reciprocal condition number =      3.8e-02
Error bound                  =      6.2e-14

Eigenvalue 3 = (      3.0000e+00,     -4.0000e+00)

Reciprocal condition number =      1.7e-01
Error bound                  =      1.4e-14

Right Eigenvector 3
(      3.0000e+00,     -4.0000e+00)
(      6.0000e-01,     -8.0000e-01)
(      1.0000e+00,     -3.9126e-16)
(      1.0000e+00,      0.0000e+00)

Reciprocal condition number =      3.8e-02
Error bound                  =      6.2e-14

Eigenvalue 4 =      4.0000e+00

Reciprocal condition number =      5.1e-01
Error bound                  =      4.6e-15

Right Eigenvector 4
  6.4286e+00

```

7.1429e-02
-2.1429e-01
1.0000e+00

Reciprocal condition number = 7.1e-02
Error bound = 3.3e-14
