

# NAG Library Function Document

## nag\_dggesx (f08xbc)

### 1 Purpose

nag\_dggesx (f08xbc) computes the generalized eigenvalues, the generalized real Schur form  $(S, T)$  and, optionally, the left and/or right generalized Schur vectors for a pair of  $n$  by  $n$  real nonsymmetric matrices  $(A, B)$ .

Estimates of condition numbers for selected generalized eigenvalue clusters and Schur vectors are also computed.

### 2 Specification

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

void nag_dggesx (Nag_OrderType order, Nag_LeftVecsType jobvsl,
                Nag_RightVecsType jobvsr, Nag_SortEigValsType sort,
                Nag_Boolean (*selctg)(double ar, double ai, double b),
                Nag_RCondType sense, Integer n, double a[], Integer pda, double b[],
                Integer pdb, Integer *sdim, double alphas[], double alphasl[],
                double betas[], double vsl[], Integer pdvsl, double vsr[], Integer pdvsr,
                double rconde[], double rcondv[], NagError *fail)
```

### 3 Description

The generalized real Schur factorization of  $(A, B)$  is given by

$$A = QSZ^T, \quad B = QTZ^T,$$

where  $Q$  and  $Z$  are orthogonal,  $T$  is upper triangular and  $S$  is upper quasi-triangular with 1 by 1 and 2 by 2 diagonal blocks. The generalized eigenvalues,  $\lambda$ , of  $(A, B)$  are computed from the diagonals of  $T$  and  $S$  and satisfy

$$Az = \lambda Bz,$$

where  $z$  is the corresponding generalized eigenvector.  $\lambda$  is actually returned as the pair  $(\alpha, \beta)$  such that

$$\lambda = \alpha/\beta$$

since  $\beta$ , or even both  $\alpha$  and  $\beta$  can be zero. The columns of  $Q$  and  $Z$  are the left and right generalized Schur vectors of  $(A, B)$ .

Optionally, nag\_dggesx (f08xbc) can order the generalized eigenvalues on the diagonals of  $(S, T)$  so that selected eigenvalues are at the top left. The leading columns of  $Q$  and  $Z$  then form an orthonormal basis for the corresponding eigenspaces, the deflating subspaces.

nag\_dggesx (f08xbc) computes  $T$  to have non-negative diagonal elements, and the 2 by 2 blocks of  $S$  correspond to complex conjugate pairs of generalized eigenvalues. The generalized Schur factorization, before reordering, is computed by the  $QZ$  algorithm.

The reciprocals of the condition estimates, the reciprocal values of the left and right projection norms, are returned in **rconde**[0] and **rconde**[1] respectively, for the selected generalized eigenvalues, together with reciprocal condition estimates for the corresponding left and right deflating subspaces, in **rcondv**[0] and **rcondv**[1]. See Section 4.11 of Anderson *et al.* (1999) for further information.

## 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

## 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: **jobvsl** – Nag\_LeftVecsType *Input*  
*On entry:* if **jobvsl** = Nag\_NotLeftVecs, do not compute the left Schur vectors.  
 If **jobvsl** = Nag\_LeftVecs, compute the left Schur vectors.  
*Constraint:* **jobvsl** = Nag\_NotLeftVecs or Nag\_LeftVecs.
- 3: **jobvsr** – Nag\_RightVecsType *Input*  
*On entry:* if **jobvsr** = Nag\_NotRightVecs, do not compute the right Schur vectors.  
 If **jobvsr** = Nag\_RightVecs, compute the right Schur vectors.  
*Constraint:* **jobvsr** = Nag\_NotRightVecs or Nag\_RightVecs.
- 4: **sort** – Nag\_SortEigValsType *Input*  
*On entry:* specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.  
**sort** = Nag\_NoSortEigVals  
 Eigenvalues are not ordered.  
**sort** = Nag\_SortEigVals  
 Eigenvalues are ordered (see **selctg**).  
*Constraint:* **sort** = Nag\_NoSortEigVals or Nag\_SortEigVals.
- 5: **selctg** – function, supplied by the user *External Function*  
 If **sort** = Nag\_SortEigVals, **selctg** is used to select generalized eigenvalues to be moved to the top left of the generalized Schur form.  
 If **sort** = Nag\_NoSortEigVals, **selctg** is not referenced by nag\_dggesx (f08xbc), and may be specified as NULLFN.

The specification of **selctg** is:

```
Nag_Boolean selctg (double ar, double ai, double b)
```

1:	<b>ar</b> – double	<i>Input</i>
2:	<b>ai</b> – double	<i>Input</i>
3:	<b>b</b> – double	<i>Input</i>

*On entry:* an eigenvalue  $(\mathbf{ar}[j-1] + \sqrt{-1} \times \mathbf{ai}[j-1])/\mathbf{b}[j-1]$  is selected if **selectg**(**ar**[*j* – 1], **ai**[*j* – 1], **b**[*j* – 1]) is Nag\_TRUE. If either one of a complex conjugate pair is selected, then both complex generalized eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex generalized eigenvalue may no longer satisfy **selectg**(**ar**[*j* – 1], **ai**[*j* – 1], **b**[*j* – 1]) = Nag\_TRUE after ordering. **fail.code** = NE\_SCHUR\_REORDER\_SELECT in this case.

6: **sense** – Nag\_RCondType *Input*

*On entry:* determines which reciprocal condition numbers are computed.

**sense** = Nag\_NotRCond  
None are computed.

**sense** = Nag\_RCondEigVals  
Computed for average of selected eigenvalues only.

**sense** = Nag\_RCondEigVecs  
Computed for selected deflating subspaces only.

**sense** = Nag\_RCondBoth  
Computed for both.

If **sense** = Nag\_RCondEigVals, Nag\_RCondEigVecs or Nag\_RCondBoth,  
**sort** = Nag\_SortEigVals.

*Constraint:* **sense** = Nag\_NotRCond, Nag\_RCondEigVals, Nag\_RCondEigVecs or Nag\_RCondBoth.

7: **n** – Integer *Input*

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

*Constraint:* **n** ≥ 0.

8: **a**[*dim*] – double *Input/Output*

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

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

**a**[(*j* – 1) × **pda** + *i* – 1] when **order** = Nag\_ColMajor;  
**a**[(*i* – 1) × **pda** + *j* – 1] when **order** = Nag\_RowMajor.

*On entry:* the first of the pair of matrices, *A*.

*On exit:* **a** has been overwritten by its generalized Schur form *S*.

9: **pda** – Integer *Input*

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

*Constraint:* **pda** ≥ max(1, **n**).

10: **b**[*dim*] – double *Input/Output*

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

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

**b**[(*j* – 1) × **pdb** + *i* – 1] when **order** = Nag\_ColMajor;  
**b**[(*i* – 1) × **pdb** + *j* – 1] when **order** = Nag\_RowMajor.

*On entry:* the second of the pair of matrices,  $B$ .

*On exit:*  $\mathbf{b}$  has been overwritten by its generalized Schur form  $T$ .

11: **pdb** – Integer *Input*

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

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

12: **sdim** – Integer \* *Output*

*On exit:* if **sort** = Nag\_NoSortEigVals, **sdim** = 0.

If **sort** = Nag\_SortEigVals, **sdim** = number of eigenvalues (after sorting) for which **selectg** is Nag\_TRUE. (Complex conjugate pairs for which **selectg** is Nag\_TRUE for either eigenvalue count as 2.)

13: **alphar**[ $\mathbf{n}$ ] – double *Output*

*On exit:* see the description of **beta**.

14: **alphai**[ $\mathbf{n}$ ] – double *Output*

*On exit:* see the description of **beta**.

15: **beta**[ $\mathbf{n}$ ] – double *Output*

*On exit:* ( $\mathbf{alphar}[j-1] + \mathbf{alphai}[j-1] \times i$ )/ $\mathbf{beta}[j-1]$ , for  $j = 1, 2, \dots, \mathbf{n}$ , will be the generalized eigenvalues.  $\mathbf{alphar}[j-1] + \mathbf{alphai}[j-1] \times i$ , and  $\mathbf{beta}[j-1]$ , for  $j = 1, 2, \dots, \mathbf{n}$ , are the diagonals of the complex Schur form ( $S, T$ ) that would result if the 2 by 2 diagonal blocks of the real Schur form of  $(A, B)$  were further reduced to triangular form using 2 by 2 complex unitary transformations.

If  $\mathbf{alphai}[j-1]$  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  $\mathbf{alphai}[j]$  negative.

**Note:** the quotients  $\mathbf{alphar}[j-1]/\mathbf{beta}[j-1]$  and  $\mathbf{alphai}[j-1]/\mathbf{beta}[j-1]$  may easily overflow or underflow, and  $\mathbf{beta}[j-1]$  may even be zero. Thus, you should avoid naively computing the ratio  $\alpha/\beta$ . However, **alphar** and **alphai** will always be less than and usually comparable with  $\|A\|_2$  in magnitude, and **beta** will always be less than and usually comparable with  $\|B\|_2$ .

16: **vsl**[ $\mathit{dim}$ ] – double *Output*

**Note:** the dimension,  $\mathit{dim}$ , of the array **vsl** must be at least

$\max(1, \mathbf{pdvsl} \times \mathbf{n})$  when **jobvsl** = Nag\_LeftVecs;  
1 otherwise.

The  $i$ th element of the  $j$ th vector is stored in

$\mathbf{vsl}[(j-1) \times \mathbf{pdvsl} + i - 1]$  when **order** = Nag\_ColMajor;  
 $\mathbf{vsl}[(i-1) \times \mathbf{pdvsl} + j - 1]$  when **order** = Nag\_RowMajor.

*On exit:* if **jobvsl** = Nag\_LeftVecs, **vsl** will contain the left Schur vectors,  $Q$ .

If **jobvsl** = Nag\_NotLeftVecs, **vsl** is not referenced.

17: **pdvsl** – Integer *Input*

*On entry:* the stride used in the array **vsl**.

*Constraints:*

if **jobvsl** = Nag\_LeftVecs,  $\mathbf{pdvsl} \geq \max(1, \mathbf{n})$ ;  
otherwise  $\mathbf{pdvsl} \geq 1$ .

- 18: **vsr**[*dim*] – double *Output*
- Note:** the dimension, *dim*, of the array **vsr** must be at least  
 $\max(1, \mathbf{pdvsr} \times \mathbf{n})$  when **jobvsr** = Nag\_RightVecs;  
 1 otherwise.
- The *i*th element of the *j*th vector is stored in  
 $\mathbf{vsr}[(j-1) \times \mathbf{pdvsr} + i - 1]$  when **order** = Nag\_ColMajor;  
 $\mathbf{vsr}[(i-1) \times \mathbf{pdvsr} + j - 1]$  when **order** = Nag\_RowMajor.
- On exit:* if **jobvsr** = Nag\_RightVecs, **vsr** will contain the right Schur vectors, *Z*.  
 If **jobvsr** = Nag\_NotRightVecs, **vsr** is not referenced.
- 19: **pdvsr** – Integer *Input*
- On entry:* the stride used in the array **vsr**.
- Constraints:*  
 if **jobvsr** = Nag\_RightVecs,  $\mathbf{pdvsr} \geq \max(1, \mathbf{n})$ ;  
 otherwise  $\mathbf{pdvsr} \geq 1$ .
- 20: **rconde**[2] – double *Output*
- On exit:* if **sense** = Nag\_RCondEigVals or Nag\_RCondBoth, **rconde**[0] and **rconde**[1] contain the reciprocal condition numbers for the average of the selected eigenvalues.  
 If **sense** = Nag\_NotRCond or Nag\_RCondEigVecs, **rconde** is not referenced.
- 21: **rcondv**[2] – double *Output*
- On exit:* if **sense** = Nag\_RCondEigVecs or Nag\_RCondBoth, **rcondv**[0] and **rcondv**[1] contain the reciprocal condition numbers for the selected deflating subspaces.  
 if **sense** = Nag\_NotRCond or Nag\_RCondEigVals, **rcondv** is not referenced.
- 22: **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\_ENUM\_INT\_2

On entry, **jobvsl** = *<value>*, **pdvsl** = *<value>* and **n** = *<value>*.

Constraint: if **jobvsl** = Nag\_LeftVecs,  $\mathbf{pdvsl} \geq \max(1, \mathbf{n})$ ;  
 otherwise  $\mathbf{pdvsl} \geq 1$ .

On entry, **jobvsr** = *<value>*, **pdvsr** = *<value>* and **n** = *<value>*.

Constraint: if **jobvsr** = Nag\_RightVecs,  $\mathbf{pdvsr} \geq \max(1, \mathbf{n})$ ;  
 otherwise  $\mathbf{pdvsr} \geq 1$ .

**NE\_INT**

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

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

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

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

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

**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 2.7.6 in How to Use the NAG Library and its Documentation for further information.

**NE\_ITERATION\_QZ**

The *QZ* iteration failed. No eigenvectors have been calculated but **alphar**[*j*], **alpha**[*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 2.7.5 in How to Use the NAG Library and its Documentation for further information.

**NE\_SCHUR\_REORDER**

The eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned).

**NE\_SCHUR\_REORDER\_SELECT**

After reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy **selectg** = Nag.TRUE. This could also be caused by underflow due to scaling.

**7 Accuracy**

The computed generalized Schur factorization satisfies

$$A + E = QSZ^T, \quad B + F = QTZ^T,$$

where

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

and  $\epsilon$  is the *machine precision*. See Section 4.11 of Anderson *et al.* (1999) for further details.

## 8 Parallelism and Performance

nag\_dggesx (f08xbc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag\_dggesx (f08xbc) 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\_zggesx (f08xpc).

## 10 Example

This example finds the generalized Schur factorization of the matrix pair  $(A, B)$ , where

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

such that the real positive eigenvalues of  $(A, B)$  correspond to the top left diagonal elements of the generalized Schur form,  $(S, T)$ . Estimates of the condition numbers for the selected eigenvalue cluster and corresponding deflating subspaces are also returned.

### 10.1 Program Text

```

/* nag_dggesx (f08xbc) 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 <nagf08.h>
#include <nagf16.h>
#include <nagx02.h>
#include <nagx04.h>

#ifdef __cplusplus
extern "C"
{
#endif
    static Nag_Boolean NAG_CALL selctg(const double ar, const double ai,
                                       const double b);
#ifdef __cplusplus
}
#endif

int main(void)
{
    /* Scalars */

```

```

double abnorm, dg_a, dg_b, eps, norma, normb, normd, norme, tol;
Integer i, j, n, sdim, pda, pdb, pdc, pdd, pde, pdvsl, pdvsr;
Integer exit_status = 0;

/* Arrays */
double *a = 0, *alpha_i = 0, *alpha_r = 0, *b = 0, *beta = 0;
double *c = 0, *d = 0, *e = 0, *vsl = 0, *vsr = 0;
double rconde[2], rcondv[2];
char nag_enum_arg[40];

/* Nag Types */
NagError fail;
Nag_OrderType order;
Nag_LeftVecsType jobvsl;
Nag_RightVecsType jobvsr;
Nag_SortEigValsType sort = Nag_SortEigVals;
Nag_RCondType sense = Nag_RCondBoth;

#ifdef NAG_COLUMN_MAJOR
#define A(I, J) a[(J-1)*pda + I - 1]
#define B(I, J) b[(J-1)*pdb + 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]
    order = Nag_RowMajor;
#endif

    INIT_FAIL(fail);

    printf("nag_dggesx (f08xbc) Example Program Results\n\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;
        return exit_status;
    }
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, (unsigned)_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
     */
    jobvsl = (Nag_LeftVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    jobvsr = (Nag_RightVecsType) nag_enum_name_to_value(nag_enum_arg);
#ifdef _WIN32
    scanf_s(" %39s%*[\n]", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
    scanf(" %39s%*[\n]", nag_enum_arg);
#endif
    sense = (Nag_RCondType) nag_enum_name_to_value(nag_enum_arg);

    pdvsl = (jobvsl == Nag_LeftVecs ? n : 1);

```



```

pdvsr = (jobvsr == Nag_RightVecs ? n : 1);
pda = n;
pdb = n;
pdc = n;
pdd = n;
pde = n;
/* Allocate memory */
if (!(a = NAG_ALLOC(n * n, double)) ||
    !(b = NAG_ALLOC(n * n, double)) ||
    !(c = NAG_ALLOC(n * n, double)) ||
    !(d = NAG_ALLOC(n * n, double)) ||
    !(e = NAG_ALLOC(n * n, double)) ||
    !(alpha = NAG_ALLOC(n, double)) ||
    !(alpha = NAG_ALLOC(n, double)) ||
    !(beta = NAG_ALLOC(n, double)) ||
    !(vsl = NAG_ALLOC(pdvsr * pdvsr, double)) ||
    !(vsr = NAG_ALLOC(pdvsr * pdvsr, 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
#ifdef _WIN32
    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

/* Copy matrices A and B to matrices D and E using nag_dge_copy (f16qfc),
 * real valued general matrix copy.
 * The copies will be used as comparison against reconstructed matrices.
 */
nag_dge_copy(order, Nag_NoTrans, n, n, a, pda, d, pdd, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dge_copy (f16qfc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
nag_dge_copy(order, Nag_NoTrans, n, n, b, pdb, e, pde, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dge_copy (f16qfc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* nag_dge_norm (f16rac): Find norms of input matrices A and B. */
nag_dge_norm(order, Nag_FrobeniusNorm, n, n, a, pda, &norma, &fail);
if (fail.code != NE_NOERROR) {

```

```

    printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
nag_dge_norm(order, Nag_FrobeniusNorm, n, n, b, pdb, &normb, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* nag_gen_real_mat_print (x04cac): Print Matrices A and B. */
fflush(stdout);
nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
    a, pda, "Matrix A", 0, &fail);
printf("\n");
if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
fflush(stdout);
nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
    b, pdb, "Matrix B", 0, &fail);
printf("\n");
if (fail.code != NE_NOERROR) {
    printf("Error from nag_gen_real_mat_print (x04cac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Find the generalized Schur form using nag_dggesx (f08xbc). */
nag_dggesx(order, jobvsl, jobvsr, sort, selctg, sense, n, a, pda, b, pdb,
    &sdim, alphas, alphas, beta, vsl, pdvsl, vsr, pdvsr, rconde,
    rcondv, &fail);

if (fail.code != NE_NOERROR && fail.code != NE_SCHUR_REORDER_SELECT) {
    printf("Error from nag_dggesx (f08xbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Check generalized Schur Form by reconstruction of Schur vectors are
 * available.
 */
if (jobvsl == Nag_NotLeftVecs || jobvsr == Nag_NotRightVecs) {
    /* Cannot check factorization by reconstruction Schur vectors. */
    goto END;
}

/* Reconstruct A as Q*S*Z^T and subtract from original (D) using the steps
 * C = Q*S (Q in vsl, S in a) using nag_dgemm (f16yac).
 * Note: not nag_dtrmm since S may not be strictly triangular.
 * D = D - C*Z^T (Z in vsr) using nag_dgemm (f16yac).
 */
dg_a = 1.0;
dg_b = 0.0;
nag_dgemm(order, Nag_NoTrans, Nag_NoTrans, n, n, n, dg_a, vsl, pdvsl, a,
    pda, dg_b, c, pdc, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
dg_a = -1.0;
dg_b = 1.0;
nag_dgemm(order, Nag_NoTrans, Nag_Trans, n, n, n, dg_a, c, pdc, vsr, pdvsr,
    dg_b, d, pdd, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
    exit_status = 1;
}

```

```

    goto END;
}

/* Reconstruct B as Q*T*Z^T and subtract from original (E) using the steps
 * C = Q*T (Q in vs1, T in b) using nag_dgemm (f16yac).
 * E = E - C*Z^T (Z in vsr) using nag_dgemm (f16yac).
 */
dg_a = 1.0;
dg_b = 0.0;
nag_dgemm(order, Nag_NoTrans, Nag_NoTrans, n, n, n, dg_a, vs1, pdvsl, b,
          pdb, dg_b, c, pdc, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
dg_a = -1.0;
dg_b = 1.0;
nag_dgemm(order, Nag_NoTrans, Nag_Trans, n, n, n, dg_a, c, pdc, vsr, pdvsr,
          dg_b, e, pde, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* nag_dge_norm (f16rac): Find norms of difference matrices D and E. */
nag_dge_norm(order, Nag_FrobeniusNorm, n, n, d, pdd, &normd, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
nag_dge_norm(order, Nag_FrobeniusNorm, n, n, e, pde, &norme, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Get the machine precision, using nag_machine_precision (x02ajc) */
eps = nag_machine_precision;
if (MAX(normd, norme) > pow(eps, 0.8) * MAX(norma, normb)) {
    printf("The norm of the error in the reconstructed matrices is greater "
          "than expected.\nThe Schur factorization has failed.\n");
    exit_status = 1;
    goto END;
}

/* Print details on eigenvalues */
printf("Number of sorted eigenvalues = %4" NAG_IFMT "\n\n", sdim);
if (fail.code == NE_SCHUR_REORDER_SELECT) {
    printf("*** Note that rounding errors mean that leading eigenvalues in the"
          " generalized\n    Schur form no longer satisfy selctg = Nag_TRUE"
          "\n\n");
}
else {
    printf("The selected eigenvalues are:\n");
    for (i = 0; i < sdim; i++) {
        if (beta[i] != 0.0)
            printf("%3" NAG_IFMT " (%13.4e, %13.4e)\n",
                  i + 1, alphas[i] / beta[i], alphas[i] / beta[i]);
        else
            printf("%3" NAG_IFMT " Eigenvalue is infinite\n", i + 1);
    }
}

abnorm = sqrt(pow(norma, 2) + pow(normb, 2));
tol = eps * abnorm;

if (sense == Nag_RCondEigVals || sense == Nag_RCondBoth) {

```

```

    /* Print out the reciprocal condition number and error bound */
    printf("\n");
    printf("For the selected eigenvalues,\nthe reciprocals of projection "
           "norms onto the deflating subspaces are\n");
    printf(" for left  subspace, rcond = %10.1e\n for right subspace, rcond = "
           "%10.1e\n\n", rconde[0], rconde[1]);
    printf(" asymptotic error bound    = %10.1e\n\n", tol / rconde[0]);
}
if (sense == Nag_RCondEigVecs || sense == Nag_RCondBoth) {
    /* Print out the reciprocal condition numbers and error bound. */
    printf("For the left and right deflating subspaces,\n");
    printf("reciprocal condition numbers are:\n");
    printf(" for left  subspace, rcond = %10.1e\n for right subspace, rcond = "
           "%10.1e\n\n", rcondv[0], rcondv[1]);
    printf(" approximate error bound    = %10.1e\n", tol / rcondv[1]);
}
}

END:
NAG_FREE(a);
NAG_FREE(b);
NAG_FREE(c);
NAG_FREE(d);
NAG_FREE(e);
NAG_FREE(alphai);
NAG_FREE(alphar);
NAG_FREE(beta);
NAG_FREE(vsl);
NAG_FREE(vsr);

return exit_status;
}

static Nag_Boolean NAG_CALL selctg(const double ar, const double ai,
                                   const double b)
{
    /* Boolean function selctg for use with nag_dggesx (f08xbc)
     * Returns the value Nag_TRUE if the eigenvalue is real and positive.
     */

    return (ar > 0.0 && ai == 0.0 && b != 0.0 ? Nag_TRUE : Nag_FALSE);
}

```

## 10.2 Program Data

nag\_dggesx (f08xbc) Example Program Data

```

4                               : n

Nag_LeftVecs                    : jobvsl
Nag_RightVecs                   : jobvsr
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\_dggesx (f08xbc) Example Program Results

```

Matrix A
      1      2      3      4
1     3.9000 12.5000 -34.5000 -0.5000
2     4.3000 21.5000 -47.5000  7.5000

```

3	4.3000	21.5000	-43.5000	3.5000
4	4.4000	26.0000	-46.0000	6.0000

Matrix B

	1	2	3	4
1	1.0000	2.0000	-3.0000	1.0000
2	1.0000	3.0000	-5.0000	4.0000
3	1.0000	3.0000	-4.0000	3.0000
4	1.0000	3.0000	-4.0000	4.0000

Number of sorted eigenvalues = 2

The selected eigenvalues are:

1	(	2.0000e+00,	0.0000e+00)
2	(	4.0000e+00,	0.0000e+00)

For the selected eigenvalues,  
the reciprocals of projection norms onto the deflating subspaces are  
for left subspace, rcond = 1.9e-01  
for right subspace, rcond = 1.8e-02

asymptotic error bound = 5.7e-14

For the left and right deflating subspaces,  
reciprocal condition numbers are:

for left subspace, rcond =	5.4e-02
for right subspace, rcond =	9.0e-02

approximate error bound = 1.2e-13

---