NAG Library Function Document

nag_zhgeqz (f08xsc)

1 Purpose

nag_zhgeqz (f08xsc) implements the QZ method for finding generalized eigenvalues of the complex matrix pair \((A, B)\) of order \(n\), which is in the generalized upper Hessenberg form.

2 Specification

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

void nag_zhgeqz (Nag_OrderType order, Nag_JobType job,
                  Nag_ComputeQType compq, Nag_ComputeZType compz, Integer n, Integer ilo,
                  Integer ihi, Complex a[], Integer pda, Complex b[], Integer pdb,
                  Complex alpha[], Complex beta[], Integer pdq, Complex z[],
                  Integer pdz, NagError *fail)
```

3 Description

nag_zhgeqz (f08xsc) implements a single-shift version of the QZ method for finding the generalized eigenvalues of the complex matrix pair \((A, B)\) which is in the generalized upper Hessenberg form. If the matrix pair \((A, B)\) is not in the generalized upper Hessenberg form, then the function nag_zgghrd (f08wsc) should be called before invoking nag_zhgeqz (f08xsc).

This problem is mathematically equivalent to solving the matrix equation

\[ \det(A - \lambda B) = 0. \]

Note that, to avoid underflow, overflow and other arithmetic problems, the generalized eigenvalues \(\lambda_j\) are never computed explicitly by this function but defined as ratios between two computed values, \(\alpha_j\) and \(\beta_j\):

\[ \lambda_j = \alpha_j / \beta_j. \]

The arguments \(\alpha_j\), in general, are finite complex values and \(\beta_j\) are finite real non-negative values.

If desired, the matrix pair \((A, B)\) may be reduced to generalized Schur form. That is, the transformed matrices \(A\) and \(B\) are upper triangular and the diagonal values of \(A\) and \(B\) provide \(\alpha\) and \(\beta\).

The argument \(\text{job}\) specifies two options. If \(\text{job} = \text{Nag\_Schur}\) then the matrix pair \((A, B)\) is simultaneously reduced to Schur form by applying one unitary transformation (usually called \(Q\)) on the left and another (usually called \(Z\)) on the right. That is,

\[ A \leftarrow Q^H A Z \]
\[ B \leftarrow Q^H B Z \]

If \(\text{job} = \text{Nag\_EigVals}\), then at each iteration the same transformations are computed but they are only applied to those parts of \(A\) and \(B\) which are needed to compute \(\alpha\) and \(\beta\). This option could be used if generalized eigenvectors are required but not generalized eigenvectors.

If \(\text{job} = \text{Nag\_Schur}\) and \(\text{compq} = \text{Nag\_AccumulateQ}\) or \(\text{Nag\_InitQ}\), and \(\text{compz} = \text{Nag\_AccumulateZ}\) or \(\text{Nag\_InitZ}\), then the unitary transformations used to reduce the pair \((A, B)\) are accumulated into the input arrays \(q\) and \(z\). If generalized eigenvectors are required then \(\text{job}\) must be set to \(\text{job} = \text{Nag\_Schur}\) and if left (right) generalized eigenvectors are to be computed then \(\text{compq} (\text{compz})\) must be set to \(\text{compq} = \text{Nag\_AccumulateQ}\) or \(\text{Nag\_InitQ}\) rather than \(\text{compq} = \text{Nag\_NotQ}\).

If \(\text{compq} = \text{Nag\_InitQ}\), then eigenvectors are accumulated on the identity matrix and on exit the array \(q\) contains the left eigenvector matrix \(Q\). However, if \(\text{compq} = \text{Nag\_AccumulateQ}\) then the
transformations are accumulated in the user-supplied matrix $Q_0$ in array $q$ on entry and thus on exit $q$ contains the matrix product $QQ_0$. A similar convention is used for $compz$.

4 References


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:  
job – Nag_JobType

Input

On entry: specifies the operations to be performed on $(A, B)$.

job = Nag_EigVals
   The matrix pair $(A, B)$ on exit might not be in the generalized Schur form.

job = Nag_Schur
   The matrix pair $(A, B)$ on exit will be in the generalized Schur form.

Constraint: job = Nag_EigVals or Nag_Schur.

3:  
compq – Nag_ComputeQType

Input

On entry: specifies the operations to be performed on $Q$:

compq = Nag_NotQ
   The array $q$ is unchanged.

compq = Nag_AccumulateQ
   The left transformation $Q$ is accumulated on the array $q$.

compq = Nag_InitQ
   The array $q$ is initialized to the identity matrix before the left transformation $Q$ is accumulated in $q$.

Constraint: compq = Nag_NotQ, Nag_AccumulateQ or Nag_InitQ.

4:  
compz – Nag_ComputeZType

Input

On entry: specifies the operations to be performed on $Z$.

compz = Nag_NotZ
   The array $z$ is unchanged.

compz = Nag_AccumulateZ
   The right transformation $Z$ is accumulated on the array $z$. 
\textbf{compz} = \texttt{Nag_InitZ}

The array \textit{z} is initialized to the identity matrix before the right transformation \(Z\) is accumulated in \textit{z}.

\textit{Constraint: compz} = \texttt{Nag_NotZ}, \texttt{Nag_AccumulateZ} or \texttt{Nag_InitZ}.

5: \quad \textbf{n} – Integer \quad \textit{Input}

\textit{On entry:} \(n\), the order of the matrices \(A, B, Q\) and \(Z\).

\textit{Constraint:} \(n \geq 0\).

6: \quad \textbf{ilo} – Integer \quad \textit{Input}

7: \quad \textbf{ihi} – Integer \quad \textit{Input}

\textit{On entry:} the indices \(i_{lo}\) and \(i_{hi}\), respectively which define the upper triangular parts of \(A\). The submatrices \(A(1 : i_{lo} - 1, 1 : i_{lo} - 1)\) and \(A(i_{hi} + 1 : n, i_{hi} + 1 : n)\) are then upper triangular. These arguments are provided by nag_zggbal (f08wvc) if the matrix pair was previously balanced; otherwise, \(ilo = 1\) and \(ihi = n\).

\textit{Constraints:}
\begin{align*}
    \text{if } n > 0, & \quad 1 \leq ilo \leq ihi \leq n; \\
    \text{if } n = 0, & \quad ilo = 1 \text{ and } ihi = 0.
\end{align*}

8: \quad \textbf{a[\text{dim}]} – Complex \quad \textit{Input/Output}

\textit{Note:} the dimension, \texttt{dim}, of the array \textit{a} must be at least \(\max(1, \text{pda} \times n)\).

The \((i,j)\)th element of the matrix \(A\) is stored in
\begin{align*}
    a[(j - 1) \times \text{pda} + i - 1] & \text{ when order} = \text{Nag_ColMajor}; \\
    a[(i - 1) \times \text{pda} + j - 1] & \text{ when order} = \text{Nag_RowMajor}.
\end{align*}

\textit{On entry:} the \(n\) by \(n\) upper Hessenberg matrix \(A\). The elements below the first subdiagonal must be set to zero.

\textit{On exit:} if \textit{job} = \texttt{Nag_Schur}, the matrix pair \((A, B)\) will be simultaneously reduced to generalized Schur form.

If \textit{job} = \texttt{Nag_EigVals}, the 1 by 1 and 2 by 2 diagonal blocks of the matrix pair \((A, B)\) will give generalized eigenvalues but the remaining elements will be irrelevant.

9: \quad \textbf{pda} – Integer \quad \textit{Input}

\textit{On entry:} the stride separating row or column elements (depending on the value of \texttt{order}) in the array \textit{a}.

\textit{Constraint:} \texttt{pda} \geq \max(1, n).

10: \quad \textbf{b[\text{dim}]} – Complex \quad \textit{Input/Output}

\textit{Note:} the dimension, \texttt{dim}, of the array \textit{b} must be at least \(\max(1, \text{pdb} \times n)\).

The \((i,j)\)th element of the matrix \(B\) is stored in
\begin{align*}
    b[(j - 1) \times \text{pdb} + i - 1] & \text{ when order} = \text{Nag_ColMajor}; \\
    b[(i - 1) \times \text{pdb} + j - 1] & \text{ when order} = \text{Nag_RowMajor}.
\end{align*}

\textit{On entry:} the \(n\) by \(n\) upper triangular matrix \(B\). The elements below the diagonal must be zero.

\textit{On exit:} if \textit{job} = \texttt{Nag_Schur}, the matrix pair \((A, B)\) will be simultaneously reduced to generalized Schur form.

If \textit{job} = \texttt{Nag_EigVals}, the 1 by 1 and 2 by 2 diagonal blocks of the matrix pair \((A, B)\) will give generalized eigenvalues but the remaining elements will be irrelevant.
On entry: the stride separating row or column elements (depending on the value of order) in the array b.

Constraint: \( \text{pdb} \geq \max(1, n) \).

12: \textbf{alpha}[n] – Complex

On exit: \( \alpha_j \), for \( j = 1, 2, \ldots, n \).

13: \textbf{beta}[n] – Complex

On exit: \( \beta_j \), for \( j = 1, 2, \ldots, n \).

14: \textbf{q}[\text{dim}] – Complex

Note: the dimension, \( \text{dim} \), of the array \( \textbf{q} \) must be at least
\[ \max(1, \text{pdq} \times n) \] when \( \text{compq} = \text{Nag\_AccumulateQ} \) or \( \text{Nag\_InitQ} \);
1 when \( \text{compq} = \text{Nag\_NotQ} \).

The \((i, j)\)th element of the matrix \( Q \) is stored in
\[ q[(j - 1) \times \text{pdq} + i - 1] \] when \( \text{order} = \text{Nag\_ColMajor} \);
\[ q[(i - 1) \times \text{pdq} + j - 1] \] when \( \text{order} = \text{Nag\_RowMajor} \).

On entry: if \( \text{compq} = \text{Nag\_AccumulateQ} \), the matrix \( Q_0 \). The matrix \( Q_0 \) is usually the matrix \( Q \) returned by \textit{nag\_zgghrd} (f08wsc).

If \( \text{compq} = \text{Nag\_NotQ} \), \( \textbf{q} \) is not referenced.

On exit: if \( \text{compq} = \text{Nag\_AccumulateQ} \), \( \textbf{q} \) contains the matrix product \( QQ_0 \).

If \( \text{compq} = \text{Nag\_InitQ} \), \( \textbf{q} \) contains the transformation matrix \( Q \).

15: \textbf{pdq} – Integer

On entry: the stride separating row or column elements (depending on the value of order) in the array \( \textbf{q} \).

Constraints:

if \( \text{order} = \text{Nag\_ColMajor} \),
\[ \begin{align*}
    & \text{if } \text{compq} = \text{Nag\_AccumulateQ} \text{ or } \text{Nag\_InitQ}, \text{pdq} \geq n; \\
    & \text{if } \text{compq} = \text{Nag\_NotQ}, \text{pdq} \geq 1; \\
    & \text{if } \text{order} = \text{Nag\_RowMajor} \text{,} \\
    & \text{if } \text{compq} = \text{Nag\_AccumulateQ} \text{ or } \text{Nag\_InitQ}, \text{pdq} \geq \max(1, n); \\
    & \text{if } \text{compq} = \text{Nag\_NotQ}, \text{pdq} \geq 1. \\
\end{align*} \]

16: \textbf{z}[\text{dim}] – Complex

Note: the dimension, \( \text{dim} \), of the array \( \textbf{z} \) must be at least
\[ \max(1, \text{pdz} \times n) \] when \( \text{compz} = \text{Nag\_AccumulateZ} \) or \( \text{Nag\_InitZ} \);
1 when \( \text{compz} = \text{Nag\_NotZ} \).

The \((i, j)\)th element of the matrix \( Z \) is stored in
\[ z[(j - 1) \times \text{pdz} + i - 1] \] when \( \text{order} = \text{Nag\_ColMajor} \);
\[ z[(i - 1) \times \text{pdz} + j - 1] \] when \( \text{order} = \text{Nag\_RowMajor} \).

On entry: if \( \text{compz} = \text{Nag\_AccumulateZ} \), the matrix \( Z_0 \). The matrix \( Z_0 \) is usually the matrix \( Z \) returned by \textit{nag\_zgghrd} (f08wsc).

If \( \text{compz} = \text{Nag\_NotZ} \), \( \textbf{z} \) is not referenced.

On exit: if \( \text{compz} = \text{Nag\_AccumulateZ} \), \( \textbf{z} \) contains the matrix product \( ZZ_0 \).
If \texttt{compz} = \texttt{Nag_InitZ}, \texttt{z} contains the transformation matrix \texttt{Z}.

17: \texttt{pdz} – Integer \hspace{1cm} \textit{Input}

\textit{On entry:} the stride separating row or column elements (depending on the value of \texttt{order}) in the array \texttt{z}.

\textit{Constraints:}

if \texttt{order} = \texttt{Nag_ColMajor},
  \begin{align*}
  \text{if} \ \texttt{compz} = \texttt{Nag_AccumulateZ} \text{ or } \texttt{Nag_InitZ}, \ \texttt{pdz} & \geq \texttt{n}; \\
  \text{if} \ \texttt{compz} = \texttt{Nag_NotZ}, \ \texttt{pdz} & \geq 1.; \\
  \text{if} \ \texttt{order} = \texttt{Nag_RowMajor}, \\
  \text{if} \ \texttt{compz} = \texttt{Nag_AccumulateZ} \text{ or } \texttt{Nag_InitZ}, \ \texttt{pdz} & \geq \max(1, \texttt{n}); \\
  \text{if} \ \texttt{compz} = \texttt{Nag_NotZ}, \ \texttt{pdz} & \geq 1.;
  \end{align*}

18: \texttt{fail} – NagError * \hspace{1cm} \textit{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 \texttt{value} \rangle had an illegal value.

NE_ENUM_INT_2

On entry, \texttt{compq} = \langle \texttt{value} \rangle, \texttt{pdq} = \langle \texttt{value} \rangle, \texttt{n} = \langle \texttt{value} \rangle.

Constraint: if \texttt{compq} = \texttt{Nag_AccumulateQ} \text{ or } \texttt{Nag_InitQ}, \ \texttt{pdq} \geq \max(1, \texttt{n});

\texttt{if} \ \texttt{compq} = \texttt{Nag_NotQ}, \ \texttt{pdq} \geq 1.

On entry, \texttt{compq} = \langle \texttt{value} \rangle, \texttt{pdq} = \langle \texttt{value} \rangle and \texttt{n} = \langle \texttt{value} \rangle.

Constraint: if \texttt{compq} = \texttt{Nag_AccumulateQ} \text{ or } \texttt{Nag_InitQ}, \ \texttt{pdq} \geq \texttt{n};

\texttt{if} \ \texttt{compq} = \texttt{Nag_NotQ}, \ \texttt{pdq} \geq 1.

On entry, \texttt{compz} = \langle \texttt{value} \rangle, \texttt{pdz} = \langle \texttt{value} \rangle, \texttt{n} = \langle \texttt{value} \rangle.

Constraint: if \texttt{compz} = \texttt{Nag_AccumulateZ} \text{ or } \texttt{Nag_InitZ}, \ \texttt{pdz} \geq \max(1, \texttt{n});

\texttt{if} \ \texttt{compz} = \texttt{Nag_NotZ}, \ \texttt{pdz} \geq 1.

On entry, \texttt{compz} = \langle \texttt{value} \rangle, \texttt{pdz} = \langle \texttt{value} \rangle and \texttt{n} = \langle \texttt{value} \rangle.

Constraint: if \texttt{compz} = \texttt{Nag_AccumulateZ} \text{ or } \texttt{Nag_InitZ}, \ \texttt{pdz} \geq \texttt{n};

\texttt{if} \ \texttt{compz} = \texttt{Nag_NotZ}, \ \texttt{pdz} \geq 1.

NE_INT

On entry, \texttt{n} = \langle \texttt{value} \rangle.

Constraint: \texttt{n} \geq 0.

On entry, \texttt{pda} = \langle \texttt{value} \rangle.

Constraint: \texttt{pda} > 0.

On entry, \texttt{pdb} = \langle \texttt{value} \rangle.

Constraint: \texttt{pdb} > 0.

On entry, \texttt{pdq} = \langle \texttt{value} \rangle.

Constraint: \texttt{pdq} > 0.
On entry, \( \text{pdz} = \langle \text{value} \rangle \).
Constraint: \( \text{pdz} > 0 \).

**NE_INT_2**

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

**NE_INT_3**

On entry, \( \text{n} = \langle \text{value} \rangle \), \( \text{ilo} = \langle \text{value} \rangle \) and \( \text{ihi} = \langle \text{value} \rangle \).
Constraint: if \( \text{n} > 0 \), \( 1 \leq \text{ilo} \leq \text{ihi} \leq \text{n} \);
if \( \text{n} = 0 \), \( \text{ilo} = 1 \) and \( \text{ihi} = 0 \).

**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.
An unexpected Library error has occurred.

**NE_ITERATION_QZ**

The \( QZ \) iteration did not converge and the matrix pair \( (A, B) \) is not in the generalized Schur form. The computed \( \alpha_i \) and \( \beta_i \) should be correct for \( i = \langle \text{value} \rangle, \ldots, \langle \text{value} \rangle \).

**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.

**NE_SCHUR**

The computation of shifts failed and the matrix pair \( (A, B) \) is not in the generalized Schur form.
The computed \( \alpha_i \) and \( \beta_i \) should be correct for \( i = \langle \text{value} \rangle, \ldots, \langle \text{value} \rangle \).

7  **Accuracy**

Please consult Section 4.11 of the LAPACK Users’ Guide (see Anderson et al. (1999)) and Chapter 6 of Stewart and Sun (1990), for more information.

8  **Parallelism and Performance**

\text{nag\_zhgeqz} (f08xsc) is not threaded by NAG in any implementation.
\text{nag\_zhgeqz} (f08xsc) 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**

\text{nag\_zhgeqz} (f08xsc) is the fifth step in the solution of the complex generalized eigenvalue problem and is called after \text{nag\_zgghrd} (f08wsc).
The number of floating-point operations taken by this function is proportional to $n^3$.

The real analogue of this function is nag_dhgeqz (f08xec).

## 10 Example

This example computes the $\alpha$ and $\beta$ arguments, which defines the generalized eigenvalues, 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

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

This requires calls to 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 using the $QZ$ algorithm.

### 10.1 Program Text

```c
/* nag_zhgeqz (f08xsc) Example Program. */
/* Copyright 2014 Numerical Algorithms Group. */
/* Mark 7, 2001. */

#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <naga02.h>
#include <nagf08.h>
#include <nagx04.h>

int main(void)
{
    /* Scalars */
    Integer i, ihi, ilo, irows, j, n, pda, pdb;
    Integer alpha_len, beta_len, scale_len, tau_len;
    Integer exit_status = 0;
    NagError fail;
    Nag_OrderType order;
    /* Arrays */
    Complex *a = 0, *alpha = 0, *b = 0, *beta = 0, *q = 0, *tau = 0;
    Complex *z = 0;
    Complex e;
    double *lscale = 0, *rscale = 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 ]
    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_zheqz (f08xsc) Example Program Results\n\n");
/* Skip heading in data file */
#endif _WIN32
scanf_s("%*[`\n ] ");
#else
scanf("%*[`\n ] ");
#endif _WIN32
scanf_s("%"NAG_IFMT"%*[`\n ] ", &n);
#else
scanf("%"NAG_IFMT"%*[`\n ] ", &n);
#endif NAG_COLUMN_MAJOR
pda = n;
pdb = n;
#else
pda = n;
pdb = n;
#endif
alpha_len = n;
beta_len = n;
scale_len = n;
tau_len = n;

/* Allocate memory */
if (! (a = NAG_ALLOC(n * n, Complex)) ||
    ! (alpha = NAG_ALLOC(alpha_len, Complex)) ||
    ! (b = NAG_ALLOC(n * n, Complex)) ||
    ! (beta = NAG_ALLOC(beta_len, Complex)) ||
    ! (q = NAG_ALLOC(1 * 1, Complex)) ||
    ! (tau = NAG_ALLOC(tau_len, Complex)) ||
    ! (lscale = NAG_ALLOC(scale_len, double)) ||
    ! (rscale = NAG_ALLOC(scale_len, double)) ||
    ! (z = NAG_ALLOC(1 * 1, Complex)))
{
    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)
    {
        if _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 _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)
    {
        if _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 _WIN32
scanf_s("%*[`\n ] ");
#else

```c
/* Balance matrix pair (A,B) */
*nag_zggbal (f08wvc).
* Balance a pair of complex general matrices */
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;
}

/* Matrix A after balancing */
/* nag_gen_complx_mat_print_comp (x04dbc).
* Print complex general matrix (comprehensive) */
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)
{
    printf("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
printf("\n");

/* Matrix B after balancing */
/* nag_gen_complx_mat_print_comp (x04dbc), see above. */
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 */
irows = ihi + 1 - ilo;
/* nag_zgeqrf (f08asc).
* QR factorization of complex general rectangular matrix */
nag_zgeqrf(order, irows, irows, &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 to matrix A */
/* nag_zunmqr (f08auc).
* Apply unitary transformation determined by nag_zgeqrf */
```

Mark 25

f08 – Least-squares and Eigenvalue Problems (LAPACK)
nag_zunmqr(order, Nag_LeftSide, Nag_ConjTrans, irows, irows, 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;
}

/* Compute the generalized Hessenberg form of (A,B) */
/* nag_zgghrd (f08wsc). */
* Unitary reduction of a pair of complex general matrices
* to generalized upper Hessenberg form *

nag_zgghrd(order, Nag_NotQ, Nag_NotZ, irows, 1, irows, &A(ilo, ilo),
pda, &B(ilo, ilo), pdb, q, 1, z, 1, &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_zgghrd (f08wsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

/* Matrix A in generalized Hessenberg form */
/* nag_gen_complx_mat_print_comp (x04dbc), see above. */
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("Error from nag_gen_complx_mat_print_comp (x04dbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}

printf("\n");
/* Matrix B in generalized Hessenberg form */
/* nag_gen_complx_mat_print_comp (x04dbc), see above. */
fflush(stdout);
nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n,
n, b, pdb, Nag_BracketForm, "%7.3f", "Matrix B is triangular",
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_EigVals, Nag_NotQ, Nag_NotZ, n, ilo, ihi, a,
pda, b, pdb, alpha, beta, q, 1, z, 1, &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_zhgeqz (f08xsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
/* Print the generalized eigenvalues */
printf("\n Generalized eigenvalues\n");
for (i = 0; i < n; ++i)
{
    if (beta[i].re != 0.0 || beta[i].im != 0.0)
    { /* nag_complex_divide (a02cdc).
         Quotient of two complex numbers */
        e = nag_complex_divide(alpha[i], beta[i]);
        printf(" %4"NAG_IFMT" (%7.3f,%7.3f)\n", i+1, e.re, e.im);
    } else
        printf(" %4"NAG_IFMT" Infinite eigenvalue\n", i+1);
}
END:
NAG_FREE(a);
NAG_FREE(alpha);
NAG_FREE(b);
NAG_FREE(beta);
NAG_FREE(lscale);
NAG_FREE(q);
NAG_FREE(rscale);
NAG_FREE(tau);
NAG_FREE(z);
return exit_status;
}

10.2 Program Data

nag_zhgeqz (f08xsc) Example Program Data

<table>
<thead>
<tr>
<th></th>
<th>Value of N</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.00, 3.00</td>
</tr>
<tr>
<td></td>
<td>2.00, 2.00</td>
</tr>
<tr>
<td></td>
<td>3.00, 1.00</td>
</tr>
<tr>
<td></td>
<td>4.00, 0.00</td>
</tr>
</tbody>
</table>

: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_zhgeqz (f08xsc) Example Program Results

Matrix A after balancing

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1.0000, 3.0000)</td>
<td>(1.0000, 4.0000)</td>
<td>(0.1000, 0.5000)</td>
<td>(0.1000, 0.6000)</td>
</tr>
<tr>
<td>2</td>
<td>(2.0000, 2.0000)</td>
<td>(4.0000, 3.0000)</td>
<td>(0.8000, 0.4000)</td>
<td>(1.6000, 0.5000)</td>
</tr>
<tr>
<td>3</td>
<td>(0.3000, 0.1000)</td>
<td>(0.9000, 0.2000)</td>
<td>(0.2700, 0.0300)</td>
<td>(0.8100, 0.0400)</td>
</tr>
<tr>
<td>4</td>
<td>(0.4000, 0.0000)</td>
<td>(1.6000, 0.1000)</td>
<td>(0.6400, 0.0200)</td>
<td>(2.5600, 0.0300)</td>
</tr>
</tbody>
</table>

Matrix B after balancing

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1.0000, 0.0000)</td>
<td>(2.0000, 1.0000)</td>
<td>(0.3000, 0.2000)</td>
<td>(0.4000, 0.3000)</td>
</tr>
<tr>
<td>2</td>
<td>(1.0000, 1.0000)</td>
<td>(4.0000, 2.0000)</td>
<td>(0.9000, 0.3000)</td>
<td>(1.6000, 0.4000)</td>
</tr>
<tr>
<td>3</td>
<td>(0.1000, 0.2000)</td>
<td>(0.8000, 0.3000)</td>
<td>(0.2700, 0.0400)</td>
<td>(0.6400, 0.0500)</td>
</tr>
<tr>
<td>4</td>
<td>(0.1000, 0.3000)</td>
<td>(1.6000, 0.4000)</td>
<td>(0.8100, 0.0500)</td>
<td>(2.5600, 0.0600)</td>
</tr>
</tbody>
</table>

Matrix A in Hessenberg form

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(-2.868, -1.595)</td>
<td>(-0.809, -0.328)</td>
<td>(-4.900, -0.987)</td>
<td>(-0.048, 1.163)</td>
</tr>
<tr>
<td>2</td>
<td>(-2.672, 0.595)</td>
<td>(-0.790, 0.049)</td>
<td>(-4.955, -0.163)</td>
<td>(-0.439, -0.574)</td>
</tr>
<tr>
<td>3</td>
<td>(0.000, 0.000)</td>
<td>(-0.098, -0.011)</td>
<td>(-1.168, -0.137)</td>
<td>(-1.756, -0.205)</td>
</tr>
<tr>
<td>4</td>
<td>(0.000, 0.000)</td>
<td>(0.000, 0.000)</td>
<td>(0.087, 0.004)</td>
<td>(0.032, 0.001)</td>
</tr>
</tbody>
</table>

Matrix B is triangular
Generalized eigenvalues

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