Module 6.5: nag_sym_gen_eig
Symmetric-definite Generalized Eigenvalue Problems

`nag_sym_gen_eig` provides procedures for solving generalized eigenvalue problems of the forms:

$$Az = \lambda Bz$$

$$ABz = \lambda z$$

$$BAz = \lambda z$$

where A and B are *real symmetric* or *complex Hermitian* and B is *positive definite*.

Contents

Introduction

1 Notation

The most common form of *symmetric-definite generalized eigenvalue problem* is to find the *eigenvalues* λ_i, and corresponding eigenvectors z_i, satisfying

$$Az_i = \lambda_i B z_i,$$

where A and B are real symmetric matrices and B is positive definite. A *Hermitian-definite* problem is defined likewise for complex Hermitian matrices. For both problems the eigenvalues λ_i are real.

Symmetric-definite (or Hermitian-definite) problems may also be posed in one of the alternative forms

$$ABz_i = \lambda_i z_i \quad \text{or} \quad BAz_i = \lambda_i z_i.$$

Each of these problems can be reduced to a standard symmetric or Hermitian eigenvalue problem, using a Cholesky factorization of B as either $U^H U$ or LL^H (if B is real $U^H = U^T$ and $L^H = L^T$).

With $B = LL^H$, we have

$$Az = \lambda Bz \quad \Rightarrow \quad (L^{-1}AL^{-H})(L^H z) = \lambda(L^H z).$$

Hence the eigenvalues of the generalized problem $Az = \lambda Bz$ are those of the standard problem $Cy = \lambda y$, where $C = L^{-1}AL^{-H}$ and $y = L^H z$.

The reduced problem $Cy = \lambda y$ can be solved by the methods described in the module `nag_sym_eig` (6.1), and the eigenvalues z of the generalized problem recovered from the eigenvectors y of the reduced problem by $z = L^{-H}y$. Note however that the reduction implicitly involves the inversion of B, and hence may lead to unreliable results if B is ill conditioned with respect to inversion.

The table below shows how each of the three types of problem can be reduced to standard form:

	Problem	Factorization of B	Reduction	Recovery of eigenvectors
1.	$Az = \lambda Bz$	$B = U^H U$ $B = LL^H$	$C = U^{-H}AU^{-1}$ $C = L^{-1}AL^{-H}$	$z = U^{-1}y$ $z = L^{-H}y$
2.	$ABz = \lambda z$	$B = U^H U$ $B = LL^H$	$C = UAU^H$ $C = L^H AL$	$z = U^{-1}y$ $z = L^{-H}y$
3.	$BAz = \lambda z$	$B = U^H U$ $B = LL^H$	$C = UAU^H$ $C = L^H AL$	$z = U^H y$ $z = Ly$

2 Choice of Procedures

The procedures `nag_sym_gen_eig_all` and `nag_sym_gen_eig_sel` have been designed to meet most requirements. They solve the most frequent types of problems in a single call, namely:

All the eigenvalues (`nag_sym_gen_eig_all`)

All the eigenvalues and the eigenvectors (`nag_sym_gen_eig_all` with optional argument)

Selected eigenvalues (`nag_sym_gen_eig_sel`)

Selected eigenvalues and the corresponding eigenvectors (`nag_sym_gen_eig_sel` with optional argument)

3 Storage of Matrices

The procedures in this module allow a choice of storage schemes for the symmetric or Hermitian matrix A: conventional storage or packed storage. The choice is determined by the rank of the corresponding argument `a`.

3.1 Conventional Storage

`a` is a rank-2 array, of shape (n,n). Matrix element a_{ij} is stored in `a(`i,j`)`. Only the elements of either the upper or the lower triangle need be stored, as specified by the mandatory argument `uplo`; the remaining elements of `a` need not be set.

This storage scheme is more straightforward and carries less risk of user error than packed storage; on some machines it may result in more efficient execution. But it requires almost twice as much memory, although the other triangle of `a` can sometimes be used to store other data, and if the matrix Z of eigenvectors is required, it can also be stored in `a`, overwriting the matrix A.

3.2 Packed Storage

`a` is a rank-1 array of shape $(n(n+1)/2)$. The elements of either the upper or the lower triangle of A, as specified by `uplo`, are packed by columns into contiguous elements of `a`.

Packed storage is more economical in use of memory than conventional storage, but if all eigenvectors are required, a separate rank-2 array `z` must be supplied to store them. Packed storage may also result in less efficient execution on some machines.

The details of packed storage are as follows.

- If `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$i+j(j-1)/2$`)`, for $i \leq j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i+(2n-j)(j-1)/2$`)`, for $i \geq j$.

For example

`uplo`	Hermitian Matrix	Packed storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \overline{a}_{12} & a_{22} & a_{23} & a_{24} \\ \overline{a}_{13} & \overline{a}_{23} & a_{33} & a_{34} \\ \overline{a}_{14} & \overline{a}_{24} & \overline{a}_{34} & a_{44} \end{pmatrix}$	a_{11} $\underbrace{a_{12}\, a_{22}}$ $\underbrace{a_{13}\, a_{23}\, a_{33}}$ $\underbrace{a_{14}\, a_{24}\, a_{34}\, a_{44}}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & \overline{a}_{21} & \overline{a}_{31} & \overline{a}_{41} \\ a_{21} & a_{22} & \overline{a}_{32} & \overline{a}_{42} \\ a_{31} & a_{32} & a_{33} & \overline{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\, a_{21}\, a_{31}\, a_{41}}$ $\underbrace{a_{22}\, a_{32}\, a_{42}}$ $\underbrace{a_{33}\, a_{43}}$ a_{44}

Procedure: nag_sym_gen_eig_all

1 Description

nag_sym_gen_eig_all is a generic procedure which computes all the eigenvalues, and optionally all the eigenvectors, of a generalized real symmetric-definite or complex Hermitian-definite generalized eigenvalue problem.

By default, the problem has the form $Az = \lambda Bz$, where A and B are real symmetric or complex Hermitian and B is *positive definite*. The procedure allows either conventional or packed storage for A and B (see the Module Introduction).

The procedure can also handle problems of the alternative types $ABz = \lambda z$ or $BAz = \lambda z$, depending on the value of the optional argument type.

By default, only the eigenvalues are computed. If the optional argument z_on_a is present and set to .true., the eigenvectors are computed and overwritten on the original matrix A (this option is only available if conventional storage is used); otherwise if the optional argument z is present, the eigenvectors are computed and stored in z.

We write, for the different types of problem:

1. $Az_i = \lambda_i Bz_i$ for $i = 1, \ldots, n$,
2. $ABz_i = \lambda_i z_i$ for $i = 1, \ldots, n$,
3. $BAz_i = \lambda_i z_i$ for $i = 1, \ldots, n$,

where λ_i is an eigenvalue and z_i is the corresponding eigenvector.

We use Z to denote the matrix whose columns are the eigenvectors z_i. This matrix is *not* orthogonal or unitary (as it is for a standard eigenvalue problem), but satisfies:

$Z^H BZ = I$ for problems of type 1 or 2;
$Z^H B^{-1} Z = I$ for problems of type 3.

2 Usage

```
USE nag_sym_gen_eig
CALL nag_sym_gen_eig_all(uplo, a, b, lambda  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the four combinations of the following cases.

Real / complex data

Real data: a, b and the optional argument z are of type real(kind=*wp*).

Complex data: a, b and the optional argument z are of type complex(kind=*wp*).

Conventional / packed storage (see the Module Introduction)

Conventional: a and b are rank-2 arrays.

Packed: a and b are rank-1 arrays.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

n — the order of the matrices A and B

3.1 Mandatory Arguments

uplo — character(len=1), intent(in)

Input: specifies whether the upper or lower triangle of A and B is supplied.

If `uplo` = `'U'` or `'u'`, the upper triangle is supplied;

if `uplo` = `'L'` or `'l'`, the lower triangle is supplied.

Constraints: `uplo` = `'U'`, `'u'`, `'L'` or `'l'`.

a(n,n) **/ a**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix A.

Conventional storage (`a` has shape (n,n))

If `uplo` = `'u'`, the upper triangle of A must be stored, and elements below the diagonal need not be set;

if `uplo` = `'l'`, the lower triangle of A must be stored, and elements above the diagonal need not be set.

Packed storage (`a` has shape $(n(n+1)/2)$)

If `uplo` = `'u'`, the upper triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+j(j-1)/2$`)` for $i \leq j$;

if `uplo` = `'l'`, the lower triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+(2n-j)(j-1)/2$`)` for $i \geq j$.

Output: if `z_on_a` is present and set to `.true.`(conventional storage only), `a` is overwritten by the matrix Z of eigenvectors; otherwise (by default), `a` is overwritten by intermediate results.

b(n,n) **/ b**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix B.

Conventional storage (`b` has shape (n,n))

If `uplo` = `'u'`, the upper triangle of B must be stored, and elements below the diagonal need not be set;

if `uplo` = `'l'`, the lower triangle of B must be stored, and elements above the diagonal need not be set.

Packed storage (`b` has shape $(n(n+1)/2)$)

If `uplo` = `'u'`, the upper triangle of B must be stored, packed by columns, with b_{ij} in `b(`$i+j(j-1)/2$`)` for $i \leq j$;

if `uplo` = `'l'`, the lower triangle of B must be stored, packed by columns, with b_{ij} in `b(`$i+(2n-j)(j-1)/2$`)` for $i \geq j$.

Output: the upper or lower triangle of `b` is overwritten by the upper or lower triangular Cholesky factor of B, as specified by `uplo`.

Constraints: `b` must be of the same rank and type as `a`.

lambda(n) — real(kind=*wp*), intent(out)

Output: the eigenvalues in ascending order.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

type — integer, intent(in), optional

Input: specifies the type of problem:

If `type` = 1, the problem has the form $Az = \lambda Bz$;

if `type` = 2, the problem has the form $ABz = \lambda z$;

if `type` = 3, the problem has the form $BAz = \lambda z$.

Default: `type` = 1.

Constraints: `type` = 1, 2 or 3.

z_on_a — logical, intent(in), optional

Input: specifies whether the matrix Z of eigenvectors is to be overwritten on `a`.

If `z_on_a` = `.false.`, Z is not computed unless `z` is present;

if `z_on_a` = `.true.`, Z is overwritten on `a`.

Default: `z_on_a` = `.false.`.

Constraints: `z_on_a` must *not* be present if packed storage is used (`a` and `b` have rank 1).

z(n, n) — real(kind=*wp*) / complex(kind=*wp*), intent(out), optional

Output: the matrix Z of eigenvectors. The ith column `z(:,`i`)` holds the eigenvector corresponding to the eigenvalue `lambda(`i`)`.

Note: if `z_on_a` is present and set to `.true.`, and `z` is also present, then `z` is not used and a warning is raised.

Constraints: `z` must be of the same type as `a`.

rcond_b — real(kind=*wp*), intent(out), optional

Output: an estimate of the reciprocal of the condition number of B in the 1-norm. `rcond_b` is set to zero if exact singularity is detected or the estimate underflows. If `rcond_b` is less than `EPSILON(1.0_`*wp*`)`, then B is singular to working precision and the results may be completely unreliable.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Failure to converge. (This error is not likely to occur.) The QR algorithm failed to compute all the eigenvalues in the permitted number of iterations.
202	Matrix B not positive definite. Either B is close to singularity, or it has at least one negative eigenvalue. The problem could not be reduced to standard form (see Section 6.1).

Warnings (error%level = 1):

error%code	Description
101	Optional argument present but not used. `z` is present when `z_on_a` is `.true.`; the eigenvectors are returned in `a`, and `z` is not used.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure first calls `nag_sym_lin_fac` to perform a Cholesky factorization of B. It reduces the original generalized problem to an equivalent standard problem $Cy = \lambda y$ with the same eigenvalues, as described in the Module Introduction. It calls `nag_sym_eig_all` to compute all the eigenvalues and (if required) the eigenvectors of the problem $Cy = \lambda y$. Finally, it recovers the eigenvectors z of the original problem (if required) from the eigenvectors y of the reduced problem. See Chapter 8 of Golub and Van Loan [2] or Parlett [3] for background information.

The algorithms are derived from LAPACK (see Anderson *et al.* [1]).

6.2 Accuracy

Error bounds for the computed eigenvalues and eigenvectors involve the condition number of B; the computed eigenvalues and eigenvectors may be inaccurate if B is ill conditioned; that is, if the condition number $\kappa_2(B)$ is large, where $\kappa_2(B) = \|B\|_2\|B^{-1}\|_2$. An approximate estimate for the reciprocal of the condition number of B is returned in the optional argument `rcond_b`. (`rcond_b` returns an estimate for the reciprocal of the condition number in 1-norm, $\kappa_1(B)$; this differs by a factor of at most n from the condition number in the 2-norm, $\kappa_2(B)$, which appears in the error analysis.)

In more detail: let λ_i be an exact eigenvalue, and $\tilde{\lambda}_i$ be the corresponding computed value; let z_i be the corresponding exact eigenvector and $\tilde{z}_i$ the computed eigenvector, and let $\theta(\tilde{z}_i, z_i)$ denote the angle between them.

Then for problems of the form $Az = \lambda Bz$:

$$|\tilde{\lambda}_i - \lambda_i| \leq c(n)\epsilon(\|B^{-1}\|_2\|A\|_2 + \kappa_2(B)|\tilde{\lambda}_i|)$$

$$\theta(\tilde{z}_i, z_i) \leq c(n)\epsilon\left(\frac{\|B^{-1}\|_2\|A\|_2(\kappa_2(B))^{1/2} + \kappa_2(B)|\tilde{\lambda}_i|}{\mathrm{gap}_i}\right).$$

For problems of the form $ABz = \lambda z$ or $BAz - \lambda z$:

$$|\tilde{\lambda}_i - \lambda_i| \leq c(n)\epsilon(\|B\|_2\|A\|_2 + \kappa_2(B)|\tilde{\lambda}_i|)$$

$$\theta(\tilde{z}_i, z_i) \leq c(n)\epsilon \left(\frac{\|B\|_2 \|A\|_2 (\kappa_2(B))^{1/2}}{\text{gap}_i} + \kappa_2(B)|\tilde{\lambda}_i| \right).$$

Here $\epsilon =$ EPSILON(1.0_*wp*), $c(n)$ is a modestly increasing function of n, and $\text{gap}_i = \min_{i \neq j} |\lambda_i - \lambda_j|$.

6.3 Timing

The time taken by the procedure is approximately proportional to n^3. Computing both eigenvectors and eigenvalues is likely to take about 5 times as long as computing eigenvalues alone.

Procedure: nag_sym_gen_eig_sel

1 Description

nag_sym_gen_eig_sel is a generic procedure which computes selected eigenvalues, and optionally the corresponding eigenvectors, of a generalized real symmetric-definite or complex Hermitian-definite eigenvalue problem.

By default, the problem has the form $Az = \lambda Bz$, where A and B are real symmetric or complex Hermitian and B is *positive definite*. The procedure allows either conventional or packed storage for A and B (see the Module Introduction).

The procedure can also handle problems of the alternative types $ABz = \lambda z$ or $BAz = \lambda z$, depending on the value of the optional argument type.

We write, for the different types of problem:

1. $Az_i = \lambda_i Bz_i$ for $i = 1, \ldots, n$,
2. $ABz_i = \lambda_i z_i$ for $i = 1, \ldots, n$,
3. $BAz_i = \lambda_i z_i$ for $i = 1, \ldots, n$,

where λ_i is an eigenvalue and z_i is the corresponding eigenvector. The eigenvalues are arranged in ascending order:

$$\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n.$$

Eigenvalues may be selected either by index or by value (but not by a combination of the two). If either or both of the optional arguments il and iu are present, the procedure computes those eigenvalues λ_i whose indices i satisfy

$$\texttt{il} \leq i \leq \texttt{iu}.$$

If either or both of the optional arguments vl and vu are present, it computes those eigenvalues λ which satisfy

$$\texttt{vl} < \lambda \leq \texttt{vu}.$$

By default, only eigenvalues are computed. The eigenvectors corresponding to the selected eigenvalues are computed only if the optional argument z is present.

The number of selected eigenvalues is denoted by m. The argument lambda and the optional arguments z and fail are pointer arrays, because, if eigenvalues are selected by value, the number of them in the specified range may not be known in advance. If eigenvalues are selected by index, $m =$ iu $-$ il $+$ 1. The procedure allocates the required amount of memory to lambda, z and fail; on exit from the procedure, $m =$ SIZE(lambda).

Each eigenvector z_i satisfies $z_i^H Bz_i = 1$ for problems of types 1 and 2, and $z_i^H B^{-1} z_i = 1$ for problems of type 3.

2 Usage

```
USE nag_sym_gen_eig
CALL nag_sym_gen_eig_sel(uplo, a, b, lambda  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the four combinations of the following cases.

Real / complex data

Real data: a, b and the optional argument z are of type real(kind=*wp*).

Complex data: a, b and the optional argument z are of type complex(kind=*wp*).

Conventional / packed storage (see the Module Introduction)

Conventional: a and b are rank-2 arrays.

Packed: a and b are rank-1 arrays.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

n — the order of the matrices A and B

3.1 Mandatory Arguments

uplo — character(len=1), intent(in)

Input: specifies whether the upper or lower triangle of A and B is supplied.

If uplo = 'U' or 'u', the upper triangle is supplied;

if uplo = 'L' or 'l', the lower triangle is supplied.

Constraints: uplo = 'U', 'u', 'L' or 'l'.

a(n,n) **/ a**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix A.

Conventional storage (a has shape (n,n))

If uplo = 'u', the upper triangle of A must be stored, and elements below the diagonal need not be set;

if uplo = 'l', the lower triangle of A must be stored, and elements above the diagonal need not be set.

Packed storage (a has shape $(n(n+1)/2)$)

If uplo = 'u', the upper triangle of A must be stored, packed by columns, with a_{ij} in a($i+j(j-1)/2$) for $i \leq j$;

if uplo = 'l', the lower triangle of A must be stored, packed by columns, with a_{ij} in a($i+(2n-j)(j-1)/2$) for $i \geq j$.

Output: a is overwritten by intermediate results.

b(n,n) **/ b**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix B.

Conventional storage (b has shape (n,n))

If uplo = 'u', the upper triangle of B must be stored, and elements below the diagonal need not be set;

if uplo = 'l', the lower triangle of B must be stored, and elements above the diagonal need not be set.

Packed storage (b has shape $(n(n+1)/2)$)

If uplo = 'u', the upper triangle of B must be stored, packed by columns, with b_{ij} in b($i+j(j-1)/2$) for $i \leq j$;

if uplo = 'l', the lower triangle of B must be stored, packed by columns, with b_{ij} in b($i+(2n-j)(j-1)/2$) for $i \geq j$.

Output: the upper or lower triangle of `b` is overwritten by the upper or lower triangular Cholesky factor of B, as specified by `uplo`.

Constraints: `b` must be of the same rank and type as `a`.

lambda(:) — real(kind=*wp*), pointer

Output: the m selected eigenvalues in ascending order.

Note: the procedure creates a target array of shape (m). If there are no eigenvalues in the selected interval, then $m = 0$.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

type — integer, intent(in), optional

Input: specifies the type of problem:

if `type` $= 1$, the problem has the form $Az = \lambda Bz$;

if `type` $= 2$, the problem has the form $ABz = \lambda z$;

if `type` $= 3$, the problem has the form $BAz = \lambda z$.

Default: `type` $= 1$.

Constraints: `type` $= 1$, 2 or 3.

il — integer, intent(in), optional

iu — integer, intent(in), optional

Input: the first and last indices, respectively, of the selected eigenvalues, where the eigenvalues are arranged in ascending order. An eigenvalue λ_i is selected if `il` $\leq i \leq$ `iu`.

Default: `il` $= 1$, `iu` $= n$.

Constraints: $\min(n, 1) \leq$ `il` $\leq$ `iu` $\leq n$; `il` and `iu` must not be present if either `vl` or `vu` is present.

vl — real(kind=*wp*), intent(in), optional

vu — real(kind=*wp*), intent(in), optional

Input: the lower and upper bounds, respectively, on the selected eigenvalues. An eigenvalue λ is selected if `vl` $< \lambda \leq$ `vu`.

Default: `vl` $= -\infty$, `vu` $= +\infty$ (i.e. $-$`HUGE(1.0_`*wp*`)` $<$ $\lambda \leq$ `HUGE(1.0_`*wp*`)`).

Constraints: `vl` $\leq$ `vu`; `vl` and `vu` must not be present if either `il` or `iu` is present.

abs_tol — real(kind=*wp*), intent(in), optional

Input: the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is accepted if it has been determined to lie in an interval whose width is less than or equal to `abs_tol`. If `abs_tol` $\leq$ 0.0, then the default value is used.

Default: `abs_tol` $= \epsilon \|C\|_1$, where $\epsilon =$ `EPSILON(1.0_`*wp*`)` and C is defined in Section 6.1.

z(:,:) — real(kind=*wp*) / complex(kind=*wp*), pointer, optional

Output: the m selected eigenvectors. The ith column `z(:,`i`)` holds the eigenvector corresponding to the eigenvalue `lambda(`i`)`. See also `fail`.

Note: the procedure creates a target array of shape (n, m).

If there are no eigenvalues in the selected interval, then $m = 0$. *Constraints:* `z` must be of the same type as `a`.

rcond_b — real(kind=*wp*), intent(out), optional

Output: an estimate of the reciprocal of the condition number of B in the 1-norm. `rcond_b` is set to zero if exact singularity is detected or the estimate underflows. If `rcond_b` is less than `EPSILON(1.0_wp)`, then B is singular to working precision, and the results may be completely unreliable.

fail(:) — integer, pointer, optional

Output: on successful exit, all elements of `fail` are set to 0. If error code 202 is returned, the leading elements of `fail` hold the column indices (in `z`) of those eigenvectors which failed to converge, and the remaining elements are set to 0. For example, if `fail(1)` = 2, the eigenvector in column 2 of `z` failed to converge.

Note: the procedure creates a target array of shape (m).

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
304	Invalid presence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Failure to converge. The bisection algorithm failed to find all the specified eigenvalues.
202	Failure to converge. The inverse iteration algorithm failed to converge to one or more eigenvectors in 5 iterations; the most recent iterate is stored in the corresponding column of `z`. If k eigenvectors failed to converge, their indices are returned in `fail(1 : k)` (if present).
203	Matrix B not positive definite. Either B is close to singularity, or it has at least one negative eigenvalue. The problem could not be reduced to standard form (see Section 6.1).

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure first calls nag_sym_lin_fac to perform a Cholesky factorization of B. It reduces the original generalized problem to an equivalent standard problem $Cy = \lambda y$ with the same eigenvalues, as described in the Module Introduction. It calls nag_sym_eig_sel to compute selected eigenvalues and (if required) eigenvectors of the problem $Cy = \lambda y$. Finally, it recovers the eigenvectors z of the original problem (if required) from the eigenvectors y of the reduced problem. See Chapter 8 of Golub and Van Loan [2] or Parlett [3] for background.

The algorithms are derived from LAPACK (see Anderson *et al.* [1]).

6.2 Accuracy

Error bounds for the computed eigenvalues and eigenvectors involve the condition number of B; the computed eigenvalues and eigenvectors may be inaccurate if B is ill conditioned, that is, if the condition number $\kappa_2(B)$ is large, where $\kappa_2(B) = \|B\|_2\|B^{-1}\|_2$. An approximate estimate for the reciprocal of the condition number of B is returned in the optional argument rcond_b. (rcond_b returns an estimate for the reciprocal of the condition number in 1-norm, $\kappa_1(B)$; this differs by a factor of at most n from the condition number in the 2-norm, $\kappa_2(B)$, which appears in the error analysis.)

In more detail: let λ_i be an exact eigenvalue, and $\tilde{\lambda}_i$ be the corresponding computed value; let z_i be the corresponding exact eigenvector and $\tilde{z}_i$ the computed eigenvector, and let $\theta(\tilde{z}_i, z_i)$ denote the angle between them.

Then for problems of the form $Az = \lambda Bz$:

$$|\tilde{\lambda}_i - \lambda_i| \leq c(n)\epsilon(\|B^{-1}\|_2\|A\|_2 + \kappa_2(B)|\tilde{\lambda}_i|)$$

$$\theta(\tilde{z}_i, z_i) \leq c(n)\epsilon\left(\frac{\|B^{-1}\|_2\|A\|_2(\kappa_2(B))^{1/2} + \kappa_2(B)|\tilde{\lambda}_i|}{\mathrm{gap}_i}\right).$$

For problems of the form $ABz = \lambda z$ or $BAz - \lambda z$:

$$|\tilde{\lambda}_i - \lambda_i| \leq c(n)\epsilon(\|B\|_2\|A\|_2 + \kappa_2(B)|\tilde{\lambda}_i|)$$

$$\theta(\tilde{z}_i, z_i) \leq c(n)\epsilon\left(\frac{\|B\|_2\|A\|_2(\kappa_2(B))^{1/2}}{\mathrm{gap}_i} + \kappa_2(B)|\tilde{\lambda}_i|\right).$$

Here $\epsilon =$ EPSILON(1.0_*wp*), $c(n)$ is a modestly increasing function of n, and $\mathrm{gap}_i = \min_{i \neq j}|\lambda_i - \lambda_j|$.

Example 1: All eigenvalues and eigenvectors of a symmetric-definite generalized eigenvalue problem

Compute all the eigenvalues and eigenvectors of a generalized symmetric-definite eigenvalue problem $Az = \lambda Bz$. This example calls the procedure `nag_sym_gen_eig_all`, using conventional storage.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sym_gen_eig_ex01

  ! Example Program Text for nag_sym_gen_eig
  ! NAG f190, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE nag_sym_gen_eig, ONLY : nag_sym_gen_eig_all
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n
  CHARACTER (1) :: uplo
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: a(:,:), b(:,:), lambda(:), z(:,:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_sym_gen_eig_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, uplo

  ALLOCATE (a(n,n),b(n,n),z(n,n),lambda(n)) ! Allocate storage

  SELECT CASE (uplo)
  CASE ('U','u')
    READ (nag_std_in,*) (a(i,i:),i=1,n)
    READ (nag_std_in,*) (b(i,i:),i=1,n)
  CASE ('L','l')
    READ (nag_std_in,*) (a(i,:i),i=1,n)
    READ (nag_std_in,*) (b(i,:i),i=1,n)
  END SELECT

  ! Compute eigenvalues and eigenvectors

  CALL nag_sym_gen_eig_all(uplo,a,b,lambda,z=z)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Eigenvalues'
  WRITE (nag_std_out,'(2X,6(F9.3:))') lambda
  WRITE (nag_std_out,*)

  CALL nag_write_gen_mat(z,format='(F9.3)',title='Eigenvectors')

  DEALLOCATE (a,b,lambda,z)    ! Deallocate storage
```

```
    END PROGRAM nag_sym_gen_eig_ex01
```

2 Program Data

```
Example Program Data for nag_sym_gen_eig_ex01
4 'L'                   : n and uplo
 0.24
 0.39 -0.11
 0.42  0.79 -0.25
-0.16  0.63  0.48 -0.03 : Matrix A (Lower triangle)
 4.16
-3.12  5.03
 0.56 -0.83  0.76
-0.10  1.09  0.34  1.18 : Matrix B (Lower triangle)
```

3 Program Results

```
 Example Program Results for nag_sym_gen_eig_ex01

 Eigenvalues
     -2.225   -0.455    0.100    1.127

 Eigenvectors
     -0.069   -0.308    0.447    0.553
     -0.574   -0.533    0.037    0.677
     -1.543    0.350   -0.050    0.928
      1.400    0.621   -0.474   -0.251
```

Example 2: Selected eigenvalues and eigenvectors of a Hermitian-definite generalized eigenvalue problem

Compute selected eigenvalues and the corresponding eigenvectors of a generalized Hermitian-definite eigenvalue problem $ABz = \lambda z$. The eigenvalues are selected by index: eigenvalues with indices from `il` to `iu` are computed, the values of `il` and `iu` being read from the data file. This example calls the procedure `nag_sym_gen_eig_sel`, using packed storage.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sym_gen_eig_ex02

  ! Example Program Text for nag_sym_gen_eig
  ! NAG f190, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE nag_sym_gen_eig, ONLY : nag_sym_gen_eig_sel
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, j, n
  REAL (wp) :: vl, vu
  CHARACTER (1) :: uplo
  ! .. Local Arrays ..
  REAL (wp), POINTER :: lambda(:)
  COMPLEX (wp), ALLOCATABLE :: a(:), b(:)
  COMPLEX (wp), POINTER :: z(:,:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_sym_gen_eig_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, uplo
  READ (nag_std_in,*) vl, vu

  ALLOCATE (a(n*(n+1)/2),b(n*(n+1)/2)) ! Allocate storage

  SELECT CASE (uplo)
  CASE ('U','u')
    DO i = 1, n
      READ (nag_std_in,*) (a(i+j*(j-1)/2),j=i,n)
    END DO
    DO i = 1, n
      READ (nag_std_in,*) (b(i+j*(j-1)/2),j=i,n)
    END DO
  CASE ('L','l')
    DO i = 1, n
      READ (nag_std_in,*) (a(i+(2*n-j)*(j-1)/2),j=1,i)
    END DO
    DO i = 1, n
      READ (nag_std_in,*) (b(i+(2*n-j)*(j-1)/2),j=1,i)
    END DO
```

```
      END SELECT

      ! Compute eigenvalues and eigenvectors

      CALL nag_sym_gen_eig_sel(uplo,a,b,lambda,vl=vl,vu=vu,z=z)

      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Selected eigenvalues'
      WRITE (nag_std_out,'(11X,5(F6.3:,10X))') lambda
      WRITE (nag_std_out,*)

      CALL nag_write_gen_mat(z,format='(F6.3)',title='Selected eigenvectors')

      DEALLOCATE (a,b,lambda,z)    ! Deallocate storage

      NULLIFY (lambda,z)

    END PROGRAM nag_sym_gen_eig_ex02
```

2 Program Data

```
Example Program Data for nag_sym_gen_eig_ex02
4 'L'                                                  : n, uplo
-10.00 10.00                                           : vl, vu
(-7.36, 0.00)
( 0.77, 0.43) (3.49, 0.00)
(-0.64, 0.92) (2.19,-4.45) ( 0.12, 0.00)
( 3.01, 6.97) (1.90,-3.73) ( 2.88, 3.17) (-2.54, 0.00) : Matrix A
(3.23, 0.00)
(1.51, 1.92) ( 3.58, 0.00)
(1.90,-0.84) (-0.23,-1.11) (4.09,0.00)
(0.42,-2.50) (-1.18,-1.37) (2.33,0.14) (4.29,0.00)     : Matrix B
```

3 Program Results

```
 Example Program Results for nag_sym_gen_eig_ex02

 Selected eigenvalues
          -5.999          -2.994           0.505           3.999

 Selected eigenvectors
   ( 1.737, 0.106) ( 0.489,-0.501) ( 0.616, 0.194) ( 0.231,-1.216)
   (-0.384,-0.493) ( 0.112,-0.037) ( 0.260,-0.420) (-0.471, 0.481)
   (-0.824,-0.299) (-0.811, 0.411) (-0.037,-0.332) (-0.224, 0.634)
   ( 0.264, 0.628) ( 0.788, 0.200) ( 0.099, 0.659) ( 0.852, 0.000)
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_sym_gen_eig_ex03`

Computes all the eigenvalues and eigenvectors of a generalized Hermitian eigenvalue problem, using conventional storage.

`nag_sym_gen_eig_ex04`

Computes all the eigenvalues and eigenvectors of a generalized Hermitian eigenvalue problem, using packed storage.

`nag_sym_gen_eig_ex05`

Computes selected eigenvalues and the corresponding eigenvectors of a generalized symmetric eigenvalue problem, using conventional storage.

`nag_sym_gen_eig_ex06`

Computes selected eigenvalues and the corresponding eigenvectors of a generalized symmetric eigenvalue problem, using packed storage.

`nag_sym_gen_eig_ex07`

Computes selected eigenvalues and the corresponding eigenvectors of a generalized Hermitian eigenvalue problem, using conventional storage.

`nag_sym_gen_eig_ex08`

Computes all the eigenvalues and eigenvectors of a generalized symmetric eigenvalue problem, using packed storage.

References

[1] Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S and Sorensen D (1995) *LAPACK Users' Guide* (2nd Edition) SIAM, Philadelphia

[2] Golub G H and Van Loan C F (1989) *Matrix Computations* Johns Hopkins University Press (2nd Edition)

[3] Parlett B N (1980) *The Symmetric Eigenvalue Problem* Prentice-Hall