HSL_EA19

1 SUMMARY

HSL_EA19 is designed to compute the n_e leftmost eigenvalues $\lambda_1 \le \lambda_2 \le ... \le \lambda_{n_e}$ and corresponding eigenvectors $x_1, \ldots, x_{n_e}$ of a real symmetric (or Hermitian) operator A acting in the n-dimensional real (or complex) Euclidean space $\mathcal{R}^n$, or, more generally, of the problem

$$Ax = \lambda Bx, \tag{1.1}$$

where B a real symmetric (or Hermitian) positive definite operator. By applying HSL_EA19 to $-A$, the user can compute the n_e rightmost eigenvalues of A and the corresponding eigenvectors. HSL_EA19 does not perform factorizations of A or B and thus is suitable for solving large-scale problems for which a sparse direct solver for factorizing A or B is either not available or is too expensive.

The convergence may be accelerated by the provision of a symmetric positive-definite operator T that approximates the inverse of $(A - \sigma B)$ for a value of σ that initially does not exceed λ_1. The operator T is called *the preconditioner*. The choice of T is discussed further in §2.11 and Technical Report RAL-TR-2010-019 .

We stress that neither A nor B nor T needs to be available explicitly: HSL_EA19 only requires the multiplication of sets of vectors by A, B, and T.

HSL_EA19 implements a subspace iteration method, i.e. it generates a sequence of subspaces $\mathcal{V}^i$, $i = 1, 2, \ldots$, that contain approximations to the eigenvectors of the problem. All subspaces $\mathcal{V}^i$ are of the same dimension $m \ge n_e$. The simplest choice for m is $m = n_e$ but it is desirable that there is a significant gap $\lambda_{m+1} - \lambda_{n_e}$ both for a satisfactory rate of convergence to the rightmost eigenvalues of interest and for the error estimation scheme (see §4.2). The subspace iteration method implemented by HSL_EA19 is based on the Jacobi-conjugate preconditioned gradients (JCPG) scheme of Ovtchinnikov described in [4], [5], and [6]. This method requires at least $4m$ vectors of length n (which include m approximate eigenvectors) and two $2m$-by-$2m$ matrices to be stored in main memory. $2m$ additional vectors of length n are needed if the user opts for reducing the number of multiplications by A; in the case of the generalized problem (1.1), the same applies to B.

The user may supply any number $n_i \le m$ of vectors to be used by the package for the construction of a basis in the initial subspace: this option may be used to reduce the computation time in cases where good approximations to some eigenvectors are available. One way of providing such approximations is by restarting from previously calculated eigenvectors, see §2.8.

ATTRIBUTES — Version: 1.4.2 (6th April 2022). **Date:** July 2007. **Types:** Real (single, double), Complex (single, double). **Calls:** Real: _COPY, _AXPY, _DOT, _NRM2, _SYRK, _GEMM, _SYGV, ILAENV, KB05. Complex: _COPY, _AXPY, _DOTC, SCNRM2/DZNRM2, _HERK, _GEMM, _HEGV, ILAENV, KB07. **Origin:** E. Ovtchinnikov, Harrow School of Computer Science, University of Westminster, London, UK; and J. K. Reid. **Language:** Fortran 95 + TR 15581 (allocatable components).

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

Access to the package requires a USE statement

Single precision version

```
use HSL_EA19_single
```

Double precision version

```
use HSL_EA19_double
```
Complex version

```
use HSL_EA19_complex
```

Double complex version

```
use HSL_EA19_double_complex
```

If it is required to use more than one module at the same time, the derived types (§2.3) must be renamed in all but one of the `use` statements.

To compute several leftmost eigenpairs of the problem (1.1), the user must call the following subroutines:

`EA19_initialize` : must be called once to set the parameters of the problem and initialize private data.

`EA19_solve` : must be called in a reverse communication loop (see §2.4.2).

`EA19_terminate` : must be called once to deallocate all arrays that have been allocated by `EA19_solve`.

Several problems can be solved simultaneously, i.e. the package does not require the solution of one problem to be finished before the solution of the next starts, as long as for each problem a separate set of arguments for the above three subroutines is used. However, if two or more problems of the same type need to be solved, it is reasonable to solve them one after another in order to reduce the amount of memory used.

2.2 Package types

We use the term **package type** to mean default real if the single precision version is being used, double precision real for the double precision version, default complex for the complex version and double precision complex for the double complex version.

We also use **package real type** to mean default real for the single precision and complex versions, and double precision real for the double precision and double precision complex versions.

2.3 The derived data types

The module has three derived types: `EA19_control`, `EA19_keep` and `EA19_info`. The components of `EA19_control` are control parameters that are described in §2.5. All the control parameters are set to their defaults when a variable of the type `EA19_control` is declared. The user can change the value of any control before calling `EA19_initialize`, and the values of some controls at any time, see end of §2.5. The components of `EA19_keep` are used to keep private data between the calls to `EA19_solve`. The components of `EA19_info` contain information about the execution of subroutine `EA19_solve` (see §2.6).

2.4 Argument lists

2.4.1 The initialization subroutine

For each problem, the user must first call the initialization subroutine `EA19_initialize`:

```
call EA19_initialize( control, standard, n, m, nep, kw, keep, err )
```

`control` is a scalar structure of `INTENT(IN)` and of the type `EA19_control`. Its components are described in §2.5.

`standard` is a scalar of `INTENT(IN)` and `LOGICAL` type that defines the type of the problem. For a standard problem ($B = I$), set `standard = .true.`, and for a generalized one set `standard = .false.`.

`n` is a scalar of `INTENT(IN)` and of default `INTEGER` type that holds the size of the problem. **Restriction:** `n` $\geq$ `1`.

`m` is a scalar of `INTENT(IN)` and of default `INTEGER` type that holds the dimension of the iterated subspace. **Restriction:** `1` $\leq$ `m` $\leq$ `n`.

`nep` is a scalar of `INTENT(IN)` and of default `INTEGER` type that holds the number of eigenpairs needed. **Restriction:** $1 \le$ `nep` $\le$ `m`.

`kw` is a scalar of `INTENT(OUT)` and of default `INTEGER` type that holds the third dimension of the work array `W` used by the solver subroutine `EA19_solve`.

`keep` is a scalar structure of `INTENT(OUT)` and of the type `EA19_keep` that holds private data.

`err` is a scalar of `INTENT(OUT)` of default `INTEGER` type that is used as an error flag. On successful return, `err = 0`; non-zero values indicate errors and warnings, which are described in §2.12.

2.4.2 The solver

To compute several leftmost eigenpairs of the problem (1.1), the following subroutine call must be made repeatedly:

```
call EA19_solve( control, lambda, X, W, V, U, ido, nvec, arg, res, keep, info )
```

`control` is a scalar of `INTENT(IN)` and of the type `EA19_control`, whose components are explained in §2.5.

`lambda` is a one-dimensional array of package real type, `INTENT(INOUT)`, and size that is not less than `m`. At no stage of the computation should `lambda` be changed by the user. It holds approximate eigenvalues. When the computation is complete (`ido = -1`), they are in ascending order.

`X` is a two-dimensional array of package type, `INTENT(INOUT)`, and dimensions `n` and `m`. If the user has a set of $n_i \le$ `m` vectors that span an approximation to the space spanned by some of the leftmost eigenvectors, the computation time may be reduced by setting `control%user_X` to n_i and putting these vectors into the first n_i columns of `X` before the first call to `EA19_solve`. They need not be orthonormalized, that is, $X^H BX$ need not be near I. At no other stage of the computation should `X` be changed by the user. `X(:,i)` holds an approximate eigenvector corresponding to the approximate eigenvalue `lambda(i)`, `i` = 1, 2, ... `m`. They are orthonormalized, that is, $X^H BX = I$. On completion (`ido = -1`), the first `nep` approximate eigenpairs satisfy the convergence test (see §2.7).

`W` is a three-dimensional work array of package type, `INTENT(INOUT)`, and dimensions `n`, `m` and `kw`, where `n` and `m` are specified by the user when calling `EA19_initialize` and `kw` is returned by it with the value 3, 5, or 7. `W` must not be changed by the user except as specified in the description of `ido`, `nvec`, `arg` and `res`.

`V` is a three-dimensional work array of package type, `INTENT(INOUT)`, and dimensions `2*m`, `2*m` and `2`, where `m` is specified by the user when calling `EA19_initialize`. At no stage of the computation should `V` be changed by the user.

`U` is a one-dimensional work array of package real type, `INTENT(INOUT)`, and size not less than `2*m`, as specified by the user when calling `EA19_initialize`. At no stage of the computation should `U` be changed by the user.

`ido` is a scalar of `INTENT(INOUT)` and of default `INTEGER` type whose value ranges between `-3` and `4`. Before the first call to `EA19_solve`, `ido` must be set to `0`. For a restart (see §2.8), `ido` must be set to `4`. No other values may be assigned to `ido` by the user. After each call, the value of `ido` must be inspected by the user's code, so that the necessary action is taken, viz.

- `ido = -3` indicates an error return;
- `ido = -2` indicates that the computation has been stopped because the limit, `control%max_it`, on the number of iterations has been reached (for restarting see §2.8);
- if `ido = -1`, then the computation is complete;

- if `ido = 1`, then the user must apply the operator A to the set of vectors specified by the arguments `nvec` and `arg`;
- if `ido = 2`, then the user must apply the operator T to the vectors specified by `nvec` and `arg` (T may be the identity operator);
- if `ido = 3`, the user must apply the operator B to the vectors specified by `nvec` and `arg` (note that this will not happen if `standard = .true.`).

`nvec` is a scalar of `INTENT(OUT)` and of default `INTEGER` type that holds the number of vectors to which A, B or T must be applied.

`arg` is a scalar of `INTENT(OUT)` and of default `INTEGER` type that is given a value if `ido>0`; if `arg>0` then the operator specified by `ido` (A, B or T) must be applied to the columns of `W(:,1:nvec,arg)`, otherwise it must be applied to the columns of `X(:,1:nvec)`; note that `arg` is always positive if `ido=2`.

`res` is a scalar of `INTENT(INOUT)` and of default `INTEGER` type that is given a value if `ido>0`; the `nvec` column vectors resulting from the application of the operator specified by `ido` (A, B or T) must be placed into `W(:,1:nvec,res)`, where `res`$\neq$`arg`.

`keep` is a scalar structure of `INTENT(INOUT)` and of the type `EA19_keep` that holds private data. At no stage of the computation should `keep` be changed by the user.

`info` is a scalar structure of `INTENT(INOUT)` and of the type `EA19_info` that contains information about the execution of the subroutine (see §2.6). At no stage of the computation should `info` be changed by the user.

2.4.3 The terminating subroutine

The memory allocated by `EA19_solve` may be released by making the following subroutine call:

```
EA19_terminate( keep, info )
```

`keep` is a scalar structure of `INTENT(INOUT)` and of the type `EA19_keep`. On exit, its components that are allocatable arrays will have been deallocated.

`info` is a scalar structure of `INTENT(INOUT)` and of the type `EA19_info`. On exit, its components that are allocatable arrays will have been deallocated.

2.5 Derived data type for control parameters

The module contains a derived type called `EA19_control` that has the following components.

`max_it` is a scalar of type default `INTEGER` that contains the maximum number of JCPG iterations to be performed since the previous call with `ido` = `0` or `ido` = `4`. If `minAprod` is true, the number of JCPG iterations is approximately equal to the number of calls with `ido=1`. If `minAprod` is false (its default value), it is approximately one third of the number of calls with `ido=1`. The default is `max_it=100`. **Restriction:** `max_it` $\geq$ `0`.

`abs_tol_lambda` is a scalar of package real type that holds an absolute tolerance used when testing the estimated eigenvalue error, see §2.7. The default value is `abs_tol=0`. Negative values are treated as the default.

`rel_tol_lambda` is a scalar of package real type that holds a relative tolerance used when testing the estimated eigenvalue error, see §2.7. The default value is `10*epsilon(lambda)`. Negative values are treated as the default.

`tol_vector` is a scalar of package real type that holds a tolerance used when testing the estimated eigenvector error, see §2.7. If `tol_vector` is set to zero, the eigenvector error is not estimated. If a negative value is assigned, the tolerance is set to `10*epsilon(lambda)`. The default value is 0.

`abs_tol_residual` is a scalar of package real type that holds an absolute tolerance used when testing the residual, see §2.7. If a negative value is assigned, the tolerance is set to `epsilon(lambda)*`$\sqrt{n}$ times the estimated norm of A. The default value is 0.

`rel_tol_residual` is a scalar of package real type that holds a relative tolerance used when testing the residual, see §2.7. If both `abs_tol_residual` and `rel_tol_residual` are set to 0, then the residual norms are not taken into consideration by the convergence test, see §2.7. If a negative value is assigned, the tolerance is set to `sqrt(epsilon(lambda))`. The default value is 0.

`u_err` is a scalar of type default `INTEGER` that holds the unit number for error messages. Printing is suppressed if `u_err<0`. The default is `u_err=6`.

`u_wrn` is a scalar of type default `INTEGER` that holds the unit number for warning messages. Printing is suppressed if `u_wrn<0`. The default is `u_wrn=6`.

`u_mon` is a scalar of type default `INTEGER` that holds the unit number for messages monitoring the convergence. Printing is suppressed if `u_mon<0`. The default is `u_mon=6`.

`verbosity` is a scalar of type default `INTEGER` that determines the level of detail in printing. Possible values are:

< 0	:	no printing;
0	:	error and warning messages only;
1	:	the type (standard or generalized) and the size of the problem, the number of eigenpairs requested, the error tolerances and the size of the subspace are printed before the iterations start;
2	:	as 1 but, for each eigenpair tested for convergence (see §2.7), the iteration number, the index of the eigenpair, the eigenvalue, whether it has converged, the residual norm, and the error estimates are printed;
> 2	:	as 1 but with all eigenvalues, whether converged, residual norms and eigenvalue/eigenvector error estimates printed on each iteration.

The default is `verbosity=0`. Note that for eigenpairs that are far from convergence, 'rough' error estimates are printed (the estimates that are actually used by the stopping criteria, see §2.7, only become available on the last few iterations).

`minAprod` is a scalar of type default `LOGICAL` that determines whether the number of multiplications by A is to be reduced at the expense of memory. If `minAprod=.false.`, then 3 returns with `ido` = 1 are made for multiplications of `nvec` vectors by A on each iteration. If `minAprod=.true.`, then only one such return is made on each iteration; however, `kw` (the final dimension of the array `W`) is increased by 2 since $2*m$ additional package type vectors of length n need to be stored in this case. The default is `minAprod = .false.`.

`minBprod` is a scalar of type default `LOGICAL` that determines whether the number of multiplications by B is to be reduced at the expense of memory. If `minBprod=.false.`, then 4 returns with `ido` = 3 are made for multiplications of `nvec` vectors by B on each iteration. If `minBprod=.true.`, then only one such return is made on each iteration; however, `kw` (the final dimension of the array `W`) is increased by 2 since $2*m$ additional package type vectors of length n need to be stored in this case. The default is `minBprod=.false.`.

`user_X` is a scalar of type default `INTEGER`. If `user_X>0` then the first `user_X` columns of `X` will be incorporated into the initial subspace $\mathcal{V}^0$ on a call with `ido=0`. Hence, if the user has good approximations to some of the required eigenvectors, the computation time may be reduced by putting these approximations into the first `user_X` columns of `X`. We note that any eigenvectors that are known *a priori* (e.g. the so-called rigid body motions in liner elasticity problems) should be utilized in this way, and that if the desired accuracy has not been achieved within max_it iterations, the iterations can be continued by setting user_X to m and resuming the reverse communication loop (see §2.8). The default is user_X=0, i.e. the columns of X are overwritten by the solver. **Restriction:** $0 \leq$ user_X $\leq$ m.

err_est is a scalar of type default INTEGER that defines which error estimation scheme for eigenvalues and eigenvectors is to be used by the stopping criterion. In the current version of the package, two schemes are implemented. If err_est=1, then residual error bounds are used, notably, a modified Davis-Kahan estimate for the eigenvector error and the Lehmann bounds for the eigenvalue error (see §4.2 and [8]). If err_est=2, then the eigenvector and eigenvalue errors are estimated by analysing the convergence curve for the eigenvalues (see §4.2 and [8]). The default is err_est=2. **Restriction:** err_est=1 or 2.

After the call to EA19_initialize, the values of the components minAprod and minBprod are recorded in keep and subsequent changes to these parameters by the user are ignored. When EA19_solve is called with ido=0 or ido=4, the values of the components max_it, err_est, abs_tol_lambda, rel_tol_lambda, tol_vector, abs_tol_residual, and rel_tol_residual are also recorded in keep; subsequent changes to these parameters by the user are ignored, except that max_it may be made smaller than the value it had the last time EA19_solve was called with ido=0 or ido=4.

2.6 Information returned to the user

The derived type EA19_info has the following components that are used to return information to the user.

flag is an INTEGER scalar that is used as an error flag. If a call is successful, flag has value 0 on exit. A nonzero value of flag indicates an error or a warning (see §2.12.2).

data is an INTEGER scalar that holds additional information about errors and warnings (see §2.12.2).

iteration is an INTEGER scalar that holds the number of iterations since the previous ido = 0 or ido = 4 call.

converged is a one-dimensional allocatable array of type LOGICAL. This array is allocated of size m by EA19_solve at the call with ido=0, and deallocated by EA19_terminate. For i = 1,2,..., m, converged(i) has the value .true. if the eigenvector X(i) has been accepted as converged and .false. otherwise.

res_norms is a one-dimensional allocatable array of package real type. This array is allocated of size m by EA19_solve at the call with ido=0, and deallocated by EA19_terminate. For a converged eigenpair (converged(i) having the value .true.), res_norms(i) contains the Euclidean norm of the residual for the eigenpair lambda(i), X(i) (see §4.2). For a non-converged eigenpair, it holds the residual norm for the current approximate eigenpair lambda(i), X(i) or the corresponding approximation of the previous iteration., or -1.0 if not available. If an estimate is available for the norm $||A||$, then it is worth checking that all values res_norms(i) are substantially above the value $||A||\varepsilon$, where ε is the machine accuracy (the value returned by the Fortran 95 intrinsic function epsilon). Residual vectors whose norm approaches this 'residual floor' value are likely to be too polluted by the round-off errors involved in their calculations to contain any useful information, and it might be better to stop the iterations as they would not make much of a progress in the circumstances (mind that lambda(i) and X(i) are not ordered until full convergence).

err_lmd is a one-dimensional allocatable array of package real type. This array is allocated of size m by EA19_solve at the call with ido=0, and deallocated by EA19_terminate. For a converged eigenpair (converged(i) having the value .true.), err_lmd(i) contains the estimated eigenvalue error for the approximate eigenpair lambda(i), X(i). For a non-converged eigenpair, it holds such an estimate for the current approximate eigenpair lambda(i), X(i) or the corresponding approximation of the previous iteration, or -1.0 if not available. These estimates take no account of round-off error and refer to the errors that would be obtained by solving exactly the eigenvalue problem (4.1) using the array `X` for the matrix V. Eigenvalues of such accuracy may be obtained by restarting the computation in higher precision (see the example in §5.2).

`err_X` is a one-dimensional allocatable array of package real type. This array is allocated of size `m` by `EA19_solve` at the call with `ido=0`, and deallocated by `EA19_terminate`. For a converged eigenpair (`converged(i)` having the value `.true.`), `err_X(i)` contains the estimated Euclidean norm error for the approximate eigenvector `X(i)`. For a non-converged eigenpair, it holds such an estimate for the current approximate eigenpair `lambda(i)`, `X(i)` or the corresponding approximation of the previous iteration, or -1.0 if not available.

2.7 Convergence test

An approximate eigenpair $\{x,\lambda\}$ is considered to have converged if all of the following three conditions are satisfied:

1. if `control%abs_tol_lambda` and `control%rel_tol_lambda` are not both equal to zero, then the estimated error in the approximate eigenvalue must be less than or equal to

   `max(control%abs_tol_lambda,` δ`*control%rel_tol_lambda),`

   where δ is the estimated average distance between eigenvalues.

2. if `control%tol_vector` is not zero, then the estimated sine of the angle between the approximate eigenvector and the invariant subspace corresponding to the eigenvalue approximated by λ must be less than or equal to `control%tol_vector`.

3. if `control%abs_tol_residual` and `control%rel_tol_residual` are not both equal to zero, then the Euclidean norm of the residual, $\|Ax-\lambda Bx\|_2$, must be less than or equal to

   `max(control%abs_tol_residual, control%rel_tol_residual*`$\|\lambda Bx\|_2$`).`

We note that the stopping criteria just described are applied to the first n_e eigenpairs only, i.e. if $n_e < m$ then the errors and residuals of the last $m-n_e$ eigenpairs are not taken into account. Eigenpairs are tested for convergence until the first non-converged one is found; the rest are considered non-converged no matter what the estimated error.

2.8 Restarting the computation

Failure for all the sought eigenpairs to have converged within the limit specified by `control%max_it` is indicated by `ido` having the value `-2`. In this case, `info%data` contains the number of eigenpairs that have not converged to the desired accuracy, either because of the insufficient number of iterations or the use of stopping criteria that are not applicable to some eigenpairs (see the description of `err_est` in §2.5). The remaining eigenpairs can be computed by restarting the computation as follows:

1. Set `control%user_X` to `m`.

2. Set `ido=4`.

3. Consider whether the same limit on number of iterations is appropriate for the new sequence of iterations. If not, reset `control%max_it`.

4. Resume the reverse communication loop described in §2.4.2 with the same `X` and `keep`.

2.9 The use of initial approximations to eigenvectors

If the user has a set of $n_i \leq$ `m` vectors that span an approximation to the space spanned by some of the leftmost eigenvectors, the computation time may be reduced by setting `control%user_X` to n_i and putting these vectors into the first n_i columns of `X` before the first call to `EA19_solve`. They need not be orthonormalized, that is, $X^H BX$ need not be near I. Such approximations may come from previous calls of `EA19_solve` for the problem, perhaps with smaller values of m or higher tolerances, or from a call that failed to compute all wanted eigenpairs.

2.10 The computational cost

Subroutine `EA19_solve` implements an iterative method, and its computational cost depends on the number of iterations and the computational cost of each iteration. The number of iterations can be reduced by preconditioning, described in §2.11. The computational cost of an iteration depends on the computational cost of multiplication by A, B and T, the size of the problem n, the number of iterated vectors m and the values of the control parameters `minAprod` and `minBprod`.

Let us denote by $t(A)$, $t(B)$ and $t(T)$ the CPU times for one multiplication by A, B and T respectively, and by $t(n,m,k)$ the CPU time for the multiplication of an n-by-k matrix by a k-by-m matrix.

If the problem is standard (B is the identity) and `minAprod=.false.`, the CPU time per iteration is approximately

$$3t(A)+t(T)+8t(m_n,m_n,n)+5t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c),$$

where m_c and m_n are the number of converged and non-converged eigenpairs respectively. The required memory is approximately $4mn$ package type variables.

If the problem is standard and `minAprod=.true.`, the CPU time per iteration is approximately

$$t(A)+t(T)+8t(m_n,m_n,n)+9t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c).$$

The required memory is approximately $6mn$ package type variables.

If the problem is generalized, `minAprod=.false.`, and `minBprod=.false.`, then the CPU time per iteration is approximately

$$3t(A)+3t(B)+t(T)+8t(m_n,m_n,n)+5t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c).$$

The required memory is approximately $4mn$ package type variables.

If the problem is generalized, `minAprod=.true.`, and `minBprod=.false.`, then the CPU time per iteration is approximately

$$t(A)+3t(B)+t(T)+8t(m_n,m_n,n)+9t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c).$$

The required memory is approximately $6mn$ package type variables.

If the problem is generalized, `minAprod=.false.`, and `minBprod=.true.`, then the CPU time per iteration is approximately

$$3t(A)+t(B)+t(T)+8t(m_n,m_n,n)+9t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c).$$

The required memory is approximately $6mn$ package type variables.

If the problem is generalized, `minAprod=.true.`, and `minBprod=.true.`, then the CPU time per iteration is approximately

$$t(A)+t(B)+t(T)+8t(m_n,m_n,n)+13t(n,m_n,m_n)+2t(m_c,m_n,n)+2t(n,m_n,m_c).$$

The required memory is approximately $8mn$ package type variables.

The above estimates assume that no nearly-linearly-dependent search direction vectors have been encountered, which is normally the case.

2.11 The use of preconditioning

The estimates of the previous section show that the computational cost of an iteration may be large if large numbers of eigenpairs are needed. Hence, quick convergence is essential. As mentioned in §1, convergence may be accelerated by applying a suitable preconditioner T when requested by `EA19_solve` (i.e., when `ido=2`) and a good initial choice for T is an operator that approximates the inverse of $(A-\sigma B)$ for a shift σ that does not exceed λ_1.

In the real case, one way to compute $V = TU$ is to use an iterative method for solving the system $(A-\sigma B)V = U$, e.g. `MA61A/AD`, which computes an incomplete LDL^T decomposition of a positive-definite matrix. We stress that this system does not have to be solved to a high accuracy; in fact, with `MA61A/AD` the number of iterations can be set to 1, in which case `MA61A/AD` just solves the system $LDL^T V = U$, where L is lower-triangular, D diagonal and $LDL^T \approx A$ (i.e. the preconditioner T is the inverse of LDL^T).

Another possibility is the use of the *multigrid* or *algebraic multigrid* preconditioning (see [1]).

Once k eigenpairs have been computed, it is desirable to use a shift σ that is near λ_k. In this case, preconditioning can be done by approximately solving the system $(A-\sigma B)V = U$ in a subspace of vectors that are B-orthogonal to the converged eigenvectors (we say that vectors u and v are B-orthogonal if $u^H Bv = 0$). We note that `EA19_solve` provides a set of vectors U that are B-orthogonal to all the converged eigenvectors and B-orthogonalizes the set of vectors $V = TX$ to the converged eigenvectors. If the preconditioner setup is expensive, the user does not need to revisit it while k is unchanged. Note that information about the latest converged eigenpair is available in `info` (see §2.6).

2.12 Error codes

A successful return from `EA19_initialize` and `EA19_solve` is indicated respectively by `err` and `info%flag` having the value zero. A negative value indicates an error, a positive value indicates a warning; and, for `EA19_solve`, `info%data` provides further information about some errors and warnings.

2.12.1 Error codes for `EA19_initialize`

Possible negative values of `err` are as follows:

-1 Error in the problem size: `n<1`.

-2 Error in the subspace dimension: `m<1` or `m>n`.

-3 Error in the number of eigenpairs: `nep<1` or `nep>m`.

The value `1` indicates that `n=m`, and hence the Rayleigh-Ritz procedure in the initial subspace is going to be used instead of the conjugate gradient iterations.

2.12.2 Error codes for `EA19_solve`

On return from `EA19_solve`, possible negative values of `info%flag` are as follows:

-1 The initialization subroutine `EA19_initialize` has not been called.

-2 The value of `ido` assigned by the user is neither `0` nor `4`.

-3 Wrong `control%max_it` (negative or greater than the value it had last time `EA19_solve` was called with `ido=0` or `ido=4` – see §2.5).

-4 Error in `control%err_est`: the value of `control%err_est` is neither `1` nor `2`.

-5 control%user_X<0 or control%user_X>m.

-6 All convergence tolerances have the value zero.

-10 Not enough memory; info%data contains the value of the Fortran stat parameter returned by an allocate statement.

-20 Operator B is not positive definite or an unauthorized change has been made to X, U, V or W.

Possible positive values of info%flag are:

1 The number of iterations exceeded control%max_it before all eigenpairs had converged.

2 The iterations have been terminated because no further improvement in accuracy is possible (this may happen if the preconditioner is not positive definite, or if the components of the residual vectors are so small that the round-off errors make them essentially random).

In both cases, the value of info%data is set to the number of non-converged eigenpairs.

3 GENERAL INFORMATION

Other routines called directly: The HSL routine KB07A/AD. The BLAS routines SCOPY/DCOPY/CCOPY/ZCOPY, SAXPY/DAXPY/CAXPY/ZAXPY, SDOT/DDOT/CDOTC/ZDOTC, SNRM2/DNRM2/SCNRM2/DZNRM2, SSYRK/DSYRK/CHERK/ZHERK and SGEMM/DGEMM/CGEMM/ZGEMM. The LAPACK routines ILAENV, SSYGV/DSYGV/CHEGV/ZHEGV.

Other modules used directly: The package contains other modules with names starting HSL_EA19. These should not be accessed directly by the user.

Input/output: Error and warning messages are printed to control%u_err and control%u_wrn output units respectively, and messages monitoring the convergence are printed to the output unit control%u_mon (see §2.5).

Restrictions:

n≥1, 1≤m≤n, 1≤nep≤m

control%max_it≥0

control%user_X≥0

control%err_est=1 or 2

ido should not be set by the user to a value other than 0 or 4.

A, B and T are real symmetric or Hermitian

B and T are positive definite

The convergence tolerances are not all zero

4 METHOD

4.1 The iterative scheme

HSL_EA19 implements the Jacobi-conjugate preconditioned gradients (JCPG) method of Ovtchinnikov described in [4], [5], and [6]. In this method, a sequence of subspaces $\mathcal{V}^i$ of dimension m is generated until a subspace is produced that contains good enough approximations to the required eigenvectors. The iterative scheme that generates $\mathcal{V}^{i+1}$ out of $\mathcal{V}^i$ has two main components: the Rayleigh-Ritz procedure and the conjugate gradient method.

The Rayleigh-Ritz procedure is an algorithm for finding approximations to eigenvectors of a given problem in a given trial subspace. Let V be a matrix whose columns form a basis of the trial subspace $\mathcal{V}$. The approximations in question, called *Ritz vectors* are defined as $\tilde{x}_j = V\hat{x}_j$, where $\hat{x}_j$ are the eigenvectors of the generalized eigenvalue problem

$$V^H AV\hat{x} = \hat{\lambda} V^H BV\hat{x}, \tag{4.1}$$

where V^H is the adjoint matrix for V (transpose of V in the real case). The corresponding eigenvalues of (4.1) are called *Ritz values*. We note that the Ritz vectors are B-orthogonal, i.e. $(B\tilde{x}_i, \tilde{x}_j) = 0$ for $i \neq j$.

The second component, the conjugate gradient method, is a linear search minimization method whereby the new approximation x^{i+1} to the minimum point x of a functional is sought in the direction defined by the linear combination of the gradient g^i and the previous search direction:

$$x^{i+1} = x^i - \alpha_i y^i, \quad y^i = g^i + \beta_i y^{i-1}. \tag{4.2}$$

Here α_i is defined by the condition $(g^{i+1}, y^i) = 0$, which implies that x^{i+1} is the minimum point in the direction defined by the search vector y^i, and β_i is such that the angle between y^i and the optimal search direction $x^i - x$ is minimal possible. In the case of the quadratic functional $\phi(x) = (Lx, x) - 2(f, x)$, where L is symmetric (Hermitian) positive definite, the latter condition on β_i is equivalent to the condition that $(Ly^i, y^{i-1}) = 0$, and y^i is said to be L-conjugate to y^{i-1}. In the case of the eigenvalue problem (1.1), the leftmost eigenvalue λ_1 of which minimizes the Rayleigh quotient functional $\lambda(x) = (Ax, x)/(Bx, x)$, the optimal search direction is found by conjugating with respect to the so-called Jacobi orthogonal complement correction operator – here we refrain from introducing this operator explicitly; further details can be found in [4], [5], and [6]. It remains to note that the convergence of the conjugate gradient iterations (4.2) can be substantially accelerated by applying a properly chosen symmetric (Hermitian) positive definite operator T, referred to as the preconditioner, to the gradient (see e.g. [2]). The theory of preconditioning for eigenvalue computation is fairly difficult – the only comment we can afford here is that preconditioning aims at improving the search direction y^i in (4.2) by moving it closer to the error direction $x^i - x$; further explanations can be found in [4], [5], [6], and [8].

In JCPG the two components are blend as follows. For each Ritz vector x^i_j in the current subspace $\mathcal{V}^i$ a new search direction y^i_j is computed by Jacobi-conjugating the preconditioned gradient Tg^i of the Rayleigh quotient $\lambda(x)$ at $x = x^i_j$ to the previous search directions y^{i-1}_j. The new search directions that are almost linearly dependent are discarded as described in [8]. Assuming that this leaves $k \leq m$ search directions, the Rayleigh-Ritz procedure is then applied in the $(m+k)$-dimensional trial subspace that spans the Ritz vectors x^i_j and the orthogonalized search directions. The m Ritz vectors corresponding to the leftmost Ritz values in this trial subspace define the new subspace iterate $\mathcal{V}^i$.

The computation has two phases. In the first phase, the eigenpairs are tested for convergence from the left. Those that are accepted as converged remain unchanged for the rest of the phase. The new search vectors continue to be B-orthogonalized with respect to the converged eigenvectors, which is the only role the latter play in the computation of the remaining eigenpairs. The first phase completes when all wanted eigenpairs have been accepted. The second phase is like the first phase except that it starts from the vectors in `X` at the end of the first phase and the computation terminates when all the eigenpairs have been accepted as converged.

Further details and the theoretical and numerical investigation of the convergence properties of the JCPG method can be found in [4], [5], [6], and [8].

4.2 Error estimation

4.2.1 Standard problem

The residual stopping criterion (3. in §2.7) is motivated by the well-known fact that for any approximate eigenpair $\{\tilde{\lambda}, \tilde{x}\}$ there exists an eigenvalue λ of A such that

$$|\tilde{\lambda} - \lambda| \le \frac{\|A\tilde{x} - \tilde{\lambda}\tilde{x}\|}{\|\tilde{x}\|} \tag{4.3}$$

(see e.g. [9]).

If `control%err_est = 1`, then the error estimates for the eigenvalues are based on the so-called Lehmann bounds. A detailed discussion on these can be found in [7], where it is shown, in particular, that if, for some $k > 1$, the $(k-1)$-th Ritz value in a given trial subspace is strictly left of the exact eigenvalue λ_k, then each of the leftmost $k-1$ eigenvalues of A is right of the respective eigenvalue $\hat{\lambda}_j$ of the matrix

$$\hat{A} = \tilde{\Lambda}_k - S_k^T S_k, \tag{4.4}$$

where $\tilde{\Lambda}_k$ is a diagonal matrix with the $k-1$ leftmost Ritz values $\tilde{\lambda}_j$ on the diagonal, and the columns of S_k are the respective residual vectors $r_j = A\tilde{x}_j - \tilde{\lambda}_j\tilde{x}_j$ divided by $\sqrt{\lambda_k - \tilde{\lambda}_j}$. The minimax principle for eigenvalues implies that $\lambda_j \le \tilde{\lambda}_j$, and thus the difference $\tilde{\lambda}_j - \hat{\lambda}_j$ estimates the eigenvalue error $\tilde{\lambda}_j - \lambda_j$. Following the recommendation of [7], we replace the unknown value λ_k with $\tilde{\lambda}_k$, and select the maximal $k \le m$ for which the distance between $\tilde{\lambda}_{k-1}$ and $\tilde{\lambda}_k$ exceeds the sum of the absolute error tolerance for eigenvalues and $\|r_j\|^2$, $j = 1, \ldots, k-1$; see the cited paper for the justification of such an approach. If the difference $\tilde{\lambda}_j - \hat{\lambda}_j$ is close to the machine accuracy, it may be too polluted by round-off errors to rely upon. In such case, the following estimate implied by the asymptotics of Lehmann bounds $\hat{\lambda}_j$ (see the cited paper) is used instead:

$$\tilde{\lambda}_j - \lambda_j \le \delta_j \approx \frac{\|r_j\|^2}{\tilde{\lambda}_k - \lambda_j} \tag{4.5}$$

(in the case at hand, the difference between δ_j and the ratio in the right-hand side is of the order of machine accuracy squared). The estimate (4.5) allows one to reach beyond the machine accuracy in the following sense. The relative accuracy in eigenvalues is roughly the square of that for eigenvectors owing to the fact that the Rayleigh quotient is stationary on eigenvectors. Hence, one can compute eigenvectors to single accuracy and then compute Rayleigh quotients for them in double accuracy, thus achieving nearly double accuracy in respective eigenvalues (see §5.2).

The eigenvector errors are estimated based on the Davis-Kahan inequality:

$$\min_{x \in \mathcal{X}_{k-1}} \sin\{\tilde{x}_j; x\} \le \frac{\|r_j\|}{\lambda_k - \tilde{\lambda}_j} \approx \frac{\|r_j\|}{\tilde{\lambda}_k - \tilde{\lambda}_j}, \tag{4.6}$$

where $\mathcal{X}_{k-1}$ is the invariant subspace corresponding to $k-1$ leftmost eigenvalues.

If `control%err_est = 2`, then the eigenvalue error estimates are based on the eigenvalue decrements history (see [8] for details). Unlike the residual estimates mentioned in this section, they are not guaranteed to be upper bounds. However, the numerical tests have demonstrated that these error estimates are significantly more accurate, i.e. closer to the actual error than the other estimates. Furthermore, they straightforwardly apply to the generalized case as well. The eigenvector errors are estimated via eigenvalue errors using the results of [3] (see [8] for details).

4.2.2 Generalized problem

In the case of the generalized eigenvalue problem (1.1), all of the residual norms in the previous section must be replaced with $\|\cdot\|_{B^{-1}}$-norm of the residual $Ax_j^i - \lambda_j^i Bx_j^i$ or its upper estimate, e.g. $\beta_1^{-1/2}\|\cdot\|$, where β_1 is the smallest eigenvalue of B. Hence, if β_1 is known, then the error tolerances for eigenvalues and eigenvectors must be multiplied by β_1 and $\sqrt{\beta_1}$ respectively. If no estimate for $\|\cdot\|_{B^{-1}}$-norm is available, then the use of non-zero residual tolerances and `control%err_est = 1` is not recommended. The user should also bear in mind that the residual estimates we discuss, especially (4.3), may overestimate the actual error considerably, due to the use of the Euclidean norm of the residual, which is too strong a norm for it in the case where A is a discretization of a differential operator.

5 EXAMPLE OF USE

5.1 A simple example

The following code computes the leftmost 8 eigenvalues of the matrix of order 64 that approximates the Laplacian operator on an 8×8 grid.

```
program spec_test ! Laplacian on a square grid
    use hsl_ea19_double
    implicit none
    integer, parameter :: l = 8    ! No. of variables on each side
    integer, parameter :: n = l*l  ! No. of variables
    integer,parameter :: nep = 8   ! No. of eigenpairs wanted
    integer,parameter :: m = 10    ! dimension of the iterated space
    type(ea19_control) :: control  ! control parameters
    type(ea19_keep) :: keep        ! private data
    type(ea19_info) :: info        ! error info
    double precision :: X(n,m), lambda(m)     ! eigenvectors and eigenvalues
    double precision, allocatable :: W(:,:,:) ! work array
    double precision :: V(2*m,2*m,2), U(2*m)  ! work arrays
    integer :: err                  ! error flag
    integer :: ido, nvec, arg, res  ! reverse communication data
    integer :: kw                   ! final extent of W
    integer :: i, j                 ! do indices

    call ea19_initialize( control, .true. , n, m, nep, kw, keep, err )
    allocate( W( n, m, kw ) )
    ido = 0
    do     ! reverse communication loop
      call ea19_solve( control, lambda, X, W, V, U, ido, nvec, arg, res, &
                       keep, info )
      if (ido==-1) exit
       if (ido==1) then
         do j = 1, nvec
           if ( arg > 0 ) call laplace(l,W(:,j,arg),W(:,j,res))
           if ( arg <= 0 ) call laplace(l,X(:,j),W(:,j,res))
        end do
      else
         if ( arg > 0 ) W(:,:,res) = W(:,:,arg)
         if ( arg <= 0 ) W(:,:,res) =  X(:,:)
      end if
    end do
    write (6, '(a,i4,a,/,a)') 'After', info%iteration,   &
```

```
        ' iterations, eigenvalues, residual norms, and', &
        ' estimated eigenvalue errors are:'
    write (6, '(/8x,a)') '  lambda        |r|    est. error'
    do i = 1, nep
       write (6,  '(1x, f18.10, 3es10.0)') lambda(i), info%res_norms(i), &
          info%err_lmd(i)
    end do
    call ea19_terminate( keep,info )
    deallocate( W )
end program spec_test

subroutine laplace(l,x,Ax)
! Multiply x by Laplacian matrix for a square of side l.
   integer, intent(in) :: l
   double precision, intent(in) :: x(l,l)
   double precision, intent(out) :: Ax(l,l)

   integer :: i,j
   double precision :: z

   do i = 1,l
      do j = 1,l
         z = 4d0*x(i,j)
         if(i>1) z = z - x(i-1,j)
         if(j>1) z = z - x(i,j-1)
         if(i<l) z = z - x(i+1,j)
         if(j<l) z = z - x(i,j+1)
         Ax(i,j) = z
      end do
   end do
end subroutine laplace
```

This code produces the following output:

```
After  31 iterations the computed eigenvalues are:
          lambda
       0.2412295169
       0.5885258722
       0.5885258722
       0.9358222275
       1.1206147584
       1.1206147584
       1.4679111138
       1.4679111138
```

5.2 Restarting in higher precision

The following code performs the same computation in single precision and then restarts from the same vectors computing in double precision with a zero limit on the number of iterations. The second computation should improve the eigenvalues to the accuracy anticipated in the first computation. We verify this by comparing them with the analytic values.

```
module constants
    integer, parameter :: l = 8    ! No. of variables on each side
    integer, parameter :: n = l*l  ! No. of variables
    integer,parameter :: nep = 8   ! No. of eigenpairs wanted
    integer,parameter :: m = 10    ! dimension of the iterated space
end module constants

program spec_test ! Laplacian on a square grid
    use constants
    use hsl_ea19_single
    implicit none
    type(ea19_control) :: control  ! control parameters
    type(ea19_keep) :: keep        ! private data
    type(ea19_info) :: info        ! error info
    real :: X(n,m), lambda(m)      ! eigenvectors and eigenvalues
    double precision :: dlambda(m) ! refined eigenvalues
    real, allocatable :: W(:,:,:)  ! work array
    real :: V(2*m,2*m,2), U(2*m)   ! work arrays
    integer :: err                 ! error flag
    integer :: ido, nvec, arg, res ! reverse communication data
    integer :: kw                  ! final extent of W
    integer :: i, j                ! do indices

    call ea19_initialize( control, .true. , n, m, nep, kw, keep, err )
    allocate( W( n, m, kw ) )
    ido = 0
    do     ! reverse communication loop
      call ea19_solve( control, lambda, X, W, V, U, ido, nvec, arg, res, &
                       keep, info )
      if (ido==-1) exit
      if (ido==1) then
        do j = 1, nvec
          if ( arg > 0 ) call laplace(l,W(:,j,arg),W(:,j,res))
          if ( arg <= 0 ) call laplace(l,X(:,j),W(:,j,res))
        end do
      else
         if ( arg > 0 ) W(:,:,res) = W(:,:,arg)
         if ( arg <= 0 ) W(:,:,res) =  X(:,:)
      end if
    end do
    write (6, '(a,i3,a,/,a)') 'After', info%iteration,   &
        ' single-precision iterations, eigenvalues, residual norms, and', &
        ' estimated eigenvalue errors are:'
    write (6, '(8x,a)') '    lambda      |r|    est. error'
    do i = 1, m
       write (6,  '(1x, f18.6, 3es10.0)') lambda(i), info%res_norms(i), &
          info%err_lmd(i)
    end do
    call ea19_terminate( keep,info )
    deallocate( W )
    call refine(X, dlambda)
    call check (dlambda)
contains
   subroutine laplace(l,x,Ax)
   ! Multiply x by Laplacian matrix for a square of side l.
      integer, intent(in) :: l
      real, intent(in) :: x(l,l)
      real, intent(out) :: Ax(l,l)

      integer :: i,j
      real :: z

      do i = 1,l
         do j = 1,l
            z = 4.0*x(i,j)
            if(i>1) z = z - x(i-1,j)
            if(j>1) z = z - x(i,j-1)
            if(i<l) z = z - x(i+1,j)
            if(j<l) z = z - x(i,j+1)
            Ax(i,j) = z
         end do
      end do
   end subroutine laplace

   subroutine check (dlambda)
      double precision ::  dlambda(nep)
      double precision ::  theta, lmda(n)
      integer :: index(n),ij
      ij = 0
      theta = atan(1D0)*4d0/(1d0+l)
      do i = 1, l
        do j = 1, l
          ij = ij+1 ! spectrum of a 2D Laplacian
          lmdA(ij) = 4-2*cos(i*theta)-2*cos(j*theta)
          index(ij) = ij
        end do
      end do
      call kb07ad(lmdA,n,index) ! Sort into ascending order
      write (6, '(/a/8x,a)') 'Actual errors are:','  dlambda       error'
      do i = 1, nep
         write (6,  '(1x, f18.10, es10.0)') dlambda(i),  dlambda(i)-lmdA(i)
      end do
   end subroutine check

end program spec_test
```

```
subroutine refine(XX, dlambda) ! Laplacian on a square grid
   use constants
   use hsl_ea19_double
   implicit none
   real :: XX(n,m)                ! Initial vectors
   double precision :: dlambda(m) ! eigenvalues

   type(ea19_control) :: control  ! control parameters
   type(ea19_keep) :: keep        ! private data
   type(ea19_info) :: info        ! error info
   double precision :: X(n,m)     ! eigenvectors
   double precision, allocatable :: W(:,:,:)  ! work array
   double precision :: V(2*m,2*m,2), U(2*m)   ! work arrays
   integer :: err                  ! error flag
   integer :: ido, nvec, arg, res  ! reverse communication data
   integer :: kw                   ! final extent of W
   integer :: i, j                 ! do indices

   call ea19_initialize( control, .true. , n, m, nep, kw, keep, err )
   control%user_X = m
   control%max_it = 0
   allocate( W( n, m, kw ) )
   ido = 0
   X(:,:) = XX(:,:)
   do     ! reverse communication loop
     call ea19_solve( control, dlambda, X, W, V, U, ido, nvec, arg, res, &
                      keep, info )
     if (ido==-1) exit
     if (ido==1) then
        do j = 1, nvec
           if ( arg > 0 ) call laplace(l,W(:,j,arg),W(:,j,res))
           if ( arg <= 0 ) call laplace(l,X(:,j),W(:,j,res))
        end do
     else
         if ( arg > 0 ) W(:,:,res) = W(:,:,arg)
         if ( arg <= 0 ) W(:,:,res) =  X(:,:)
     end if
   end do
   write (6, '(/,a,i3,a,/,a)') 'After', info%iteration,   &
        ' double-precision iterations, eigenvalues, residual norms, and', &
        ' estimated eigenvalue errors are:'
   write (6, '(8x,a)') '  dlambda       |r|    est. error'
   do i = 1, nep
       write (6,  '(1x, f18.10, 3es10.0)') dlambda(i), info%res_norms(i), &
          info%err_lmd(i)
   end do
   call ea19_terminate( keep,info )
   deallocate( W )
contains
```

```
   subroutine laplace(l,x,Ax)
   ! Multiply x by Laplacian matrix for a square of side l.
      integer, intent(in) :: l
      double precision, intent(in) :: x(l,l)
      double precision, intent(out) :: Ax(l,l)

      integer :: i,j
      double precision :: z

      do i = 1,l
         do j = 1,l
            z = 4.0*x(i,j)
            if(i>1) z = z - x(i-1,j)
            if(j>1) z = z - x(i,j-1)
            if(i<l) z = z - x(i+1,j)
            if(j<l) z = z - x(i,j+1)
            Ax(i,j) = z
         end do
      end do
   end subroutine laplace

end subroutine refine
```

This code produces the following output:

```
After 18 single-precision iterations, eigenvalues, residual norms, and
 estimated eigenvalue errors are:
           lambda      |r|    est. error
          0.241230    1.E-05    1.E-10
          0.588526    2.E-05    4.E-10
          0.588526    1.E-05    1.E-10
          0.935822    1.E-05    3.E-10
          1.120615    5.E-06    5.E-11
          1.120615    5.E-06    5.E-11
          1.467911    4.E-05    3.E-09
          1.467911    4.E-05    5.E-09
          1.773319    9.E-04    9.E-10
          1.773318    3.E-05    3.E-05
After  0 double-precision iterations, eigenvalues, residual norms, and
 estimated eigenvalue errors are:
         dlambda       |r|    est. error
      0.2412295169    1.E-05    1.E-05
      0.5885258722    1.E-05    1.E-05
      0.5885258724    2.E-05    2.E-05
      0.9358222276    1.E-05    1.E-05
      1.1206147584    8.E-07    8.E-07
      1.1206147584    7.E-06    7.E-06
      1.4679111138    7.E-06    7.E-06
      1.4679111149    5.E-05    5.E-05
Actual errors are:
   dlambda       error
0.2412295169    2.E-11
0.5885258722    3.E-11
0.5885258724    2.E-10
0.9358222276    4.E-11
1.1206147584    2.E-13
1.1206147584    1.E-11
1.4679111138    2.E-11
1.4679111149    1.E-09
```

References

[1] W. Briggs, Van Emden Henson, S. McCormick. A Multigrid Tutorial. Cambridge University Press, 2000.

[2] Y. T. Feng and D. R. Owen. Conjugate gradient methods for solving the smallest eigenpair of large symmetric eigenvalue problems, *Int. J. Numer. Meth. Engrg.*, **39**, 2209–2229, 1996.

[3] E. E. Ovtchinnikov. Cluster robust error estimates for the Rayleigh-Ritz approximation I: Estimates for invariant subspaces, *Linear Algebra Appl.*, **415**, 167–187, 2006.

[4] E. E. Ovtchinnikov. Computing several eigenpairs of Hermitian problems by conjugate gradient iterations, *J. Comput. Phys.*, **227**, 9477–9497, 2008.

[5] E. E. Ovtchinnikov. Jacobi correction equation, line search and conjugate gradients in Hermitian eigenvalue computation I: Computing an extreme eigenvalue, *SIAM J. Numer. Anal.*, **46**, 2567–2592, 2008.

[6] E. E. Ovtchinnikov. Jacobi correction equation, line search and conjugate gradients in Hermitian eigenvalue computation II: Computing several extreme eigenvalues, *SIAM J. Numer. Anal.*, **46**, 2593–2619, 2008.

[7] E. E. Ovtchinnikov. Lehmann bounds and eigenvalue error estimation. *Linear Algebra Appl.*, submitted in August 2009.

[8] E. E. Ovtchinnikov and J. K. Reid. A preconditioned block conjugate gradient algorithm for computing extreme eigenpairs of symmetric and Hermitian problems. Technical Report RAL-TR-2010-019, 2010.

[9] B. N. Parlett. *The Symmetric Eigenvalue Problem.* SIAM, 1998.