EIGIFP: A MATLAB Program for Solving Large Symmetric Generalized Eigenvalue Problems

JAMES H. MONEY[†] and QIANG YE[*]
UNIVERSITY OF KENTUCKY

eigifp is a MATLAB program for computing a few extreme eigenvalues and eigenvectors of the large symmetric generalized eigenvalue problem $Ax = \lambda Bx$. It is a black-box implementation of an inverse free preconditioned Krylov subspace projection method developed by Golub and Ye (2002). It has some important features that alow it to solve some difficult problems without any input from users. It is particularly suitable for problems where preconditioning by the standard shift-and-invert transformation is not feasible.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematical Software**]: Documentation

General Terms: Algorithms, Documentation

Additional Key Words and Phrases: Krylov subspace methods, eigenvalue, generalized eigenvalue problem, preconditioning

1. INTRODUCTION

eigifp is a MATLAB program for computing a few algebraically smallest or largest eigenvalues and their corresponding eigenvectors of the generalized eigenvalue problem

$$Ax = \lambda Bx \tag{1}$$

where A and B are large (and typically sparse) symmetric matrices and B is positive definite. This eigenvalue problem, sometimes referred to as a pencil eigenvalue problem for (A, B), arises in a large variety of scientific and engineering applications, see [Golub and Van Loan 1983; Parlett 1980; Saad 1992] for details. Solving them efficiently for large scale problems is of great importance.

The underlying algorithm of eigifp is an inverse free preconditioned Krylov subspace projection method developed in [Golub and Ye 2002]. In this method, we iteratively improve an approximate eigenvector x_k through the Rayleigh-Ritz projection on the Krylov subspace of dimension m generated by $A - \rho_k B$ and x_k, where $\rho_k = x_k^T Ax_k / x_k^T Bx_k$. The projection is carried out by constructing a basis for the Krylov subspace through an inner iteration, where the matrices A and B are only used to form matrix-vector products and $\mathcal{O}(m)$ vector memory is required. The method is proved to converge at least linearly and a congruence transformation can be implicitly applied to precondition the original eigenvalue problem so as to accelerate convergence. eigifp implements this preconditioned algorithm and has also incorporated several algorithmic enhancements and implementation details to

[†] Department of Mathematics, University of Kentucky, Lexington, KY 40506-0027. E-mail: jmoney@ms.uky.edu. Research supported in part by NSF under Grant No. CCR-0098133.
[*] Department of Mathematics, University of Kentucky, Lexington, KY 40506-0027. E-mail: qye@ms.uky.edu. Research supported in part by NSF under Grant No. CCR-0098133.

arrive at an efficient black-box implementation.

Comparing with existing programs, `eigifp` possesses some important features that alow it to solve some difficult problems without any input from users. First, for the generalized eigenvalue problem (with $B \neq I$), `eigifp` does not require inverting B as most other methods do. Second, it includes the congruent transformation based preconditioning and this is done with no required user inputs such as a user supplied preconditioner or a shift (as in a shift-and-invert transformation). The program uses an incomplete LDL^T factorization of a shifted matrix $A - \sigma B$ to generate a preconditioner, where the shift σ is an approximate eigenvalue determined also by the program. With the use of the incomplete factorization, the computational and memory cost of preconditioning can be controlled and the preconditioning is implemented in the code in a black-box fashion. Thus, `eigifp` will be most useful for problems where preconditioning by the standard shift-and-invert transformation is not feasible; but it can be competitive as well otherwise. Finally, `eigifp` is relatively simple in implementation with only one performance tuning parameter (i.e. the dimension of the Krylov subspace). This parameter can be either set by users or adaptively chosen by the program.

Over the years, many numerical methods and softwares have been developed to solve large scale eigenvalue problems, including many publicly distributed programs; we refer to [Bai et al. 2000] for a comprehensive list of references and softwares in this area. The links to most of publicly distributed programs can also be found in [Bai et al. 2000]. A large majority of the programs are based on the Lanczos algorithm, including ARPACK (implemented in the MATLAB built-in function `eigs`) of [Lehoucq et al. 1998] and `irbleigs` of [Baglama et al. 2003]. Methods of this type require inverting B and, when eigenvalues are badly separated, they typically need to use a shift-and-invert transformation, which is not always feasible or efficient. Other programs such as JDQZ [Fokkema et al. 1999; Sleijpen and van der Vorst 1996] and LOPBCG [Knyazev 2001] do not require inverting B or a shift-and-invert transformation, but they appear to require more user inputs, such as an initial approximate eigenvalue or preconditioners.

The paper is organized as follows. In Section 2, we give a brief description of the inverse free preconditioned Krylov subspace method and some enhancements developed for `eigifp` . In Section 3, we present details in calling `eigifp` , which is followed by some examples in Section 4.

2. INVERSE FREE PRECONDITIONED KRYLOV SUBSPACE METHOD

The core of `eigifp` is the inverse free preconditioned Krylov subspace projection method developed in [Golub and Ye 2002]. We describe in this section first this basic method and then some enhancements incorporated into `eigifp` .

Throughout, we shall consider the smallest eigenvalue of (A, B). Indeed, a direct call to `eigifp` computes the k smallest eigenvalues. To compute the largest eigenvalue of (A, B), we just need to compute the smallest eigenvalue of $(-A, B)$ and reverse the sign to obtain the largest eigenvalue of (A, B).

2.1 Basic Method

Given an approximate eigenvector x_k, we construct a new approximation x_{k+1} by the Rayleigh-Ritz projection of (A, B) onto the Krylov subspace

$$K_m(A - \rho_k B, x_k) \equiv \text{span}\{x_k, (A - \rho_k B)x_k, \ldots, (A - \rho_k B)^m x_k\}$$

where $\rho_k = x_k^T A x_k / x_k^T B x_k$ is the Rayligh quotient and m is a parameter to be chosen. Specifically, let Z_m be the matrix consisting of the basis vectors of K_m. We then form the matrices

$$A_m = Z_m^T (A - \rho_k B) Z_m$$

and

$$B_m = Z_m^T B Z_m,$$

and find the smallest eigenpair (μ_1, v_1) for (A_m, B_m). Then the new approximate eigenvector is

$$x_{k+1} = Z_m v_1$$

and, correspondingly, the Rayleigh quotient

$$\rho_{k+1} = \rho_k + \mu_1$$

is a new approximate eigenvalue.

Iterating with k, the above forms the outer iteration of the method. Now, to construct the basis vectors Z_m, an inner iteration will be used. We use either the Lanczos algorithm to compute an orthonormal basis or the Arnoldi algorithm to compute a B-orthonormal basis. While in theory the outer iteration is independent of the bases constructed, they have different numerical stability. Experiments lead us to use an orthonormal basis by the Lanczos method when the outer iteration is not preconditioned and to use a B-orthonormal basis by the Arnoldi algorithm when the outer iteration is preconditioned (see below).

It is shown in [Golub and Ye 2002] that ρ_k converges to an eigenvalue and x_k converges in direction to an eigenvector. Furthermore, we have the following local convergence result.

THEOREM 2.1. *Let $\lambda_1 < \lambda_2 \leq \cdots \leq \lambda_n$ be the eigenvalues of (A, B) and (ρ_{k+1}, x_{k+1}) be the approximate eigenpair obtained from (ρ_k, x_k) by one step of the inverse free Krylov subspace method. Let $\sigma_1 < \sigma_2 \leq \cdots \leq \sigma_n$ be the eigenvalues of $A - \rho_k B$. If $\lambda_1 < \rho_k < \lambda_2$, then*

$$\rho_{k+1} - \lambda_1 \leq (\rho_k - \lambda_1)\epsilon_m^2 + \mathcal{O}((\rho_k - \lambda_1)^{3/2}) \tag{2}$$

where

$$\epsilon_m = \min_{p \in \mathcal{P}_m, p(\sigma_1)=1} \max_{i \neq 1} |p(\sigma_i)| \leq 2 \left(\frac{1 - \sqrt{\psi}}{1 + \sqrt{\psi}} \right)^m$$

$\mathcal{P}_m$ denotes the set of all polynomials of degree not greater than m and

$$\psi = \frac{\sigma_2 - \sigma_1}{\sigma_n - \sigma_1}.$$

The bound shows that the speed of convergence depends on ψ, the relative gap between σ_1 and σ_2, in addition to m. We also note that a key condition of the theorem is that $\rho_k \in (\lambda_1, \lambda_2)$.

Now we would like to speed up the convergence by increasing the spectral gap ψ through an equivalent transformation that we call preconditioning. One way of doing this is the congruence transformation

$$(\hat{A}, \hat{B}) \equiv (L^{-1}AL^{-T}, L^{-1}BL^{-T}), \tag{3}$$

which preserves the eigenvalues but changes σ_i. Indeed, applying our algorithm to $(\hat{A}, \hat{B})$, the speed of convergence depends on the spectral gap of

$$\hat{A} - \rho_k \hat{B} = L^{-1}(A - \rho_k B)L^{-T}.$$

By choosing L to be the factor in the LDL^T factorization of $A - \rho_k B$ with D being a diagonal matrix of ± 1, we obtain an ideal situation of $\psi = 1$ and hence $\epsilon_m = 0$. In practice, we can use an incomplete LDL^T factorization to arrive at a small ϵ_m.

As in the preconditioned conjugate gradient method, preconditioning transformation (3) can be carried out implicitly. See [Golub and Ye 2002] for a detailed algorithm. Indeed, the only operation involving the preconditioning transformation is $L^{-T}L^{-1}$. Thus, if M is approximately $A - \lambda_1 B$ and is symmetric positive definite, then we only need M^{-1} to implement the preconditioned iteration. We call M^{-1} a preconditioner, which need not be in the factorized form $L^{-T}L^{-1}$.

2.2 LOBPCG Type Subspaces Enhancement

Our algorithm reduces to the steepest descent method when $m = 1$. [Knyazev 2001; 1998] has derived a method called locally optimal preconditioned conjugate gradient method (LOBPCG), where the previous approximate eigenvector x_{k-1} in the steepest descent method is added to span$\{x_k, (A - \rho_k B)x_k\}$ and a new approximate eigenvector is constructed from span$\{x_{k-1}, x_k, (A - \rho_k B)x_k\}$ by projection. It results in a conjugate gradient like algorithm and Knyazev has observed a dramatic speedup in convergence over the steepest descent method.

Here, we also apply this technique to our method to enhance the Krylov subspace $K_m(A - \rho_k B, x_k)$, namely, at step k, we use the Rayleigh-Ritz projection on the enhanced subspace span$\{x_{k-1}, x_k, (A - \rho_k B)x_k, \ldots, (A - \rho_k B)^m x_k\}$. In `eigifp` , we compute $x_k - x_{k-1}$ at every step and, when a basis Z_m has been constructed for $K_m(A - \rho_k B, x_k)$, we orthogonalize $x_k - x_{k-1}$ against Z_m to obtain z_{m+2} and then extend the basis matrix to

$$\hat{Z}_m = \begin{bmatrix} Z_m & z_{m+2} \end{bmatrix}.$$

Our experiments have shown that this provides noticeable speedup in convergence; yet it only incurs very little extra cost.

2.3 Deflation

The algorithm we have described finds the smallest eigenvalue. Once it is done, we can go to find the next smallest eigenvalue by the same procedure through deflation. Because of the form of the Krylov subspace, the deflation needs to be done slightly differently from standard methods like the Lanczos algorithm.

When p eigenpairs have been found, let V_p be the matrix consisting of the p eigenvectors with $V_p^T B V_p = I$ and Λ_p be the diagonal matrix consisting of the corresponding eigenvalues, i.e. $AV_p = BV_p\Lambda_p$. Then, we consider

$$(A_p, B) \equiv (A + (BV_p)\Sigma(BV_p)^T, B) \tag{4}$$

where $\Sigma = \mathrm{diag}\{\sigma_i - \lambda_i\}$ with σ_i any value chosen to be greater than λ_{p+2}. Then, it is easy to check that the eigenvalues of (4) are the union of $\{\lambda_{p+1}, \lambda_{p+2}, \cdots, \lambda_n\}$ and $\{\sigma_1, \cdots, \sigma_p\}$. Thus, its smallest eigenvalue is λ_{p+1} and, by applying our method to (4), we find λ_{p+1}.

2.4 Black-box Implementations

In order to implement our method in a black-box routine, we need to address the following issues:

(1) How do we carry out preconditioned iterations without users' input of a preconditioner?

(2) How do we choose the number of inner iterations to optimize the overall performance?

(3) When do we terminate the iteration?

We describe now how these are dealt with in `eigifp` . We note that users always have options to do differently from the default settings (see Section 3).

To implement the preconditioned iteration, we use an approximate eigenvalue σ and compute an incomplete LDL^T factorization of $A - \sigma B$ using the MATLAB threshold ILU routine `luinc` (with a default threshold 10^{-3}). If an initial approximation σ is not provided by users, `eigifp` will start with the non-preconditioned iterations and switch to the preconditioned one when a good approximate eigenvalue is identified. For this, we first estimate the error $\lambda_1 - \rho_k$ using the eigenvector residual and an estimated gap between the first two eigenvalues of $A - \rho_k B$. We then switch to the preconditioned iterations when the error is below a certain threshold. As a safeguard against that the switching may occur too early, we reverse it back to the non-preconditioned iteration if the subsequent approximate eigenvalues ρ_k significantly drift away from the shift point chosen.

The only parameter to be chosen in our method is the number of inner iterations m. Experiments have shown that an optimal value of m is larger if the problem is more difficult while it is smaller if the problem is easier (e.g. with a good preconditioner). However, we do not know which value works best for a given matrix. Without users' input, we adaptively choose m in `eigifp` as follows. Starting with a small value of m (2 for non-preconditioned iterations and 1 for preconditioned iterations), we double the value m and compute its convergence rate after some iterations. We continue increasing m as long as the rate of convergence has been roughly double, but reset it to the previous value when the rate of convergence is not increased proportionally.

Finally, we terminate the iteration when the 2-norm of the residual $r_k = (A - \rho_k B)x_k$ drops below a certain threshold (where $\|x_k\|_2 = 1$). The default threshold is

$$\|r_k\|_2 \le p(n)\epsilon(\|A\| + \|\rho_k B\|), \tag{5}$$

where ϵ is the machine roundoff unit and we set $p(n) = 10\sqrt{n}$. We note that if $p(n)$ is the maximal number of non-zero entries in each row of the matrices, (5) is approximately the size of roundoff errors encountered in computing the residual $r_k = (A - \rho_k B)x_k$ and would be the smallest threshold one can expect.

When `eigifp` terminates with a converged eigenpair, its residual satisfies the termination criterion (5). It can be easily checked that (ρ_k, x_k) is an exact eigenvalue and eigenvector of a slightly perturbed problem, i.e.

$$(A + E)x_k = \rho_k(B + F)x_k$$

where

$$\frac{\|E\|_2}{\|A\|_2} \leq p(n)\epsilon \text{ and } \frac{\|F\|_2}{\|B\|_2} \leq p(n)\epsilon.$$

3. EIGIFP CALLS

`eigifp` has a simple calling structure but also take several options. We will first discuss the basic calling structure of the program, working our way up to the more complex cases, and finally introduce all the advanced optional arguments a user can use to optimize performance.

The most basic call to `eigifp` is

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}(\texttt{A})$$

where A is a matrix in sparse format. This returns the smallest eigenvalue of A `Evalues` and and its corresponding eigenvector `Evectors`.

To compute the k smallest eigenpairs of the matrix A, one appends the value k to the above call

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}(\texttt{A}, \texttt{k})$$

where $k \geq 1$ is an integer. Then the returned results are a vector of k smallest eigenvalues `Evalues` and an $n \times k$ matrix of the corresponding eigenvectors `Evectors`.

Now, for solving the pencil eigenvalue problem of (A, B) which is $Ax = \lambda Bx$, one appends B after A in the above calls, namely,

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}(\texttt{A}, \texttt{B})$$

returns the smallest eigenpair and

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}(\texttt{A}, \texttt{B}, \texttt{k})$$

returns the k smallest eigenpairs as above.

In all cases, replacing A by $-A$ in the input argument and multiplying the output by -1 compute the largest eigenvalue correspondingly. For example,

$$>> [\texttt{Evalues}, \texttt{Evectors}] = -\texttt{eigifp}(-\texttt{A}, \texttt{B}, \texttt{k})$$

returns the k largest eigenpairs of (A, B).

`eigifp` also uses an option structure to provide extra information to the algorithm and to help improve performance. Users can pass a set of optional parameters via a structure in MATLAB. This is done by first setting the value in the structure, e.g.

$$>> \texttt{opt.initialvec} = \texttt{ones}(\texttt{n}, \texttt{1})$$

and then pass `opt` to `eigifp` by calling

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}(\texttt{A}, \texttt{B}, \texttt{k}, \texttt{opt})$$

One can pass less parameters, as long as the `opt` structure is the last parameter in the list.

The following discuss various optional parameters one can set.

3.1 Informational Options

Below is a list of the informational options `eigifp` takes:

`opt.size`	The dimension of A.
`opt.normA`	An estimate of the norm of A.
`opt.normB`	An estimate of the norm of B.
`opt.tolerance`	Sets the termination threshold for the norm of the residual.
`opt.maxiterations`	Set the maximal number of outer iterations to perform.

The first three options are primarily used when the matrices are passed as functions (discussed in Sec.3.3 below). The first one provides the dimension of vectors and is required in the use of functions. The other two are optional, and estimate the norm of A and B. If they are not provided, then the norms are estimated by $\|Ax\|$ and $\|Bx\|$ for a random vector x.

The `tolerance` option allows the user to specify a tolerance different from default (5). It can be set to be much smaller (e.g. of $\mathcal{O}(\sqrt{\epsilon})$) if only eigenvalues are desired.

The `maxiterations` option lets the user specify how many outer iterations `eigifp` will perform before it terminates without convergence. The user can set a larger value for particularly difficult problems. The default is set at 500.

3.2 Performance Options

Below is a list of performance options one can specify to optimize the performance.

`opt.initialvec`	An $n \times k$ matrix whose j-th column is the initial guess for the j-th eigenvector.
`opt.inneriteration`	Set the number of inner iteration (i.e. the dimension of the Krylov subspaces).
`opt.useprecon`	Setting this to 'NO' causes no preconditioning to happen. Otherwise, this is assumed to be the initial shift to be used for computing the preconditioner.

`opt.iluthresh`	Set the threshold used in the incomplete LU factorization that is called for preconditioning. Setting it to 0 will lead to full (exact) factorization while setting it to 1 corresponds to incomplete factorization with no fill-in.
`opt.preconditioner`	Input a matrix or a function to be used as a user specified preconditioner.

The `initialvec` option allows the user to input initial eigenvector guesses and should be provided whenever they are available. If it is an $n \times p$, then `initialvec(:,1)` is used as the initial vector for the first eigenvalue, `initialvec(:,2)` for the second, and so on.

The `inneriteration` option allows the user to set a fixed inner iteration. This will disable the adaptive setting by default. It can be used either to limit memory usage in the inner iteration or to improve convergence for more difficult problems. Thus, it should be set to a small value if `eigifp` runs out of memory but to a larger value when a very slow convergence is observed.

The `useprecon` option allows the user either to disable preconditioning totally when computing the incomplete factorization is not efficient, or to input an approximate eigenvalue to be used as a shift for computing a preconditioner. The latter setting saves the program to search for an initial approximation and will use the preconditioned iteration throughout.

The `iluthresh` option sets the threshold value for computing the incomplete LDL^T decomposition. Setting it to a smaller value leads to a better preconditioner which is more expensive to compute. The default value is 10^{-3}.

Finally, the `preconditioner` option allows the user to supply either a matrix or a function for the preconditioner. If it is a matrix, it should be the factor L; if it is a function, it should perform $L^{-T}L^{-1}x$ for an input x.

3.3 Matrices as Functions

In addition to passing A and B as sparse matrices, one can pass in the name of functions to call. The function should be of the form

$$>> \texttt{Ax} = \texttt{functionA(x)}$$

where `functionA` returns the result of $A * x$ as a vector. Similarly, one can provide a function call for B. In this case, the dimension of the matrices must be passed through `opt.size`. Then, one can call `eigifp` as

$$>> [\texttt{Evalues}, \texttt{Evectors}] = \texttt{eigifp}('\texttt{functionA}', \texttt{k}, \texttt{opt})$$

It is also desirable to pass the norms of the matrices through `opt.normA` and `opt.normB`.

4. EXAMPLES

In this section, we will present some examples of calling `eigifp` and its outputs (with part of on-screen printout omitted) from execution. All the executions were carried out using MATLAB version 6.0 with the most recent patches from MathWorks on a Pentium III Xeon 1.8Ghz with 1GB of RAM. Below, A is the 7668×7668 matrix from discretizing the 2 dimension Laplacian operator on a unit disc using a 5 point stencil. The boundary conditions are Dirichlet boundary conditions. We generate the sparse matrix in MATLAB and call `eigifp` as follows

```
 >>A=delsq(numgrid('D',100));
 >>eigenvalues=eigifp(A)
  Eigenvalue 1 converged.
  # of multiplications by A (and B):      196.
  # of multiplications by preconditioner: 8.

-----
  CPU Time:2.890000.

eigenvalues =

    0.0023
```

For the pencil problem, we use a diagonal matrix for B.

```
>>B=spdiags([1:size(A,1)]',0,size(A,1),size(A,2));
>>eigenvalues=eigifp(A,B)
   Eigenvalue 1 converged.
  # of multiplications by A (and B):      153.
  # of multiplications by preconditioner: 9.

-----
  CPU Time:3.780000.

eigenvalues =

   5.5653e-07
```

To compute the first three eigenpairs of (A, B), we use:

```
>>eigenvalues=eigifp(A,B,3)
   Eigenvalue 1 converged.
  # of multiplications by A (and B):      165.
  # of multiplications by preconditioner: 9.
   Eigenvalue 2 converged.
  # of multiplications by A (and B):      39.
  # of multiplications by preconditioner: 37.
  Eigenvalue 3 converged.
  # of multiplications by A (and B):      18.
  # of multiplications by preconditioner: 16.

-----
  CPU Time:7.910000.

eigenvalues =
   1.0e-05 *

    0.0557
    0.1365
    0.1557
```

Now we present some examples using the options. The following specifies an initial guess for x_0 and a tolerance of 10^{-5}:

```
>>opt.initialvec=ones(size(A,1),1);
>>opt.tolerance=1e-5;
>>eigenvalues=eigifp(A,B,opt)
   Eigenvalue 1 converged.
  # of multiplications by A (and B):      64.
  # of multiplications by preconditioner: 2.

-----
  CPU Time:2.020000.

eigenvalues =

   5.5653e-07
```

Next, we can disable preconditioning totally by setting the `useprecon` option.

```
>>opt.useprecon='NO';
>>eigenvalues=eigifp(A,B,opt)
   Eigenvalue 1 converged.
  # of multiplications by A (and B):      146.

-----
  CPU Time:2.770000.

eigenvalues =

   5.5653e-07
```

On the other hand, since 0 is an approximate eigenvalue here, we can use it as a shift to start the preconditioned iterations by setting `opt.useprecon` to 0.

```
>>opt.useprecon=0;
>>eigenvalues=eigifp(A,B,opt)
  Eigenvalue 1 converged.
  # of multiplications by A (and B):      7.
  # of multiplications by preconditioner: 5.

-----
  CPU Time:1.000000.

eigenvalues =
   5.5653e-07
```

Finally, here is an example to manually set a fixed value for the inner iterations:

```
>>opt.inneriteration=32;
>>eigenvalues=eigifp(A,B,opt)
  Eigenvalue 1 converged.
  # of multiplications by A (and B):      301.
  # of multiplications by preconditioner: 107.

-----
  CPU Time:10.990000.

eigenvalues =

   5.5653e-07
```

Acknowledgement: We would like to thank Andrew Knyazev for suggesting the approach of adding the previous approximate eigenvector to the Krylov subspace in Section 2.1.

REFERENCES

Baglama, J., Calvetti, D., and Reichel, L. 2003. Algorithm 827: irbleigs: A MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix. *ACM Transactions on Mathematical Software 29,* 3 (Sept.), 337–348.

Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. 2000. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia.

Fokkema, D., Sleijpen, G., and van der Vorst, H. 1999. Jacobi-davidson style qr and qz algorithms for the reduction of matrix pencils. *SIAM Journal on Scientific Computing 20*, 94–125.

Golub, G. and Van Loan, C. 1983. *Matrix Computations.* The Johns Hopkins University Press, Baltimore.

Golub, G. and Ye, Q. 2002. An inverse free preconditioned krylov subspace methods for symmetric generalized eigenvalue problems. *SIAM Journal of Scientific Computation 24*, 312–334.

Knyazev, A. 1998. Preconditioned eigensolvers - an oxymoron? *Electronic Transactions on Numerical Analysis 7*, 104–123.

Knyazev, A. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal of Scientific Computation 23*, 517–541.

Lehoucq, R., Sorenson, D., and Yang, C. 1998. *ARPACK Users' Guides, Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Method.* SIAM, Philadelphia.

Parlett, B. 1980. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, N.J.

Saad, Y. 1992. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester, UK.

Sleijpen, G. and van der Vorst, H. 1996. A jacobi-davidson iteration method for linear eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications 17*, 401–425.