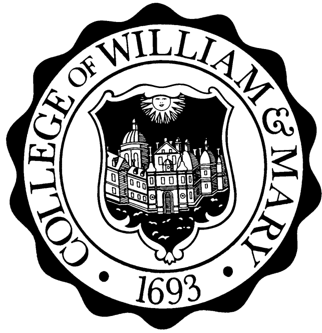

College of William & Mary
Department of Computer Science

WM-CS-2006-08

PRIMME: PReconditioned Iterative MultiMethod Eigensolver: Methods and software description

Andreas Stathopoulos

November 2006

PRIMME: PRECONDITIONED ITERATIVE MULTIMETHOD EIGENSOLVER: METHODS AND SOFTWARE DESCRIPTION *

ANDREAS STATHOPOULOS † AND JAMES R. MCCOMBS †

Abstract. This paper describes the PRIMME software package for the solving large, sparse Hermitian and real symmetric eigenvalue problems. The difficulty and importance of these problems have increased over the years, necessitating the use of preconditioning and near optimally converging iterative methods. On the other hand, the complexity of tuning or even using such methods has kept them outside the reach of many users. Responding to this problem, our goal was to develop a general purpose software that requires minimal or no tuning, yet it provides the best possible robustness and efficiency. PRIMME is a comprehensive package that brings state-of-the-art methods from "bleeding edge" to production, with a flexible, yet highly usable interface. We review the theory that gives rise to the near optimal methods GD+k and JDQMR, and present the various algorithms that constitute the basis of PRIMME. We also describe the software implementation, interface, and provide some sample experimental results.

1. Introduction. PRIMME, or PReconditioned Iterative Multi-Method Eigensolver, is a software package for the solution of large, sparse Hermitian and real symmetric eigenvalue problems. We view PRIMME as a significant step toward an "industrial strength" eigenvalue code for large, difficult eigenproblems, where it is not possible to factorize the matrix, and users can only apply the matrix operator, and possibly a preconditioning operator, on vectors.

If the matrix can be factorized, the shift-invert Lanczos code by Grimes, Lewis, and Simon has set a high standard for robustness [31]. However, even in factorizable cases, the factorization and back-substitutions can be very expensive, and a Lanczos method or a method that uses preconditioning can be more efficient, especially for extreme eigenvalues. On the other end, if only matrix-vector multiplication is available, the software ARPACK by Lehoucq, Sorensen, and Yang has set the standard for good quality code that is easy to use with very little parameter tuning [44]. Yet, the implicitly restarted Lanczos method [64], on which ARPACK is based, does not converge optimally, and it cannot directly use preconditioning, which is required for very difficult problems. The range of problems targeted by PRIMME is between the easy ones and the ones that must, and can be factorized. As problem sizes in applications continue to grow, so does PRIMME's target range.

Based on both research and integration, PRIMME's design philosophy is

1. to provide preconditioned eigenvalue methods that converge near optimally under limited memory
2. to provide the maximum robustness possible without matrix factorization,
3. to provide flexibility in mixing and matching among most currently known features,
4. to achieve efficiency at all architectural levels, and
5. to achieve all the above with a friendly user interface that requires no parameter setting from end-users, but allows full experimentation by experts.

This paper is organized as follows: In section 2 we describe the problem, its importance and difficulty, and discuss the advantages and shortcomings of other current eigenvalue software. In section 3 we present the main algorithmic framework for PRIMME, including the two near optimal methods, GD+k and JDQMR. We also discuss how a host of other algorithms can be parameterized within this framework. In section 4 we discuss how the PRIMME software meets its design goals. In section 5 we present sample comparisons with other state-of-the-art software. We conclude in section 6 with some discussion on on-going work and future extensions to PRIMME.

*Work supported partially by National Science Foundation grants: ITR/DMR 0325218, ITR/AP-0112727, ITR/ACS-0082094.

†Department of Computer Science, College of William and Mary, Williamsburg, Virginia 23187-8795, (andreas,mccombjr@cs.wm.edu).

PRIMME implements a multitude of features, algorithms, techniques, and heuristics that have emerged in research papers and software by this and other groups over many years. When their description is beyond the scope of this paper, we refer to the appropriate literature.

2. A difficult problem and current approaches. Given a real symmetric, or complex Hermitian matrix A of dimension N, we consider the problem of seeking $numEvals$ smallest, largest or interior eigenvalues λ_i, and their corresponding eigenvectors $\mathbf{x}_i$, $i = 1,\ldots,numEvals$. The numerical solution of this problem when A is large and sparse is one of the most computationally intensive tasks in a variety of applications.

One such application is structural engineering, where finite element models are used to perform vibrational and buckling analysis [31]. Electromagnetics is another area that depends on the solution of large, real symmetric eigenproblems [26, 37]. A particular demanding application comes from lattice Quantum Chromodynamics (QCD) where the pseudo-inverse of a very large Hermitian operator is approximated on the space of several of its smallest eigenpairs [22]. Recently, electronic structure applications from atomic scale physics [20] to molecular scale materials science [11] and nanotechnology, with symmetric and hermitian eigenproblems at their core, have been rivaling QCD as the top supercomputer cycle user.

The challenge is twofold; First, the matrix size, N, in these applications is routinely more than a million, while an order of a billion has also been tried [54]. Second, many applications, especially in electronic structure calculations, require the computation of hundreds or even thousands of extreme eigenpairs. Often the number of required eigenpairs, $numEvals$, is described as a small percentage of the problem size. In such cases, orthogonalization of $numEvals$ vectors, an $O(numEvals^2N)$ task, becomes $O(N^3)$, making the scaling to larger problem sizes practically infeasible.

Iterative methods are the only means of addressing these large problems. Yet, iterative methods may converge slowly, especially as the problem size grows. As with linear systems, preconditioning can be used to speed up convergence, but theoretical understanding of how it should be used in eigenvalue methods has only started to mature over the last decade [19, 40, 62]. *This probably explains the noticeable scarcity of high quality, general purpose software for preconditioned eigensolvers.* Beyond challenges in execution time, eigenvalue iterative methods must also store the iteration vectors for computing eigenvector approximations. With slow convergence, the storage demands of these applications can be staggering. Recently, iterative methods have been developed [62, 69, 41, 61, 52, 66, 68], that can use effectively the large arsenal of preconditioners for linear systems, and converge near optimally to an eigenpair under limited memory requirements.

2.1. In search of (near) optimal methods. The quest for optimality under limited memory is a natural one. In symmetric linear systems, Krylov methods such as Conjugate Gradient (CG) achieve optimal convergence in exact arithmetic through a three term recurrence. For eigenvalue problems, the Lanczos method can produce the optimal space through a three term recurrence, but the vectors must be stored, or recomputed. With preconditioning even these Lanczos properties do not hold. Restarting techniques can be employed so that approximations are obtained from a search space of limited size, see for example thick restarting [71] and its theoretically equivalent implicit restarting [64]. However this is at the expense of convergence. As we show later, these techniques coupled with locally optimal restarting can restore near optimal convergence when seeking one eigenpair.

When seeking many eigenpairs, it is an open question whether optimality can be achieved under limited memory. If one eigenvalue is known exactly, the corresponding eigenvector can be obtained optimally through a CG iteration [41, 66]. If $numEvals$ eigenvalues are known, one may think that the analogue optimality is to run $numEvals$ separate CG iterations. This is the approach taken by most limited memory, preconditioned eigensolvers for small $numEvals$

values. Yet, it is clearly suboptimal, because a method that stores all the CG iterates from each run would converge in much fewer iterations. For example, when the number of CG iterations is $O(N)$, the former approach takes $O(numEvals\, N)$, while an unrestarted Lanczos would take no more than N.

In our research, and in the development of PRIMME, we have focused on methods that do not allow their memory requirements to grow unbounded. With preconditioning this is the only realistic alternative. However, even without preconditioning, the attractive Lanczos method that does not store the iteration vectors has several drawbacks. To avoid spurious eigenvalues, expensive selective and/or partial orthogonalizations are required [56]. Without reorthogonalizations, as in the method of Cullum and Willoughby [13], the number of iterations may grow much larger than the optimal, because the method keeps recomputing copies of already converged eigenvalues. For highly ill conditioned problems, the method can be too slow. Moreover, the method needs to store a tridiagonal matrix of size equal to the number of iterations, which can be in the order of tens of thousands, and solve for *numEvals* of its eigenpairs. Finally, to obtain multiple eigenvalues, a locking or a block implementation of Lanczos must be implemented (see [12, 28], and [32] for extensive bibliography on block methods).

2.2. Current state of Hermitian eigenvalue software. Iterative methods have gained notoriety as very difficult to include in general purpose software. Dependence on special, user-defined data structures has long been resolved by the standardization of the basic linear algebra operations (through BLAS and LAPACK [43, 4]), and more flexible programming languages (e.g., C, C++) and interfaces. What remains, however, is that different problems may require different iterative solvers for robustness and/or efficiency, and often this is not known a-priori. Dependence on the preconditioner complicates matters further. Expert opinion is usually needed to tune the choice of methods and their parameters. This has led a group of experts to produce the popular series of “Templates for the solution of linear systems” [9], and “eigenvalue problems” [7]. Since then, the consensus on the relative merits of methods may not have changed for symmetric linear systems, but the area of symmetric eigenproblems has seen some remarkable progress [66, 68, 53, 1, 52, 26, 46, 30, 41, 6, 74].

This recent progress is reflected in the large number of codes for symmetric eigenproblems. In their most recent survey of eigenvalue codes, [33], Hernandez et al. list 20 eigenvalue codes that have become available since the year 1998. Eighteen of these are for symmetric eigenproblems. The good theoretical understanding of the Lanczos method is also reflected in this survey, with eight codes implementing various versions of Lanczos (block, thick restarted, indefinite, etc). In this paper we do not further discuss these Lanczos codes, because they are outside the scope of PRIMME’s target applications.

In the above survey, there are 12 codes that implement preconditioned eigensolver methods. We note that only one such code was publicly available before 1998; the Davidson code that one of the authors (Stathopoulos) developed in 1994 [67]. We do not consider this code further as it is superseded by PRIMME. From the rest 11 codes, EIGIFP [76] and JDCG [52] are written in Matlab, and although their methods are of interest, their implementation does not provide the robustness, flexibility, and efficiency required for general purpose software. Moreover, PySparse is a Python environment built on top of the JDBSYM code [25], so we only consider the underlying JDBSYM code. Finally, we do consider the SLEPc library, even though it does not currently support preconditioning, because the external software packages that it interfaces with (including PRIMME) support it. Therefore, besides PRIMME, the following 7 preconditioned symmetric eigensolver packages are currently available:
ANASAZI (2005, [73]), BLOPEX (2004, [38]), JDBSYM (1999, [25]), MPB (2003, [37]), PDACG (2000, [24]), SLEPc (2006, [34]), SPAM (2000, [60]).

ANASAZI is a well engineered package, with several features that enhance robustness and efficiency. ANASAZI implements a version of the LOBPCG method [41] and a block Davidson method (what we refer to as Generalized Davidson) for solving standard and generalized real symmetric and Hermitian eigenvalue problems. The reason for block methods is twofold: First, as an increased robustness measure, since Davidson methods have no problems identifying multiple eigenvalues [47]. Second, to take advantage of the increased cache locality in block matrix-vector, preconditioning, and BLAS operations. Although the total number of matrix-vector multiplications increases with larger block sizes, for appropriate block sizes this effect is usually balanced by better data locality [5]. ANASAZI is part of the Trilinos framework that includes highly optimized linear algebra operations, although some users may define their own matrix-multivector and preconditioner-multivector operations.

ANASAZI is still under development, but currently it does not include the near optimal GD+k or JDQMR methods. As we showed in [66, 68, 69] the difference in convergence over LOBPCG and (Generalized) Davidson can be substantial, if the preconditioner is not powerful enough to result in convergence in only a few iterations. Also, despite the very high quality implementation of ANASAZI, some users may be reluctant to use it because it is tightly coupled with the much larger Trilinos framework and its C++ object classes. For such users, PRIMME offers the alternative of a stand-alone design, that includes most ANASAZI features, while adding a choice of several near optimal methods.

BLOPEX is a software that implements the LOBPCG method for solving standard and generalized real symmetric eigenproblems. Hermitian eigenproblems are not supported yet. The code is written in C, and can be used both as stand-alone and as external package in PETSc and Hypre. The power of LOBPCG is that it combines the fast convergence of the three term locally optimal recurrence with a simple algorithm that requires no parameter tuning other than the block size [41, 69]. BLOPEX does not implement any other methods, and it cannot be used to find interior eigenpairs directly. Also, if the block size needs to be less than $numEvals$, either for memory reasons or efficiency, the user has to implement locking as a wrapper to BLOPEX. Finally, a robustness issue may arise in this particular implementation of LOBPCG, because it does not maintain an orthonormal basis for the search space [35, 68].

JDBSYM is a stand-alone software written in C that implements a block version of the Jacobi-Davidson (JD) method for solving standard and generalized real symmetric eigenproblems. Hermitian eigenproblems and parallelism are not supported. Before PRIMME, JDBSYM was the only JD implementation tailored to symmetric problems. JDBSYM finds eigenvalues near a shift σ, therefore it provides MINRES and SYMMLQ for solving the indefinite correction equation in JD, but also QMRs for using indefinite preconditioners. On the other hand, extreme eigenvalues are also found close to a σ, which should be selected carefully to avoid misconvergence to interior eigenvalues. JDBSYM is implemented efficiently but, as a classical Jacobi-Davidson method, its convergence depends on the interplay of inner and outer iterations which can be controlled through several, usually problem dependent, parameters, and a choice of three alternative correction equations. Nearly optimal dynamic stopping criteria as in [52] are not implemented.

MPB includes implementations of the original Davidson method and a Preconditioned CG minimization of the Rayleigh quotient for solving standard real symmetric or Hermitian eigenvalue problems. These eigenvalue codes are actually part of MPB, which is an Ab-Initio program for computing photonic band structures. Therefore, some of their functionality is tightly coupled with the application. MPB is written in C and supports parallelism.

PDACG is a parallel, Fortran 77 implementation of the deflation-accelerated CG method for optimizing the Rayleigh Quotient of standard and generalized real symmetric eigenvalue problems. Hermitian eigenproblems are not supported. The method is similar to other min-

imization approaches, e.g., in MPB, and uses locking to find more than one eigenvalues. It requires no parameter setting, but on the other hand it has limited functionality. For example, it is built around a provided parallel sparse matrix-vector multiplication routine with preset data structure, which depending on the user's needs, it may be either useful or difficult to use.

SLEPc is a library rather than an implementation of a single method. It is written in C and can be considered an extension to the popular PETSc toolkit [8]. This coupling with the larger PETSc package allows SLEPc to inherit a variety of tuned data structures, multivector operations, matrix-vector multiplication and preconditioning operators, but it cannot run as standalone with applications that do not use PETSc. SLEPc provides implementations of several basic and some more advanced methods for solving standard and generalized real symmetric and Hermitian eigenvalue problems. An interesting and useful feature is SLEPc's interface to several external packages, specifically: ARPACK, BLZPACK, TRLAN, BLOPEX, and PRIMME. Currently, SLEPc does not support preconditioning or finding interior eigenvalues, even if these functionalities are available in the underlying package (e.g., PRIMME). When these features are included, SLEPc can be a powerful testbed for experimenting with various eigenvalue software packages.

SPAM is a Fortran 90 code that solves standard, real symmetric eigenvalue problems. The underlying basic method is Davidson (Generalized Davidson), but the interesting algorithmic feature is that it solves the given problem through a sequence of hierarchically simpler problems. Simpler could mean sparser, coarser grids, smaller rank, etc. This algorithm, which is reminiscent of multigrid, can be very effective for certain problems, provided the user can provide the operator functions for the sequence of simpler problems. When this is not possible, the method reduces to simple Generalized Davidson.

In conclusion, we note that from the above codes the ones that come closest to a robust, efficient, general purpose code are the ANASAZI and SLEPc. Yet, neither of the two implement the methods we have shown to be nearly optimal, and neither implements a Jacobi-Davidson variant (although JD versions from PRIMME can be used through SLEPc). Moreover, SLEPc does not yet support preconditioning. Our goal in this software project has been to bring state-of-the-art methods from "bleeding edge" to production.

3. Developing robust, near optimal methods and software.

3.1. Newton approaches. We can view the eigenvalue problem as a constrained minimization problem, for minimizing the Rayleigh quotient $\mathbf{x}^T A\mathbf{x}$ on the unit sphere, or equivalently minimizing $\mathbf{x}^T A\mathbf{x}/\mathbf{x}^T\mathbf{x}$ [19]. For many eigenpairs, the same formulation applies for minimizing the trace of a block of vectors [59], working with *numEvals*-dimensional spaces [2]. As we discussed in [66, 68], most eigenmethods can be interpreted through the inexact Newton viewpoint or the quasi-Newton viewpoint. The following includes excerpts from these two papers.

3.1.1. The inexact Newton approach. The exact Newton method for eigenproblems can be applied on the Grassmann manifold (to enforce normalization of the eigenvectors), which is equivalent to classical Rayleigh Quotient Iteration (RQI) [19]. It is well known that when using an inner iterative method to solve the linear system at every step, converging beyond some level of accuracy, increases the overall number of matrix vector multiplications, and hence time. Inexact Newton methods attempt to balance good convergence of the outer Newton method with an inexact solution of the Hessian equation. However, when the linear system in RQI is solved to lower accuracy, RQI ceases to be equivalent to inexact Newton, and it may not converge. This has attracted a lot of attention in the literature [57, 42, 63, 29, 61].

A more appropriate representative of the inexact Newton minimization for the eigenvalue problem is the Jacobi-Davidson method [62]. Given an approximate eigenvector $\mathbf{u}^{(m)}$ and its

Ritz value $\theta^{(m)}$, the JD method obtains an approximation to the eigenvector error by solving approximately the correction equation:

$$(I-\mathbf{u}^{(m)}\mathbf{u}^{(m)T})(A-\eta I)(I-\mathbf{u}^{(m)}\mathbf{u}^{(m)T})\mathbf{t}^{(m)} = -\mathbf{r}^{(m)} = \theta^{(m)}\mathbf{u}^{(m)} - A\mathbf{u}^{(m)}, \tag{3.1}$$

where η is a shift close to the wanted eigenvalue. The next Newton iterate is then $\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} + \mathbf{t}^{(m)}$. There are two important differences from RQI. First, the pseudoinverse of the Hessian is considered to avoid the singularity when $\eta \approx \lambda$, and also to avoid yielding back $\mathbf{t}^{(m)} = \mathbf{u}^{(m)}$ when the equation is solved accurately with $\eta = \theta^{(m)}$. The latter problem could cause stagnation in the classical or the Generalized Davidson methods [15, 49, 70]. The second difference from RQI is that JD applies the pseudoinverse of the Hessian to the residual, which is the gradient of the Rayleigh quotient, not to $\mathbf{u}^{(m)}$. Some theoretical differences between Newton variants are discussed in [1]. Here, we focus on their computational ramifications.

The Generalized Davidson (GD) method obtains the next iterate as $\mathbf{t}^{(m)} = K^{-1}\mathbf{r}^{(m)}$, where the preconditioner K approximates $(A-\eta I)$. Although K can be thought of as an approximate solution to eq. (3.1), we follow prevalent nomenclature, and refer to GD as the application of a given preconditioner to the residual.

JD is typically used with subspace acceleration, where the iterates $\mathbf{t}^{(m)}$ are accumulated in a search space from which eigenvector approximations are extracted through Rayleigh-Ritz or some other projection technique [55, 48, 36]. This can be particularly beneficial, especially when looking for more than one eigenpair. Note, however, that without inner iterations, $\mathbf{t}^{(m)} = -\mathbf{r}^{(m)}$, and JD becomes subspace accelerated steepest descent, or equivalently the Lanczos method. With restarting and no inner iterations, JD is mathematically equivalent to Implicitly Restarted Lanczos.

The challenge in JD is to identify the optimal accuracy to solve each correction iteration. In [52], Notay proposed a dynamic stopping criterion based on monitoring the growing disparity in convergence rates between the eigenvalue residual and linear system residual of CG. The norm of the eigenresidual was monitored inexpensively through a scalar recurrence. In [66], we proposed JDQMR that extends JDCG by using symmetric QMR (QMRs) [23] as the inner method. The advantages are:

- the smooth convergence of QMRs allows for a set of robust and efficient stopping criteria for the inner iteration.
- it can handle indefinite correction equations. This is important when seeking interior or a large number of eigenvalues.
- QMRs, unlike MINRES, can use indefinite preconditioners, which are often needed for interior eigenproblems.

We also argued that JDQMR cannot converge more than three times slower than the optimal method, and usually it is significantly less than two times slower. Coupled with the very low QMRs costs, JDQMR has proved one the fastest and most robust methods for $numEvals = 1$.

When seeking many eigenvalues, the Newton method can be applied on the $numEvals$ dimensional Grassman manifold to compute directly the invariant subspace (see [59] and [2] for a more recent review). Practically, however, the Grassman RQI approach proposed in [2] is simply a block JD method [68]. The open computational question is how to solve $numEvals$ linear systems in block JD most efficiently, and whether to use a block method at all. Block methods that solve all the correction equations simultaneously do not consistently improve the overall runtime [26]. In our experience with block JDQMR and block JDBSYM, the single vector versions outperform their block counterparts both in execution time and matvecs. Unlike JDBSYM, however, the block JDQMR does not increase significantly the number of matvecs over the single vector version. This is because the more interior eigenvalues in the

block converge slower, and therefore their correction equations need to be solved less accurately than the more extreme ones. The dynamic stopping criteria of JDQMR realize this early, saving a lot of unnecessary matvecs.

Yet, a block size larger than one may be needed for robustness. Single vector JD methods may converge to the required eigenvalues out of order, and thus are prone to misconvergence. Moreover, in some occasions, a small block size was required for JDBSYM to converge in the presence of exact multiplicities. JDQMR did not exhibit this problem, but it may converge out of order. The alternatives are to ask for a few more eigenvalues than needed, or to use a small block size.

With large *numEvals*, however, the near optimal convergence of the JDQMR has to be repeated *numEvals* times; much like the *numEvals* independent CGs we mentioned in section 2.1. In this case, the role of a larger subspace acceleration is to obtain better approximations for nearby eigenpairs while JD converges to the targeted eigenpair. Although the convergence rate of QMR cannot improve further, increasing the basis size gives increasingly better initial guesses for the eigenpairs to be targeted next. For practical reasons, we avoid this continuum of choices and focus only on constant, limited memory basis sizes.

3.1.2. The quasi-Newton approach. An alternative to Newton is the use of the nonlinear Conjugate Gradient (NLCG) method on the Grassman manifold, which has given rise to many NLCG eigenmethod variants [19], including MPB and PDACG. However, it is natural to consider a method that minimizes the Rayleigh quotient on the whole space $\left\{\mathbf{u}^{(m-1)}, \mathbf{u}^{(m)}, \mathbf{r}^{(m)}\right\}$, instead of only along one search direction. The method:

$$\mathbf{u}^{(m+1)} = \text{RayleighRitz}\left(\{\mathbf{u}^{(m-1)}, \mathbf{u}^{(m)}, \mathbf{r}^{(m)}\}\right),\ m > 1, \tag{3.2}$$

is often called locally optimal Conjugate Gradient (LOCG) [18, 39], and seems to consistently outperform other NLCG type methods. For numerical stability, the basis can be kept orthonormal, or $\mathbf{u}^{(m)} - \tau^{(m)}\mathbf{u}^{(m-1)}$ can be used instead of $\mathbf{u}^{(m-1)}$, for some weight $\tau^{(m)}$. The latter, when used with multivectors $\mathbf{u}^{(m)}, \mathbf{r}^{(m)}$ is the LOBPCG method [41].

Because of the non-linearity of the eigenproblem, neither NLCG nor LOBPCG can be optimal. Quasi-Newton methods use the NLCG vector iterates to construct incrementally an approximation to the Hessian, and therefore they almost always converge faster than NLCG [27]. In the context of eigenvalue problems, if all the iterates of NLCG or LOBPCG are considered, certain forms of quasi-Newton methods are equivalent to unrestarted Lanczos. With thick or implicit restarting, however, IRL loses the single important direction ($\mathbf{u}^{(m-1)}$) that offers the excellent convergence to NLCG and LOBPCG. Therefore, the appropriate way to restart methods such as JD, GD and even Lanczos, would be by subspace acceleration of the LOBPCG recurrence. This was first observed in [50] for the Davidson method, although under a different viewpoint. In [69] we offered a theoretical justification of local optimality both for the Rayleigh quotient and the Ritz vector. We also provided an efficient implementation that combined this technique with thick restarting for the GD. In [66], we noted the connection of our method, which we call GD+k, to quasi-Newton and in particular to the limited memory BFGS method [51].

GD(m_{min}, m_{max})+k uses a basis of maximum size m_{max}. When m_{max} is reached, we compute the m_{min} smallest (or closest to a target value) Ritz values and their Ritz vectors, $\mathbf{u}_i^{(m)}, i = 1, m_{min}$, and also k of the corresponding Ritz vectors from step $m-1$: $\mathbf{u}_i^{(m-1)}, i = 1, k$. An orthonormal basis for this set of $m_{min} + k$ vectors, which can be computed in negligible time, becomes the restarted basis. A JD+k implementation is identical. If the GD/JD method is block, with block size b, it is advisable to keep k $\geq b$, to maintain good convergence for

TABLE 3.1
The meaning of the parameters used in Algorithms 3.1–3.4

$numEvals$	the number of required eigenvalues
m_{max}	the maximum basis size for V
m_{min}	the minimum restart size of the basis V
b	the maximum block size
k	the number of vectors from the previous step retained at restart
m	the current basis size for V
l	the number of locked eigenpairs
$numConv$	the number of vectors converged in the current block
q	the number of vectors converged since last restart
g_q	the number of initial guesses replacing locked eigenvectors
V	the basis $[\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m]$ for the search space
W	$W = AV$ array to save an extra matvec
X	on input any initial guesses, on output the eigenvectors $\mathbf{x}_i$

all block vectors. Note also that the special case of block GD(b, $3b$)+b is mathematically equivalent to LOBPCG with the same block size.

As we showed in [66], convergence of the GD+k is appreciably faster than LOBPCG for one eigenpair, even with small subspace acceleration, *and often indistinguishable from the optimal method.* For large $numEvals$ the convergence gains over LOBPCG are even larger [68]. Yet, higher iteration costs than JDQMR make it less competitive for very sparse operators. When seeking many eigenvalues, we have found block size $b = 1$ to always provide the smallest number of matrix-vector multiplications (and preconditioning operations), even with a small subspace acceleration. Apparently, convergence to an eigenvalue is so close to optimal that the synergy from other block vectors cannot improve the subspace acceleration benefits. This may also explain why slower methods tend to benefit more from a larger block size. Nevertheless, for both robustness and data locality reasons, general purpose software must implement methods with a block option.

3.2. The GD+k and the JDQMR algorithms in the PRIMME framework. In [66] we argued that most eigenvalue methods can be implemented using the basic iterative framework of GD. Algorithm 3.1 depicts a version of the block GD+k algorithm as implemented in PRIMME. For concise notation we use the abbreviations of Table 3.1 instead of the parameter names appearing in the software. The GD$(m_{min}, m_{max}) + k$ algorithm finds eigenpairs $(\lambda_i, \mathbf{x}_i)$ with smallest or largest eigenvalue, or closest to a set of user provided shifts. Vectors without subscripts are considered multivectors of variable block size between 1 and b. Vectors with subscripts are single vectors in the designated location of their array. The transposition symbol ^{T} denotes Hermitian transpose.

Although PRIMME can be used both with and without locking, Algorithm 3.1 presents only the locking option to avoid further complicated indexing and book-keeping. PRIMME includes a host of other features and handling of special cases, which are impractical to describe in one algorithm. Examples include special cases for basis size and block size, an algorithm that identifies Ritz pairs that although they cannot converge to full accuracy because of locking, they are practically converged (see [65]), an algorithm that repeats steps (4) to (20) until convergence is verified for all required pairs (see [47]), the handling of multiple user-defined shifts, and many others. Section 3.3 outlines some of them.

Algorithms 3.2–3.4 describe the PRIMME implementation of three important components of GD+k: convegence checking and forming the target block, restarting, and locking

ALGORITHM 3.1. *The Generalized Davidson(m_{min},m_{max})+k algorithm*

/* Initialization */
(1) Initial guesses are in X. Let $m = \min(m_{min}, size(X,2))$, $\mathbf{v}_{0:m} = X_{0:m}$
Build $(m_{min} - m)$ Lanczos vectors to get basis $V = [\mathbf{v}_0, ..., \mathbf{v}_{m_{min}-1}]$
(2) Set $W = AV$, $H = W^T V$, $m = nmv = m_{min}$, $q = l = 0$
(3) Compute eigendecomposition $H = S\Theta S^T$ with $\theta_0, \theta_2, \ldots, \theta_{m-1}$ sorted according to user defined criteria (smallest, largest, interior)
/* Repeat until convergence or max number of matvecs */
(4) **while** ($l < numEvals$ **and** $nmv < max_num_matvecs$)
  /* Repeat until basis reaches max size or it spans the whole space */
(5)   **while** ($m < m_{max}$ **and** $m < N - l$)
(6)   Reset b, if needed, so that $m + b \le m_{max}$
(7) $\mathbf{u}^{(m)} = V s_{0:b-1}$, $\theta^{(m)} = diag(\theta_{0:b-1})$,
    $\mathbf{w}^{(m)} = W s_{0:b-1}$, $\mathbf{r}^{(m)} = \mathbf{w}^{(m)} - \mathbf{u}^{(m)}\theta^{(m)}$
(8)   Check convergence and determine target block. See ALGORITHM 3.2
(9)   **if** ($l + numConv \ge numEvals$), **break**
(10)   Precondition the block of residuals: $\mathbf{t}^{(m)} = Prec(\mathbf{r}^{(m)})$
(11)   Orthonormalize $\mathbf{t}^{(m)}$ among themselves and against $\mathbf{v}_{0:m-1}$ and $\mathbf{x}_{0:l-1}$
(12)   Update $\mathbf{v}_{m:m+b-1} = \mathbf{t}^{(m)}$, $\mathbf{w}_{m:m+b-1} = A\mathbf{v}_{m:m+b-1}$, $nmv = nmv + b$
(13)   Update $H_{i,m:m+b-1} = \mathbf{v}_i^T \mathbf{w}_{m:m+b-1}$ for $i = 0, \ldots, m+b-1$
(14)   Remember Ritz vector coefficients: $s_i^{old} = s_i$, $i = 0, \ldots, \max(b,k) - 1$
(15) $m = m + b$
(16)   Compute eigendecomposition $H = S\Theta S^T$ with $\theta_0, \theta_2, \ldots, \theta_{m-1}$ sorted according to user defined criteria (smallest, largest, interior)
(17)   **end while**
(18)   Restart the basis and reset variables. See ALGORITHM 3.3
(19)   Lock the q flagged Ritz pairs into λ and X. See ALGORITHM 3.4
(20) **end while**

ALGORITHM 3.2. *Check convergence and determine target block*

(1) **repeat**
(2)   numConv = Number of converged vectors in block $\mathbf{r}^{(m)}$
  consider also practically converged vectors (see ALGORITHM 3.6)
(3) $q = q + numConv$. Flag these Ritz vectors as converged
(4)   Find the next numConv unconverged Ritz vectors $\mathbf{u}_i^{(m)}$ and their residuals $\mathbf{r}_i^{(m)}$ to replace the numConv ones in the block
(5)   Swap converged vectors after the block to maintain block locality
(6) **until** (b unconverged residuals are in the block **or** not enough available)
(7) **if** (not enough unconverged residuals) reduce block size

converged eigenpairs. The restart procedure in particular (Algorithm 3.3) combines thick (or dynamic thick) restarting [71], with the +k locally optimal restarting (steps (7) to (16)). Note that steps (7) to (12) apply on vectors of size m_{max}, and therefore the cost of the GD(m_{min},m_{max})+k implementation *is the same as that of the thick restarted* GD(m_{min}+k, m_{max}). In fact, GD+k is typically less expensive per iteration, because k=1 or $k \le b$ is not only sufficient [69, 66, 68], but also obviates the use of larger m_{min}.

What characterizes Algorithm 3.1 is its flexibility, It allows for complete freedom on

ALGORITHM 3.3. *The Restart procedure*
(1) *Decide the order in which to keep Ritz vectors (dynamic/thick restarting)*
(2) *Let (nx) the indices of unflagged Ritz vectors in the desired order*
Let (q, ix) the number and indices of flagged converged Ritz vectors
Let g the number of remaining initial guesses in X
/ Steps 3-6 guarantee m_{min} vectors are in the basis after locking */*
(3) $g_q = \min(q, g)$
(4) **if** $(g_q \geq m_{min})$ $m_u = \max(0, m_{min} - q)$
(5) **else** $m_u = \max(0, m_{min} - g_q)$
(6) *Consider the first m_u unconverged AND the q converged Ritz vectors*
These correspond to coefficient vectors: $[s_{nx(0)}, \ldots, s_{nx(m_u-1)}]$ and s_{ix}
/ Steps 7-16 include the coefficients of the previous step vectors */*
(7) *Orthonormalize the k Ritz vector coefficients from the previous step: s^{old} among themselves, against $[s_{nx(0)}, \ldots, s_{nx(m_u-1)}]$, and against s_{ix}*
(8) *Compute $H_{sub} = s^{oldT} H s^{old}$ ($k \times k$ submatrix)*
(9) *Compute eigendecomposition of $H_{sub} = Y \Phi Y^T$*
(10) *Set $s = [s_{nx(0)}, \ldots, s_{nx(m_u-1)}, \; s_0^{old}, \ldots, s_{k-1}^{old}, \; s_{ix}]$*
(11) *Set $\Theta = [\theta_{nx(0)}, \ldots, \theta_{nx(m_u-1)}, \; \phi_0, \ldots, \phi_{k-1}, \; \theta_{ix}]$*
(12) $m = m_u + k + q$
(13) $\mathbf{v}_i = V s_i$, $\mathbf{w}_i = W s_i$, $i = 0, \ldots, m-1$
(14) $H = 0$. *Then $H_{ii} = \theta_i$, for $i = 0 : m_u - 1$ and $i = m_u + k : m - 1$*
(15) $H(m_u : m_u + k - 1, \; m_u : m_u + k - 1) = H_{sub}$
(16) $s = I_m$. *Then $s(m_u : m_u + k - 1, \; m_u : m_u + k - 1) = Y$*

ALGORITHM 3.4. *The Locking procedure*
/ Called immediately after restart. Flagged vectors at the end of V */*
(1) *Recompute residuals $\mathbf{r}_i = \mathbf{w}_i - \mathbf{v}_i \theta_i$, $i = m - q : m - 1$ of flagged vectors*
(2) *Set (q, ix) to the number and index of flagged vectors remaining converged*
(3) *Flagged vectors that became unconverged stay in the basis V*
(4) $\lambda_{l:l+q-1} = \theta_{ix}^{(m)}$
(5) $g_q = \min(q, g)$, *update remaining initial guesses $g = g - g_q$*
(6) *Swap the next g_q initial guesses $X_{l:l+g_q-1}$ with the converged $\mathbf{v}_{ix(0:g_q-1)}$*
(7) *Lock the rest $X_{l+g_q:l+q-1} = \mathbf{v}_{ix(g_q:q-1)}$*
(8) $m = m - q + g_q$
(9) $l = l + q$
(10) *Orthonormalize new guesses among themselves and against V and $\mathbf{x}_{0:l-1}$*
(11) *Update $W_{m-g_q:m-1} = A v_{m-g_q:m-1}$, $nmv = nmv + g_q$*
(12) *Update $H_{i,m-g_q:m-1} = \mathbf{v}_i^T \mathbf{w}_{m-g_q:m-1}$ for $i = m - g_q, \ldots, m-1$*
(13) *Compute eigendecomposition $H = S \Theta S^T$ with $\theta_0, \theta_2, \ldots, \theta_{m-1}$ sorted according to user defined criteria (smallest, largest, interior)*
(14) *Reset Flags*

how to expand the space, how to extract approximations from it, and how to restart it. In PRIMME, all these choices are available by setting certain parameters. The price for this flexibility is that, at every step, it needs to compute eigenresiduals, orthogonalize new vectors against all current ones in the basis V, and maintain a work array for $W = AV$.

Step (10) is the one differentiating between most eigenmethods. When the algorithm re-

turns $\mathbf{t}^{(m)} = \mathbf{r}^{(m)}$ (and with k=0), it is equivalent to an expensive IRL implementation. However, if the matrix vector operation is expensive, the near optimal convergence of GD+k could be preferable over IRL. When the preconditioner is applied directly on the residual we have the classical GD and GD+k methods. By considering multivectors, the algorithm yields the equivalents of block (implicitly restarted) Lanczos [6, 56], subspace iteration (without preconditioning) [10], and preconditioned subspace iteration [7]. We also derive the classical block GD and the block GD+k methods [45, 67, 26, 69]. Of particular interest is the GD(b,3b)+b, which is a numerically stable implementation of the LOBPCG, maintaining full orthogonality of the basis for only 5% more floating point operations per iteration [69, 41, 66, 35]. In addition, b can be chosen independently from *numEvals*, to obtain locking implementations of LOBPCG. Variations such as GD(b, mb)+b are also plausible.

Step (10) can also return a JD correction vector. Without inner iterations, a preconditioner K can be inverted orthogonally to the space $Q = [X, \mathbf{u}^{(m)}]$ and applied to the residual. The pseudoinverse of such a preconditioner can be written as:

$$\left((I - QQ^T)K(I - QQ^T)\right)^+ = (I - K^{-1}Q(Q^TK^{-1}Q)^{-1}Q^T)K^{-1}(I - QQ^T) \tag{3.3}$$

$$= K^{-1}(I - Q(Q^TK^{-1}Q)^{-1}Q^TK^{-1})(I - QQ^T). \tag{3.4}$$

The above is known as Olsen's method [54]. "Robust shifting" [70] can be used as an approximation to Olsen's method to avoid the computation of the pseudoinverse. This applies the preconditioner on $\mathbf{r}^{(m)} + \delta\theta\, \mathbf{u}^{(m)}$, where $\delta\theta$ is an approximation to the eigenvalue error.

When the preconditioner (3.4) is used in an iterative method on eq. (3.1), we obtain the classical inner-outer JD variants. In [21, 62, 7] it is shown that JD methods can be implemented with one projection with Q per iteration. If the inner iteration solves eq. (3.1) accurately, we obtain a subspace accelerated Inverse Iteration (for a given η) or RQI (for $\eta = \theta^{(m)}$). The true flexibility of JD is that it converges even when eq. (3.1) is solved approximately.

Let $A_{\eta,u^{(m)}}$ denote the projected matrix operator in the correction eq. (3.1), and $K_{u^{(m)}}$ the projected preconditioner. Our JDQMR algorithm uses the GD+k as the underlying outer method, and at step (10) calls the symmetric, right preconditioned QMR with $A_{\eta,u^{(m)}}$, $K_{u^{(m)}}$, and right hand side $-\mathbf{r}^{(m)}$. Algorithm 3.5 shows our QMRs algorithm. The scalar recurrences for monitoring the eigenvalue residual, and the dynamic stopping criteria are shown at steps numbered with decimal points (see [66] for detailed analysis).

3.3. Other special techniques in PRIMME. The above state-of-the-art algorithms and the myriad of their combinations are complemented by several other techniques that provide additional efficiency and robustness.

3.3.1. Avoiding the JD oblique projectors. For large *numEvals*, the classical JD requires significant extra storage for $K^{-1}Q$ to avoid doubling the number of preconditioning operations. In [68] we have shown that the pseudoinverse (3.4) with only $\mathbf{u}^{(m)}$, not X, is sufficient. Intuitively, projecting out $\mathbf{u}^{(m)}$ from a very accurate preconditioner helps avoid the classical Davidson problems where the correction is almost completely in the direction of $\mathbf{u}^{(m)}$ [53]. However, projecting out X does not serve the same purpose. Instead, one would hope that it produces a better conditioned correction equation. Our analysis in [68] showed that usually there is no significant difference between the condition numbers, and often the correction equation without the pseudoinverse is better!

Avoiding the pseudoinverse with X yields significant storage savings, effectively halving the memory required by the JD method for large *numEvals*. PRIMME follows this strategy as a default, but it also implements all possible combinations of different projector and pseudoinverse strategies, for both $\mathbf{u}^{(m)}$ and X. Assume that the JD projectors for $\mathbf{u}^{(m)}$ are included

ALGORITHM 3.5. *Symmetric QMR for JDQMR with adaptive stopping criteria*

Input: $A_{\eta,u^{(m)}}, K_{u^{(m)}}, -\mathbf{r}^{(m)}, maxiter$

Output: t_k

(1) $\mathbf{t}_0 = 0, \delta_0 = 0, \mathbf{r}_0 = -\mathbf{r}^{(m)}, \mathbf{d}_0 = K^{-1}_{u^{(m)}}\mathbf{r}_0$

(2) $g_0 = \|\mathbf{r}_0\|, \Theta_0 = 0, \rho_0 = \mathbf{r}_0^T\mathbf{d}_0$

(2.0) $\mathrm{B}_0 = \Delta_0 = \Gamma_0 = \Phi_0 = \Psi_0 = 0$

(3) **if** $(maxiter = 0)$, $\mathbf{t}_0 = \mathbf{d}_0$, **return**

(4) **for** $k = 1, \ldots, maxiter$

(5) $\mathbf{w} = A_{\eta,u^{(m)}}\mathbf{d}_{k-1}$

(6) $\sigma_{k-1} = \mathbf{d}_{k-1}^T\mathbf{w}$, **if** $(\sigma_{k-1} = 0)$, **return**

(7) $\alpha_{k-1} = \frac{\rho_{k-1}}{\sigma_{k-1}}$

(8) $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1}\mathbf{w}$

(9) $\Theta_k = \frac{\|\mathbf{r}_k\|}{g_{k-1}}, c_k = \frac{1}{\sqrt{1+\Theta_k^2}}, g_k = g_{k-1}\Theta_k c_k$

(10) $\delta_k = (c_k^2\Theta_{k-1}^2)\delta_{k-1} + (c_k^2\alpha_{k-1})\mathbf{d}_{k-1}$

(11) $\mathbf{t}_k = \mathbf{t}_{k-1} + \delta_k$

(12) **if** $(\rho_{k-1} = 0)$, **return**

(12.0) $\gamma_k = c_k^2\Theta_{k-1}^2,\ \xi_k = c_k^2\alpha_{k-1}, f = 1 + \|\mathbf{t}_k\|^2$

(12.1) $\Psi_k = \gamma_k(\Psi_{k-1} + \Phi_{k-1})$

(12.2) $\Phi_k = \gamma_k^2\Phi_{k-1} + \xi_k^2\sigma_{k-1}$

(12.3) $\Gamma_k = \Gamma_{k-1} + 2\Psi_k + \Phi_k$

(12.4) $\Delta_k = \gamma_k\Delta_{k-1} - \xi_k\rho_{k-1}$

(12.5) $\mathrm{B}_k = \mathrm{B}_{k-1} + \Delta_k$

(12.6) $p = \left(\theta^{(m)} - \eta + 2\mathrm{B}_k + \Gamma_k\right)/f$

(12.7) $\theta_k^{(m+1)} = \eta + p$

(12.8) $p_k = (\theta^{(m)} - \eta + \mathrm{B}_k)^2/f - p^2$

(12.9) $r_k^{(m+1)} = \sqrt{g_k^2/f + p_k}$

(12.10) **if** *(*$r_k^{(m+1)}$ *not real),* $r_k^{(m+1)} = \sqrt{g_k^2/f}$

(12.11) **if** $\Big(g_k \le r_k^{(m+1)} \max(0.99\sqrt{f}, \sqrt{g_k/g_{k-1}})$ **or** *(* $\theta_k^{(m+1)} > \theta_{k-1}^{(m+1)}$*)*
or *(*$r_k^{(m+1)} < 0.1 r_0$*)* **or** *(* $g_k < \varepsilon_{inn}$ *)* **or** $(r_k^{(m+1)} < \varepsilon_{inn})\Big)$
then return *the correction* $\mathbf{t}_k$.

(13) $\mathbf{w} = K^{-1}_{u^{(m)}}\mathbf{r}_k, \rho_k = \mathbf{r}_k^T\mathbf{w}, \beta_k = \frac{\rho_k}{\rho_{k-1}}$

(14) $\mathbf{d}_k = \mathbf{w} + \beta_k\mathbf{d}_{k-1}$

(15) **end for**

in the notation of A and K. Define the orthogonal projector $P = I - QQ^T$, and for any matrix B the skew projector:

$$P_B = (I - BQ(Q^TBQ)^{-1}Q^T), \tag{3.5}$$

and note that the correction equation preconditioned with (3.4) can be written as:

$$PAP(PKP)^+ = PAP_{K^{-1}}K^{-1} \tag{3.6}$$

$$= PAK^{-1}P^T_{K^{-1}}. \tag{3.7}$$

Table 3.2 summarizes several variants of a projected operator based on whether we operate with a projector on the left, and/or on the right of A, and whether we relax the requirement for

TABLE 3.2
Projection alternatives to the classical Jacobi-Davidson correction equation (with right preconditioning). The 0/1 string characterizes whether there is a projection on the left of A, on the right of A, and whether the right projection is skew projection or not. Theoretically Jacobi-Davidson corresponds to (111) although it is typically implemented as (011). All options are available in PRIMME, with (100) the default for preconditioning, and (000) the default for unpreconditioned cases.

(Left Skew Right)	Operator	(Left Skew Right)	Operator
111	$PAP_{K^{-1}}K^{-1}$	011	$AP_{K^{-1}}K^{-1}$
101	$PAPK^{-1}$	001	APK^{-1}
100	PAK^{-1}	000	AK^{-1}

a right skew projector, replacing it with P. For example, the case $PAP_{K^{-1}}K^{-1} = PAK^{-1}P^T_{K^{-1}}$, which is equivalent to notation (111), requires two projections per QMR step. The case $AP_{K^{-1}}K^{-1} = AK^{-1}P^T_{K^{-1}}$, or equivalently (011), is the proposed implementation by the JD authors and includes the skew projection with X. Our default strategy with preconditioning is the (100), or equivalently PAK^{-1} where there is no skew projection, and only one, orthogonal projection on the left.

A particular impressive outcome of the above flexibility is possible when K has the same eigenvectors as A (e.g., if K is a polynomial of A, or simply $K = I$). In that case, all the QMR iterates stay in $X^\perp$ invariant space, without any orthogonalization! Floating point arithmetic, and the fact that X are converged to tol, not to machine precision, will eventually introduce X components that QMR will have to remove by additional iterations. However, a few additional iterations is a small price to pay for removing the limiting scalability factor $O(numEvals^2N)$ of orthogonalization. In our experience, unpreconditioned JDQMR-000 achieves an almost linear scaling with *numEvals*, both in convergence and in time, which is as close to optimality as possible under limited memory.

3.3.2. Avoiding stagnation because of locking. Locking is a stable form of deflation, where an eigenvector **x** is removed from the search space of an eigensolver and all subsequent vector operations are performed orthogonally to **x**. Without locking, converged eigenvectors are kept in the search space and improve with time. Locking usually provides a better mechanism than non-locking for identifying eigenvalues that are highly clustered or of very high multiplicity [47].

However, locking introduces a subtle numerical, but not floating point, problem. Specifically, a large number of locked, approximate eigenvectors, that have converged to tol residual accuracy, may impede convergence to tol accuracy for some subsequent eigenvector. This problem is described in our recent report [65]. Before that report, occurrences of the problem have been mainly anecdotal, and not well documented. Yet, many practitioners were well aware of the problem, but had no good solution to it, other than to stop the method, perform a subspace projection with all converged vectors, and then continue with new initial guesses and the already computed eigenvectors.

The problem is rather rare and it tends to surface when hundreds of eigenpairs are computed, but its existence undermines the reliability of any numerical software that implements locking. The resulting stagnation, which must be differentiated from slow convergence, is not an option in critical applications. In [65] we have provided an algorithm that identifies the problem when it occurs. Its variant as implemented in PRIMME appears in Algorithm 3.6. The interesting theoretical result is that a "practically converged" eigenvector can still be locked, because enough of its missing components are in X, so that a single Rayleigh Ritz projection at the end will produce a Ritz vector with residual norm less than the required tolerance.

ALGORITHM 3.6. *Identify a locking stagnation problem*
l is the number of locked vectors
tol is the convergence tolerance for residual norms
$mxTol$ is the maximum residual norm of any locked eigenvector
E is the guaranteed attainable tolerance without stagnation

*In step (2) of the Convergence procedure (*ALGORITHM *3.2) include:*
    Set $E = \sqrt{l}\, mxTol$
    if $(\|\mathbf{r}^{(m)}\| < tol)$ *Flag* $\mathbf{u}^{(m)}$ *as converged to be locked,* **break**
    if $(\|\mathbf{r}^{(m)}\| < E)$
        Compute $\|\mathbf{r}_d\| = \|(I - XX^T)\mathbf{r}^{(m)}\|$, $\beta = \sqrt{\|\mathbf{r}^{(m)}\|^2 - \|\mathbf{r}_d\|^2}$
        if $\left(\beta > tol \textbf{ and } \|\mathbf{r}_d\| < tol^2/(2\|\mathbf{r}^{(m)}\|)\right)$
            Flag $\mathbf{u}^{(m)}$ *as "practically converged" to be locked*
        endif
    endif

*In step (2) of the the Locking Procedure (*ALGORITHM *3.4) include:*
    Check if a recomputed norm remains "practically converged"
    if $(\|\mathbf{r}^{(m)}\| \geq E)$ *Unflag* $\mathbf{u}^{(m)}$*. It has become unconverged again*
    elseif $(\|\mathbf{r}^{(m)}\| < tol)$ *Lock* $\mathbf{u}^{(m)}$ *as converged*
    else
        Lock $\mathbf{u}^{(m)}$ *as "practically converged"*
        Update $mxTol = \max(mxTol, \|\mathbf{r}^{(m)}\|)$
    endif

3.3.3. Dynamic method selection. Many users commonly invoke the complexity of tuning the parameters of the JD method as the main reason for choosing an alternative method. The self-tuned inner-outer iteration of JDQMR has all but removed such reasons. The other important remaining choices of block size and basis size are common to most other methods. More importantly, both GD+k and JDQMR display remarkable robustness for a wide variety of choices for these parameters.

One choice remains, however; the choice between GD+k and JDQMR. As we mentioned before, GD+k converges almost identically to the optimal method, while JDQMR may repeat some subspace information in the QMRs of different outer steps. In our extensive experience, JDQMR is usually between 1.1 and 1.7 times slower than optimal[1] On the other hand, the cost per iteration of the JDQMR is significantly lower than the GD+k one. The crossover point between the two methods depends on the expense of the matrix and the preconditioner operators, on *numEvals*, and on the slowdown of the JDQMR convergence relatively to GD+k.

In [68], besides an asymptotic comparative analysis, we provided cost models for the time complexity of GD+k and JDQMR as a function of certain parameters and procedural components, rather than the traditional flop counts. Such components include the matrix-vector and preconditioning operations, the outer GD+k iteration, which is common to all methods, and the inner QMRs iteration. The parameters are the number of inner/outer iterations, and the convergence slowdown experienced by JDQMR. This approach allows a dynamic prediction of the execution time of GD+k and JDQMR based on runtime measure-

[1] The actual convergence is usually closer to optimal, but QMRs, like most iterative methods for linear systems, takes one more matvec before it exits the inner iteration, which can add up if only a few inner iterations are required.

ments of the parameters and the cost of the components.

It is beyond the current scope to describe the exact averaging we use over successive iterations to update the measured statistics. Instead, we outline in Algorithm 3.7 and motivate the extensions needed to Algorithm 3.1 to achieve a dynamic method selection between GD+k and JDQMR. For this, the following problems have to be addressed.

First, by running solely with GD+k we cannot measure the cost or predict the convergence of JDQMR. By running solely with JDQMR, we can still update the GD+k cost, but not its convergence. Therefore, at least one switch between the two methods is necessary. Because initially the search space may not contain good eigenpair approximations, which is important for Newton type methods, we start with GD+k and after a certain time we force a switch to JDQMR so that statistics for both methods are obtained.

Second, because convergence rates and even the runtime cost of various components may change since they were last measured, more than one switch may be necessary. Deciding on the frequency of switching depends on $numEvals$. If $numEvals$ is large, we can afford to have each method converge alone to one eigenpair, and thus collect better convergence statistics to evaluate what method to use for the following eigenpair. For small $numEvals$, methods must be evaluated much more frequently, because the software should adapt quickly to avoid solving almost the whole problem with the wrong method.

Third, the two methods need to be evaluated at different points. For GD+k a reasonable evaluation point is at every restart. At that point, the method has completed a full cycle, so all components have been measured, and $m_{max} - m_{min}$ iterations provide a good update for estimating its convergence rate. JDQMR, however, should not be evaluated at restarts, because it only takes a small number of outer iterations (often less than $m_{max} - m_{min}$) to converge to an eigenpair. Moreover, JDQMR may perform a large number of inner iterations. If it is clear that JDQMR should not be used further, e.g., because of a very expensive matrix operator, our dynamic algorithm should not solve another correction equation. Therefore, we must evaluate JDQMR at every outer step, just before the call to the correction equation.

Fourth, regardless of evaluation points and $numEvals$, if some eigenpairs converged during the current outer iteration, the algorithm has to update the convergence statistics. Finally, before exiting, PRIMME can use the obtained statistics to recommend a method to the user, in case problems similar to the current one need to be solved repeatedly. The algorithm that summarizes these decisions for dynamic method switching is shown in Algorithm 3.7.

We have observed that this dynamic, completely automated meta-method runs usually within 5% of the fastest of GD+k and JDQMR. More surprising was that in certain cases where JDQMR was the fastest method, the dynamic method improved the JDQMR timing! This is because it has the freedom to switch between GD+k and JDQMR when this is beneficial. The method may choose GD+k during the early stages of convergence when JDQMR takes too few inner iterations and switch later. Similarly, for large $numEvals$, GD+k could be preferable up to a certain number of eigenvalues, beyond which JDQMR should be used. Finally, we note that our dynamic method responds even to external, system load changes.

3.3.4. Orthogonalization stability and efficiency. Orthogonalization is the single most important component of an eigenvalue iterative solver. If there is orthogonality loss in the V basis, methods cannot converge to the required accuracy, and they may even stagnate or produce "eigenpairs" that do not exist. PRIMME currently uses a variation of the classical Gram-Schmidt with iterative reorthogonalization [14]. After two iterations, if an additional orthogonalization is needed, we determine that the vector has lost all significant components of the original direction and we replace it with a random vector. The procedure also guards against vectors that are zero or close to machine precision.

In parallel computers, one of the factors limiting scalability is the presence of several

ALGORITHM 3.7. *Basic algorithm for Dynamic Method Choice*

```
When dynamicMethod = 1, 3, current method is GD+k
When dynamicMethod = 2, 4, current method is JDQMR
For numEvals < 5, we alternate between 1, 2,
        evaluating GD+k every restart, and JDQMR every outer step or
        when an eigenpair converges
For numEvals ≥ 5, we alternate between 3, 4,
        evaluating GD+k and JDQMR only when an eigenpair converges
```

Extensions to ALGORITHM *3.1*

```
(3.1)  if (dynamicMethod > 0)
          initializeModel(CostModel)
          /* Start always with GD+k. Switch to JDQMR later: */
          if (numEvals < 5)
             dynamicMethod = 1; /* switch to 2 at first restart */
          else
             dynamicMethod = 3; /* switch to 4 after first pair converges */
       endif

(9.1)  if (dynamicMethod > 0)
          Measure and accumulate time spent in correction equation
          /* if some pairs converged OR we evaluate jdqmr at every step */
          if (numConv > 0 or dynamicMethod = 2)
             /* update convergence statistics and consider switching */
             Update_statistics(CostModel)
             switch (dynamicMethod)
                case 1: break /* for few evals evaluate GD+k only at restart */
                case 3: Switch_from_GDpk(CostModel); break;
                case 2: case 4: Switch_from_JDQMR(CostModel);
             end switch
          endif
       endif

(19.1) if (dynamicMethod = 1 )
          Measure outer iteration costs
          Update_statistics(CostModel)
          Switch_from_GDpk(CostModel)
       endif

(20.1) if (dynamicMethod > 0 )
          ratio = ratio of estimated overall times for JDQMR over GD+k
          if (ratio < 0.96) For this problem recommend method: JDQMR
          else if (ratio > 1.04) For this problem recommend method: GD+k
          else Ratio is too close to 1. Recommend method: DYNAMIC
```

dot products in Gram-Schmidt. We have implemented a not so well known strategy that typically removes one dot product per Gram-Schmidt iteration. Let $\mathbf{r}$ be the vector to be orthogonalized against V, and $s_0 = \|\mathbf{r}\|$. After orthogonalization $\mathbf{r}' = (I - VV^T)\mathbf{r}$, the test $s_1 = \|r'\| < 0.7071\ s_0$ determines whether to reorthogonalize. It is possible to avoid the expense of the dot product to compute the norm s_1, and the synchronization that it implies. Note that:

$$s_1^2 = \|(I - VV^T)\mathbf{r}\|^2 = (\mathbf{r}^T - (\mathbf{r}^T V)V^T)^T(\mathbf{r} - V(V^T\mathbf{r})) = s_0^2 - (V^T\mathbf{r})^T(V^T\mathbf{r}).$$

The $(V^T\mathbf{r})$ is a byproduct of the orthogonalization, and because it is a small vector of size m, not N, all processors can compute $(V^T\mathbf{r})^T(V^T\mathbf{r})$ locally, and inexpensively. If the test $s_1 < 0.7071 s_0$ is not satisfied, the resulting vector can be normalized, $\mathbf{r}'/s_1$, and the process exits. Otherwise, we set $s_0 = s_1$ and reorthogonalize.

This process hides a numerical danger; s_1 may be computed inaccurately if V and $\mathbf{r}$ are almost linearly dependent. Although the normality of $\mathbf{r}'$ is not important at this stage, this can cause the reorthogonalization test to fail, and return a vector that is neither normal nor orthogonal. An error analysis of the computation provides the following interesting result:

$$\frac{|s_1 - \hat{s_1}|}{|s_1|} = O\left((\frac{s_0}{s_1})^2\ \varepsilon_{machine} \right),$$

where $\hat{s_1}$ is the s_1 as computed by our algorithm. This result suggests that our algorithm is safe to use, with no loss of digits in s_1, if $s_1 > s_0\ \sqrt{\varepsilon_{machine}}$. If this test is not satisfied, our algorithm computes explicitly the norm $s_1 = \|r'\|$, and continues. Experiments with many ill-conditioned sets of vectors have confirmed the theoretical results and the robustness, as well as improved efficiency, of the resulting algorithm.

3.3.5. A verification iteration. PRIMME can be used without locking, when the number of required eigenvectors fit in the basis: $m_{max} > numEvals \geq m_{min}$. Converged eigenvectors remain in V, but are flagged as converged, so that they are not further placed in the target block. Still, they participate in the Rayleigh Ritz at every step, and therefore they improve as additional information is gathered in V. For the same reason, however, it is possible that a Ritz vector $\mathbf{x}_i$ that was flagged converged, it becomes unconverged during later iterations. This could occur if eigenpairs converge out of order, or if they have very high multiplicities.

PRIMME implements an outer verification loop that includes steps (2) through (20) of Algorithm 3.1. Before exiting, PRIMME verifies that all flagged Ritz vectors satisfy the convergence tolerance. If they do not, the basis V is orthonormalized, $W = AV$ is recomputed, H is rebuilt, all flags are reset, and the algorithm starts again trying to find all eigenpairs. Usually a small number of outer iterations is enough to recover the small deficiencies that have caused some eigenvectors to become unconverged. This verification is repeated until all required eigenvectors converge.

3.4. Memory requirements. The memory requirements for the underlying GD, JD, and symmetric QMR methods are established in the literature [9, 7]. The way that these basic methods are combined in PRIMME through various parameter choices determines the actual method and its memory requirements. PRIMME requires that the user provides three arrays where the computed eigenvalues, eigenvectors, and their residual norms will be placed. Without locking, and for certain method choices that do not use the expensive skew JD projections, the user may set the eigenvector array to be at the start of the work array in the primme data structure. This saves the additional $(numEvals\ N)$ storage of the eigenvector array. For general users, we do not yet recommend this undocumented feature. We focus now on the actual requirements of the PRIMME software.

Because GD+k is implemented as a thicker restarted GD, the two methods require practically the same memory footprint, as outlined in the table below:

$2m_{max}N$	storage for V and W
$2m_{max}^2$	storage for H and S
$m_{max}k$	storage for the s^{old}
$\max(k^2, m_{max}k,$ $2b(numEvals + m_{max}))$	general storage shared by various components

The above storage is clearly dominated by $2m_{max}N$ for the arrays V and W. With the use of locking, m_{max} can be kept small (e.g., 10–15 for extreme eigenvalues), but we can still compute a large number of eigenpairs.

The basic QMR method requires storage for five long vectors ($5N$) and some small work space of $2numEvals + m_{max} + 2b$. JDQMR may require additional storage if skew projectors with the preconditioner are required (see section 3.3). Specifically, skew projection only on $\mathbf{u}^{(m)}$ (method JDQMR-100) requires one additional vector (N), while the skew projector on both $\mathbf{u}^{(m)}$ and X requires storage for $N + (numEvals + m_{max})(N + numEvals + m_{max})$. The latter can be a limiting factor for applications that seek hundreds or thousands of eigenvectors, and we do not recommend it. Considering also the expense of the GD+k outer iteration, our default JDQMR-000 and JDQMR-100 methods require storage for $O((2m_{max} + 5)N)$ and $O((2m_{max} + 6)N)$ elements, respectively.

Storage for other methods is derived from the above. For example, Rayleigh Quotient Iteration without subspace acceleration, requires $m_{max} = 2$ and if we use QMRs without the skew projectors the total memory is $O(9N)$. Similarly, implementing LOBPCG as GD(b,3b)+b, the memory requirements are: $O(6bN)$.

3.5. Computational complexity. The complexity of any method in PRIMME can be decomposed to the complexity of the outer iteration component, plus the complexity of the inner iteration component. The only additional parameter is the relative frequency that a method spends in each component. For completeness, we present a model based on floating point operation counts for both GD+k and JDQMR. For a detailed complexity analysis see [66, 68]. Following classic literature, each of the GD steps requires the following flops:

reorthogonalization	$O(8Nb(m+l) + 2b^2N + 2bN)$	(step 11)
updating of H	$O(2mbN)$	(step 13)
the small eigenproblem	$O(4/3m^3)$	(step 16)
Ritz vector computation	$O(2mbN)$	(step 7)
residual computation	$O(2mbN + 4bN)$	(step 7)
norm computation	$O(2bN)$	(step 8)
restarting cost	$O(4Nm_{max}m_{min})$	(steps 18–19)

Averaging over $m = m_{min} \dots m_{max}$ and setting $\mu = m_{min}/m_{max}$, we have the average cost per outer step:

$$\text{GD cost} = O\left(\frac{7+4\mu-7\mu^2}{1-\mu}Nm_{max}b + \frac{11+2b+\mu}{1-\mu}Nb + 8Nbl + 1/3m_{max}^3\right).$$

With typical values, $m_{min} = m_{max}/3$ and $b = 1$ we have:

$$\text{GD cost} = 11.3Nm_{max} + 20N + 8Nl + m_{max}^3/3.$$

With $b = 1$, these are mostly BLAS level 2 and some BLAS level 1 operations. With $b > 1$, PRIMME uses mainly BLAS level 3 operations. Note also that when the number of locked vectors $l > 22$, the orthogonalization starts to dominate the iteration costs.

Currently, the inner QMRs solves the correction equations for each block vector independently. Based on this, it suffices to obtain the cost for each QMR step, that includes also two projections with $\mathbf{u}^{(m)}$:

$$\begin{aligned} &\text{QMR cost} = 25N + 4Nl \qquad \text{for JDQMR-100} \\ &\text{QMR cost} = 25N \qquad\qquad\;\; \text{for JDQMR-000.} \end{aligned}$$

Note that Algorithm 3.5 shows $24N$ operations, but includes non traditional vector updates, which when implemented as BLAS level 1 routines, give $26N$. We have managed to reduce it to $25N$ by alternating between buffers for d and w. The projectors against X are BLAS level 2 operations. Everything else is strictly BLAS level 1 operations. A similar implementation of JDCG as described in [52] would cost $24N$ operations, but the slightly additional cost of JDQMR is justified by the increased robustness, general purpose flexibility, and ability to derive better stopping criteria.

4. The PRIMME software. Our target is to produce an eigenvalue code as close as possible to "industrial strength" standards. To this end, our design philosophy as outlined in the introduction, consists of three components; the algorithmic, the implementation, and the user-interface. In the previous sections, we have described a long list of methods, techniques, and specialized algorithms that have been implemented in PRIMME. These address (1) what are the near optimal methods under limited memory that a state-of-the-art eigensolver should implement, (2) how to employ certain techniques to enhance robustness (block methods, verification, avoiding locking stagnation, etc), (3) how these can be combined in a unified framework. In this section we address the remaining issues, in particular implementation efficiency, rich functionality, and a flexible but usable user interface.

4.1. Choice of language and implementation. The underlying ideas for the basic structure of PRIMME have evolved starting from the 1994 Fortran 77 code DVDSON (or ACPZ) [67], which has been popular in the physics community, and are loosely based on our early Fortran version of GD+k/Jacobi-Davidson, DJADA, which we circulated in 2000. At that point, we set the goal of developing a general purpose, robust, and state-of-the-art eigensolver. Our design philosophy suggested that a project of this proportion must be engineered around a more flexible language. We have chosen the C language.

In the past, Fortran users claimed, not often without merit, that C compilers were not optimizing numerical code as efficiently as Fortran compilers. In the last ten years there is significant improvement, not only on the quality of optimization of C compilers, but also in the way programmers have learned to program numerical methods in C. Nowadays, properly written C codes run as efficiently as their Fortran versions. However, efficiency was a secondary reason for choosing C. The brunt of computation in PRIMME is handled by calls to BLAS and LAPACK functions, which are usually in Fortran or hand tuned in assembly.

Our primary reasons for choosing C are: the flexibility it allows for the user interface and parameter passing, its interoperability, as well as its cleaner memory management. A PRIMME type structure, could contain all the required information, such as function pointers to the matrix-vector multiplication and preconditioning operators, pointers to arbitrary data the user would like to pass to these operators, pointers to work space that may be already available, as well as a wide range of parameters. A judicious setting of defaults within PRIMME presents an uncluttered interface to the user. A similar functionality could be achieved from Fortran, but only through a more involved reverse communication interface. C is the most commonly used programming language for systems programming, which gives it a status of "lingua franca" among other languages. C interoperates easily with C++, Fortran 77, Fortran

90, Python, and many other scripting languages and environments (e.g., Matlab, Mathematica, etc), and thus could help PRIMME achieve a broader impact in the community. Finally, we have opted not to use the larger, more complicated C++ language, which would be a better choice if PRIMME were tightly coupled with a bigger problem solving environment, not a stand alone, general purpose package.

On the technical side of the implementation, memory for PRIMME workspace can be allocated internally, if the user does not provide enough workspace. Because most workspace in PRIMME is needed throughout the execution of the program, there is no point in allocating and freeing it in different functions. Therefore, we allocate all required memory as one chunk in the beginning of the algorithm, and use pointers to different parts of it as different parameters. For example, the pointer `Vptr` for the basis V points at the beginning of this work array, the pointer `Wptr` for $W = AV$ points at `Wptr = Vptr + N*maxBasisSize`, which is Nm_{max} elements later, and so on. After all variables that are present in the algorithm have been accounted for, the remaining memory is shared among functions as temporary storage. We have also ensured that the allocated memory is aligned with a page boundary. There are two reasons for this. First, we wanted natural memory alignment for our double precision and double complex data types (8 and 16 bytes respectively). Although in many systems `malloc` will align in multiples of 8 bytes, this is not guaranteed in general, and depending on the memory/bus architecture it may not be sufficient for our double complex data. Second, neither `memalign` or `posix_memalign` are portable, so we were led to use the older but still widely available `valloc`. The use of `valloc` is not often recommended, because to guarantee the page alignment it may waste big fractions of a page. In our case this is not an issue because memory allocation occurs only once and for very large sizes.

The PRIMME code is both sequential and parallel. By this we mean that a parallel SPMD application can invoke the same PRIMME code, providing the local vector dimensions on each processor. As with all SPMD iterative methods, vector updates are performed in parallel while dot products require a global summation of the reduced value. PRIMME, includes a wrapper function for global sum. In sequential programs, this wrapper defaults to a sequential memory dcopy. In parallel programs, the user must provide a pointer to a global sum function, such as a wrapper to `MPI_allReduce()` or `pvmfreduce()`. Hence, PRIMME is independent from the communication library. Finally, the user must also provide a parallel matrix-vector multiplication and parallel preconditioning functions.

The PRIMME library adheres to the ANSI C standard so it should be widely portable to all current platforms. We have tested our code with the following operating systems: SUSE Linux 2.6.13-15.12 (both 32 and 64 bit), CentOS Linux 2.6.9-22 (64 bit), Darwin 8.8.0 on PowerPC, SunOS 5.9, and AIX 5.2. Macros have been used to resolve name mangling issues when interfacing with Fortran libraries and functions. We have also provided macros for "extern" declarations for allowing the library to be compiled with C++ compilers.

4.1.1. Structure, maintenance, and documentation. The distribution of the PRIMME package includes in excess of 28,000 lines of code. The difficulty in maintaining this code is not only its length, but that it implements all possible combinations of several algorithms and techniques, that can also be extremely complicated themselves. In our multi-year experience, we have found that the best way to remember complicated algorithms and data structures and the many special cases is to include the critical parts of the algorithm description in the comments of functions, and in-line explanations between code lines. Input/output arguements for each function are also documented, but we have found them less useful than the above.

The complex Hermitian and double precision codes are almost identical except for calls to different BLAS/LAPACK functions, certain memory copying, and the handling of various scalar issues. As in many software packages, the BLAS/LAPACK interface is handled by

a layer of wrappers. In this layer, our Num_AXPY function is an interface that can link to ZAXPY or DAXPY, depending on the code, and it could append underscores depending on the compiler. Similarly for other BLAS/LAPACK functions. To facilitate further implementation and management of the complex/double libraries, we have developed a single source code that includes both the complex and real functionalities, differentiated by macros. A pass through the preprocessor generates the two directories found in the public distribution, each containing a different precision version of PRIMME. To allow coexistence of both complex and real versions in the library, all functions are appended either with _dprimme or _zprimme.

The directory structure of the PRIMME distribution is as follows:

```
COPYING.txt     <- LGPL License
Make_flags      <- flags to be used by makefiles to compile library and tests
Link_flags      <- flags needed in making and linking the test programs
PRIMMESRC/      <- Directory with source code in the following subdirectories:
   COMMONSRC/   <- Interface and common functions used by all precision versions
   DSRC/        <- The source code for the double precision dprimme
   ZSRC/        <- The source code for the double complex precision zprimme
DTEST/          <- dprimme sample C and F77 drivers, both seq and parallel
ZTEST/          <- zprimme sample C and F77 drivers, sequential only
libprimme.a     <- The PRIMME library (to be made)
makefile        <- makes the libraries, and the sequential/parallel tests
readme.txt      <- a detailed documentation in text
readme.html     <- the same documentation organized with hyperlinks
doc.pdf         <- a printable version of the html documentation
```

The code is distributed with a Lesser GPL license. All library functions are located in PRIMMESRC directory. The ones that are specific to the double precision version are in PRIMMESRC/DSRC/ and for the complex version in PRIMMESRC/ZSRC/. All these files are appended with _d.c, _z.c or _d.h, _z.h for the real or complex versions respectively. In PRIMMESRC/COMMONSRC/ all functions are prepended with primme_ because they are the interface functions that do not depend on the data types and are common to both precisions. This directory contains also the header files (such as primme.h and primme_f77.h that are needed to call PRIMME. The functions dprimme and zprimme are called to solve the eigenvalue problem and they are located in (D)ZSRC/primme_(d)z.c. To simplify notation consider only the double version. Algorithms 3.1 and 3.7 are implemented in functions PRIMMESRC/DSRC/main_iter_d.c. Algorithms 3.2 and 3.3 are implemented in functions PRIMMESRC/DSRC/convergence_d.c and PRIMMESRC/DSRC/restart_d.c. Algorithm 3.4 is implemented in PRIMMESRC/DSRC/locking_d.c which also implements the second part of Algorithm 3.6. Algorithm 3.5 is implemented in PRIMMESRC/DSRC/inner_solve_d.c. We note that there is a separate function correction_d.c which implements the various preconditioning options for step (10) of the outer algorithm. In particular it can perform Olsen's or GD preconditioning, robust shifting, it can set up the JD projectors in a way specified by the user and possibly call QMRs to solve the correction equation. Orthogonalization is located in ortho_d.c.

In the DTEST and ZTEST directories we have provided several sequential and one parallel sample driver programs that read matrix and solver information from files and call PRIMME to solve various eigenproblems. We have provided the function ILUT from SPARSKIT [58] as a sample sequential preconditioner, and PARASAILS as a sample parallel preconditioner. We warn the user, however, that ILUT does not necessarily yield symmetric factorizations, which may cause stagnation to iterative methods for symmetric linear systems. Compared to CG, QMRs has proved remarkably robust in this direction, but it could still slow down significantly if the preconditioner is far from symmetric. GD+k methods do not share this problem. For further details on makefiles and linking we refer the reader to the extensive

```
#include "primme.h"
primme_params primme;
primme_initialize(&primme);

primme.n = N;
primme.matrixMatvec        = Matvec_function;
primme_set_method(DYNAMIC, &primme);

ierr = dprimme(evals, evecs, rnorms, &primme);
```

FIG. 4.1. *A minimal user-interface to PRIMME. Method is set to* `DYNAMIC`. *Other self-explanatory method choices are* `DEFAULT_MIN_MATVECS`, *and* `DEFAULT_MIN_TIME`. *The function pointers* `Matvec_function` *and* `Precon_function` *are provided by the user.*

information in the documentation files of the distribution.

4.2. A multi-layer interface. A full documentation on how to install and run PRIMME is included in the distribution in text, html, and pdf formats. Despite PRIMME's complexity, we have provided a multi-layer interface that hides this complexity from the users to the level determined by their expertise. Our premise has been that the beginner, end-user would probably be unaware not only of various techniques and tuning knobs, but also of the names of the methods. More experienced users, or as end-users gain more experience with the code, they should be able to use incrementally additional functionality to match their specific needs. PRIMME caters also to expert users who might use the code not only for solution of large problems but also to experiment with new techniques, combinations of methods, etc.

Figure 4.1 shows a minimal interface required by PRIMME. All users must declare a parameter of type `primme_params` that holds all solver information, and is used both for input and some output. Although not strictly required, a call to our initialization function is strongly recommended. Then, users may set any desired problem and solver information. A required field is the dimension of the matrix `primme.n`, and the matrix vector multiplication function. The user can then set the desired method and call `dprimme` to solve the problem. For the non-expert user, we provide three generic method choices `DEFAULT_MIN_MATVECS` (which defaults to GD+k), `DEFAULT_MIN_TIME` (which defaults to JDQMR_ETol), and `DYNAMIC`. The latter switches dynamically between the first two based on Algorithm 3.7. Finally, if a preconditioning operator is available, it can be set (before setting the method) as follows:

```
primme.applyPreconditioner = Precon_function;
primme.correctionParams.precondition = 1;
```

The preconditioner and the matrix-vector functions should have the following arguments:

```
void (*function_name)
   (void *x, void *y, int *blockSize, struct primme_params *primme);
```

where `x` is the input multivector, `y` is the output (result) multivector, `blockSize` is the number of vectors in the multivectors, and `primme` is passed so that any solver or external data (as the matrix or the preconditioner) can be available in the function. A wrapper with this interface can be easily written around existing, complicated, or legacy functions. Finally note that the multivectors store individual vectors consecutively in memory.

The minimal interface makes heavy use of defaults. For example, the above snippet of code will find one, smallest algebraic eigenvalue and its eigenvector, with residual norm $\|\mathbf{r}\| < 10^{-12} * \|A\|$, while estimating $\|A\|$ internally. It will alternate between GD+1 and JDQMR, using $m_{min} = 6$, $m_{max} = 15$, $b = 1$, and $k = 1$. We emphasize that, despite the

```
#include "primme.h"
primme_params primme;
primme_initialize(&primme);

double shifts[1] = {0.5};
double evecs[N*20] = {/*initialize the first 10 vectors*/}

primme.n                = N;
primme.numEvals         = 20;
primme.target           = primme_closest_abs;
primme.numTargetShifts  = 1;
primme.targetShifts     = shifts;
primme.aNorm            = 1.0;
primme.eps              = 1.0e-10;
primme.initSize         = 10;
primme.maxMatvecs       = 30000;
primme.matrixMatvec         = Matvec_function;
primme.applyPreconditioner = Precon_function;
primme.correctionParams.precondition = 1;
primme_set_method(DEFAULT_MIN_TIME, &primme);

ierr = dprimme(evals, evecs, rnorms, &primme);
```

FIG. 4.2. *A lean user-interface to PRIMME, where a default method with default parameters are used. However, the problem to be solved is fully controlled by the user, with parameters such as what eigenvalues to target, how accurately, initial guesses, and preconditioner.*

simplicity of the interface, the defaults and the methods reflect expertly tuned, near optimal methods. In fact, the above code snippet for finding one smallest eigenvalue of difficult problems has matched or outperformed all other software we are aware of.

Most users would like to have more control on the problem they are solving, than the minimal interface. Figure 4.2 shows a detailed, but still lean interface. By detailed we mean that the user specifies the exact problem to be solved; the dimension of the matrix, the number of eigenvalues, where these eigenvalues are located (they should be found closest in absolute distance from the shift 0.5), the exact residual norm convergence tolerance (10^{-10} =`primme.eps*primme.aNorm`), the number of initial guesses available in `evecs`, the maximum number of matvecs, the operators. None of the above parameters determines any algorithmic features; so this is functionality that an end-user is well qualified to use. The user can then request the default PRIMME strategy for yielding minimum time and solve the given problem.

The list of preset methods available in PRIMME are listed in Figure 4.3. A few comments are in order. We do not recommend the use of the methods of Arnoldi and classical GD, as they are superseded by GD+k methods. The default method for min matvecs is `GD_Olsen_plusK`, which is the usual GD+k with the preconditioner applied to the "robustly shifted" $\mathbf{r}^{(m)} + \delta\theta\, \mathbf{u}^{(m)}$ as described in section 3.2. `JD_Olsen_plusK` applies the pseudoinverse of the preconditioner of eq. (3.4) to the residual. `RQI` can be used either as RQI or as Inverse Iteration; the latter if the user provides at least one target shift. `JDQR` is a classic JD method, similarly to the JDBSYM implementation. `JDQMR` is our JDQMR Algorithm 3.5 *without* the stopping criterion ($r_k^{(m+1)} < 0.1 r_0$). Near convergence, this allows the inner equation to be solved very accurately and achieve the outer Newton convergence. However, for

```
primme_preset_method method;
typedef enum {
   DYNAMIC,                  // Switches to the best method dynamically
   DEFAULT_MIN_TIME,         // Currently set as JDQMR_ETol
   DEFAULT_MIN_MATVECS,      // Currently set as GD_Olsen_plusK
   Arnoldi,                  // Anoldi implemented a la Generalized Davidson
   GD,                       // Generalized Davidson
   GD_plusK,                 // GD+k with locally optimal restarting for k evals
   GD_Olsen_plusK,           // GD+k, preconditioner applied to (r+deltaeps*x)
   JD_Olsen_plusK,           // As above, only deltaeps computed as in JD
   RQI,                      // (accelerated) Rayleigh Quotient Iteration
   JDQR,                     // Jacobi-Davidson with const number of inner steps
   JDQMR,                    // JDQMR adaptive stopping criterion for inner QMR
   JDQMR_ETol,               // JDQMR + stops after resid reduces by a 0.1 factor
   SUBSPACE_ITERATION,       // Subspace iteration
   LOBPCG_OrthoBasis,        // A LOBPCG implementation with orthogonal basis
   LOBPCG_OrthoBasis_Window  // As above, only finds evals a Window at a time
} primme_preset_method;
```

FIG. 4.3. *The set of preset methods available in PRIMME. These can be selected by* `primme_set_method`*. After setting the method, the user can still modify some of the preset* `primme` *parameters.*

some cases, we noticed that QMRs tends to repeat some information between outer steps which causes the aforementioned slowdown. By stopping the inner method also when the eigenvalue residual (not the linear system one) is reduced by an order of magnitude, we achieved much smaller slowdown. We refer to this preset method which corresponds exactly to Algorithm 3.5 as `JDQMR_ETol`. There are two versions of LOBPCG, both maintaining an orthonormal basis of the search space. `LOBPCG_OrthoBasis` uses $numEvals = blockSize$ while `LOBPCG_OrthoBasis_Window` uses $blockSize < numEvals$ and locking, to find all the eigenvalues a window of $blockSize$ at a time.

If not provided, PRIMME picks defaults for maximum basis size (m_{max}), restart size m_{min}, block size (b), etc. Maximum basis size is by default 15 for extreme eigenvalue problems, and 35 for interior ones. When only m_{max} is provided, $m_{min} = 0.4 m_{max}$ for extreme eigenvalue problems, and $m_{min} = 0.6 m_{max}$ for interior ones. When the user sets the block size, but not the m_{max} and m_{min}, these are chosen such that b divides the $m_{max} - m_{min} - k$. Depending on the method, the above parameters may change further.

Finally, a few users may opt to set a preset method, and then modify various parameters manually, or even not to set a preset method at all. Figure 4.4 shows the full PRIMME interface available through the `primme` structure. For detailed explanation for each parameter we refer the reader to the distributed documentation.

In the previous examples we have used the double precision version `dprimme`. The complex Hermitian version `zprimme` is called in an analogous way. Finally, we note that to facilitate portability and usability, we have provided a Fortran 77 interface that covers the full functionality of PRIMME. This is a set of wrappers that allows Fortran users to set all the members of the primme structure, to set methods, and to call the PRIMME interface functions. An example is given in Figure 4.5, while we refer to the documentation manual for the complete interface.

4.3. Additional special features. We would like to briefly mention a few features that improve usability of the code, and although some can be found in other software packages, they have never been incorporated in the same package.

First, users can find eigenvalues in five different ways. Two for extreme eigenvalues, and

```
#include "primme.h"
primme_params primme;

primme.n                 = N;
primme.nLocal            = N;
primme.numProcs          = 1;
primme.commInfo          = NULL;
primme.globalSumDouble   = DCOPY_;
primme.outputFile        = stdout;
primme.printLevel        = 5;
primme.numEvals          = 10;
primme.aNorm             = 1.0;
primme.eps               = 1.0e-12;
primme.dynamicMethodSwitch= 0;
primme.maxBasisSize      = 15;
primme.minRestartSize    = 7;
primme.maxBlockSize      = 1;
primme.locking           = 1;
primme.maxOuterIterations = 10000;
primme.maxMatvecs        = 300000;
primme.target            = primme_smallest;
primme.numTargetShifts   = 0;
primme.targetShifts      = Shifts;
primme.initSize          = 0;
primme.numOrthoConst     = 0;
primme.intWorkSize       = 1000;
primme.intWork           = &intWorkArray;
primme.realWorkSize      = 0;
primme.realWork          = NULL;
primme.iseed[0]          = -1;
primme.restartingParams.scheme           = primme_thick;
primme.restartingParams.maxPrevRetain    = 1;
primme.correctionParams.precondition     = 1;
primme.correctionParams.robustShifts     = 1;
primme.correctionParams.maxInnerIterations = -1;
primme.correctionParams.relTolBase       = 1.5;
primme.correctionParams.convTest         = adaptive_ETolerance;
primme.correctionParams.projectors.LeftQ  = 1;
primme.correctionParams.projectors.LeftX  = 1;
primme.correctionParams.projectors.RightQ = 0;
primme.correctionParams.projectors.SkewQ  = 0;
primme.correctionParams.projectors.RightX = 1;
primme.correctionParams.projectors.SkewX  = 1;
primme.ShiftsForPreconditioner = NULL;
primme.matrixMatvec            = Matvec_function;
primme.applyPreconditioner     = Precon_function;
primme.matrix                  = &matrixDataStruct;
primme.preconditioner          = &preconDataStruct;
```

FIG. 4.4. *The full PRIMME interface. Expert users may set this manually, or combine with preset methods.*

three for interior. This is summarized in the following table.

primme_smallest	Smallest algebraic eigenvalues. No shifts are needed.
primme_largest	Largest algebraic eigenvalues. No shifts are needed.
primme_closest_geq	Closest to, but greater or equal than a set of shifts.
primme_closest_leq	Closest to, but less or equal than a set of shifts.
primme_closest_abs	Closest in absolute value to a set of shifts.

For interior eigenvalues the user must provide at least one shift in the pointer to an array: `primme.targetShifts`. Assuming that $q =$`primme.numTargetShifts` are available in the

```
double precision norm
integer primme   (or integer*8 primme if on a 64 bit OS)
external matvec_function
external precon_function

call primme_initialize_f77(primme)
call primme_set_member_f77(primme, PRIMMEF77_n, N)
call primme_set_member_f77(primme, PRIMMEF77_matrixMatvec, matvec_function)
call primme_set_member_f77(primme, PRIMMEF77_applyPreconditioner, precon_function)
call primme_set_member_f77(primme, PRIMMEF77_correctionParams_precondition, 1)
call primme_set_method_f77(primme, PRIMMEF77_JDQMR_ETol, ierr)
call primme_display_params_f77(primme);

call zprimme_f77(evals, evecs, rnorms, primme, ierr)

call primmetop_get_member_f77(primme, PRIMME_aNorm, norm)
print*, 'The estimated 2 norm of the matrix is:', norm
```

FIG. 4.5. *An example of using the Fortran 77 interface to call PRIMME.*

above array, and for simplicity denote them as $\tau_1, \ldots, \tau_q$. If the user chooses the interior mode: `primme_closest_leq`, PRIMME will find eigenvalues $\lambda_i, i = 1, numEvals$ that are closest to those shifts in the following way:

$$(\lambda_1 \leq \tau_1),\ (\lambda_2 \leq \tau_2),\ \ldots\ (\lambda_q \leq \tau_q),\ (\lambda_{q+1} \leq \tau_q),\ \ldots\ (\lambda_{numEvals} \leq \tau_q).$$

The other interior modes work similarly. The common mode `primme_closest_abs` might be wasteful in some cases when scientists want to find eigenvalues that are on one side of a given shift. Moreover, we have noticed that it is often faster to find the eigenvalues first on the left and then on the right of a shift, instead of using the common `primme_closest_abs` mode.

Another useful feature in PRIMME is that the `primme` structure supplies an array of the Ritz values corresponding to the vectors in the block to be preconditioned (or corrected). Many applications can afford to invert the preconditioner at every step. Examples include a diagonal matrix preconditioner, an FFT transform of the Laplace operator in planewave space (where the preconditioner is diagonal), or when the preconditioner is an iterative method. In those cases, instead of $K^{-1} \approx A^{-1}$, a more appropriate preconditioner would be $(K - \lambda_i I)^{-1}$. The $\lambda_i, i = 1, b$ values are available in the array `primme.ShiftsForPreconditioner` and are accessible during the preconditioning operation. This feature is not readily available in other packages.

In some cases, we want to solve an eigenvalue problem under certain orthogonality constraints, i.e., solve the eigenvalues of $(I - QQ^T)A(I - QQ^T)$, where Q could be previously computed eigenvectors, or any set of vectors in general. PRIMME works seamlessly in this case by including Q in the first `primme.numOrthoConst` vectors of `evecs`. Computed eigenvectors will be placed after Q.

Finally, we mention that PRIMME includes a thorough parameter checking of user inputs for consistency and correctness, a calling tree traceback report for tracing errors if any occur, and five levels of output reporting, so that convergence history and algorithmic choices can be monitored or plotted.

5. Sample experimental results. We have compared PRIMME methods with three other software packages. The first is JDBSYM [25], which currently is the only other implementation of the Jacobi-Davidson for symmetric problems. The second is the BLOPEX

implementation of LOBPCG [38]. The third is ARPACK's function dsaupd, which implements IRL for symmetric matrices [44]. Although ARPACK does not use preconditioning, it is included as the default benchmark for unpreconditioned cases. We have not compared with SLEPc methods as they do not allow for preconditioning. Also, we have not yet compared with ANASAZI because of its involved installation and optimization process, but also because the methods it implements (block GD and LOBPCG) are available in PRIMME, and LOBPCG in BLOPEX. We plan to make a comparison in the near future. The experiments we present are sampled from our papers [66] and [68] which include one of the most extensive list of comparisons in the literature.

With the exception of forcing the various projector configurations (100), (111), (000), and (011), the default parameters provided by PRIMME are used in all the experiments. The methods converge when the residual norm of each of the $numEvals$ required eigenpairs is less than $\|A\|_F tol$, where $\|A\|_F$ is the Frobenious norm of A. For JDBSYM we use the same m_{min}, m_{max} as in PRIMME, block size of 1, a maximum number of 200 inner iterations, TOLDECAY = 1.5, symmetric preconditioning OPTYPE, and strategy = 0. In certain cases, strategy = 1 was necessary to achieve convergence. To find extreme eigenpairs with JDBSYM, we provide τ as a small, left perturbation of the precomputed λ_1, and we let JDBSYM switch to using the Ritz values as shifts when EPS_TR = $10^{-3}\|A\|_F/\sqrt{N}$. Convergence is declared when all residual norms fall below $\|A\|_F tol$.

ARPACK does not implement locking, so when many eigenvalues are required ARPACK must use a much larger basis size than the rest of the methods. We choose the basis size for ARPACK as $\max(40, 2numEvals)$, and supply directly the tolerance tol.

BLOPEX does not explicitly implement locking, but it allows users to find eigenvectors orthogonal to a set of a given vectors. We chose to implement a wrapper around BLOPEX(b) that uses locking to compute $numEvals$ eigenvalues a block, b, at a time. After some experimentation, we found $b = 10$ to be the best choice for most problems with large $numEvals$. In fact, for large $numEvals$, BLOPEX($numEvals$) was several times slower than BLOPEX(10). We ask for convergence tolerance of $\|A\|_F tol$.

All methods start with the same random initial guess. We have run experiments for two different tolerances: $tol =$ 1e-15 and $tol =$ 1e-7. For BLOPEX we only report results for $tol =$ 1e-7, *as it could not produce results with the lower tolerance.* We run experiments on an Apple G5 with 1 GB of memory and two 2GHz processors, each with 512 MB L2 cache. The C codes are compiled using the gcc-4.0.0 compiler with -O3 flag, and the ARPACK is compiled with the g77 compiler. We link with the Apple vecLib library that includes optimized versions of BLAS/LAPACK libraries.

We use 10 matrix problems, six from the University of Florida [16] and the FEAP [3] collections, one from vibrational analysis of molecular structures [75] and three standard 7-point 3D Laplacian matrices generated by SPARSKIT [58] with zero Dirichlet boundary conditions. The smallest side of the spectrum is hard to obtain for all matrices, while the largest side is easier for several of them.

5.1. Looking for one smallest eigenvalue. In Table 5.2 we provide comparisons between PRIMME, JDBSYM, and BLOPEX for finding one smallest eigenvalue of five matrices. The first three matrices are preconditioned with ILUT with its parameters chosen to provide a stable factorization. The last two matrices are unpreconditioned.

First, we observe that BLOPEX although it requires no parameter setting, is not competitive. In addition, in the cases where it failed to converge, it had reached a residual norm of $\|A\|_F 10^{-10}$ but then encountered numerical problems. The JDBSYM was used with a symmetric QMR as inner solver, so the primary difference from JDQMR is the stopping criteria. The experiments confirm that when the JDBSYM criteria [17] capture the Newton

TABLE 5.1
The matrices used in the experiments, their size, non-zero elements per row, and their source.

Matrix	N	nz/row	Source	Matrix	N	nz/row	Source
or56f	9000	321.13	Yang	Cone_A	22032	65.04	FEAP
torsion1	40000	4.94	UF	Plate33K_A0	39366	23.22	FEAP
Andrews	60000	12.67	UF	Lap7pt15K	12167	6.74	SKIT
cfd1	70656	25.84	UF	Lap7pt125K	110592	6.88	SKIT
finan512	74752	7.99	UF	Lap7pt1M	941192	6.94	SKIT

TABLE 5.2
Comparison of three state-of-the-art preconditioned eigensolver codes. BLOPEX implements LOBPCG, JDBSYM implements Jacobi-Davidson with sQMR as inner solver, and our PRIMME software includes both JDQMR and GD+1 which provide almost parameter-free near optimality.

	cfd1		or56f		Plate33K_A		cfd1		Cone_A	
	MV	sec	MV	sec	MV	sec	MV	sec	MV	sec
BLOPEX	669	114.14	332	21.89	-	-	6426	186.63	-	-
GD+1	270	49.92	174	11.56	272	39.89	2858	113.86	214	3.49
JDQMR	294	44.82	190	11.88	381	51.35	2370	49.45	281	3.19
JDBSYM	373	60.42	221	14.34	(747)	(102.6)	2412	48.95	(708)	(7.70)

convergence well, JDBSYM is close to JDQMR and sometimes competitive (unpreconditioned cfd1). The results in parentheses, however, show cases where JDBSYM could not converge, until the user provided a shift for the correction equation that was very close to the desired eigenvalue. GD+1 and JDQMR converge always, in the least time, and with no a priori information.

In Figure 5.1 we show results from two large dimension 3D Laplacians, the one million case, and a ten million case. The one million case is run both with and without preconditioning. The first thing we observe is that, as in Table 5.2, GD+1 always takes the smallest number of matvecs. However, because the matrices are very sparse, JDQMR takes less time. For the same reason, the timings for BLOPEX are competitive with GD+1 in this example, despite taking more iterations. ARPACK is not competitive even for the smaller matrix, and it scales worse as the problem increases in dimension.

5.2. Looking for many smallest eigenvalues.

Unpreconditioned JDQMR-000 vs ARPACK. For large *numEvals*, the number of matrix vector operations per eigenvalue found by ARPACK is expected to decrease rapidly with *numEvals*, because of the effectiveness of the large Lanczos basis. In contrast, the number of matvecs per new eigenvalue found for JDQMR-000 is expected to be at least constant or increase slightly for highly interior eigenpairs, because of the loss of implicit orthogonality during inner iterations. Despite this worse case scenario, where ARPACK is allowed to grow its memory requirements with *numEvals*, our asymptotic analysis in [68] showed that for sufficiently sparse matrices, such as those coming from many finite difference and finite element analysis, JDQMR-000 is faster than ARPACK for obtaining a very large number of eigenvalues (usually $numEvals > 1000$). For denser matrices, ARPACK becomes faster than the limited memory JDQMR-000 for smaller *numEvals*. However, according to the model, JDQMR-000 should always be faster than ARPACK for $numEvals < 5$, regardless of operator cost. Despite the approximate nature of the model for small *numEvals*, our above conclusions are confirmed by the experiments with 8 matrices in Figure 5.2.

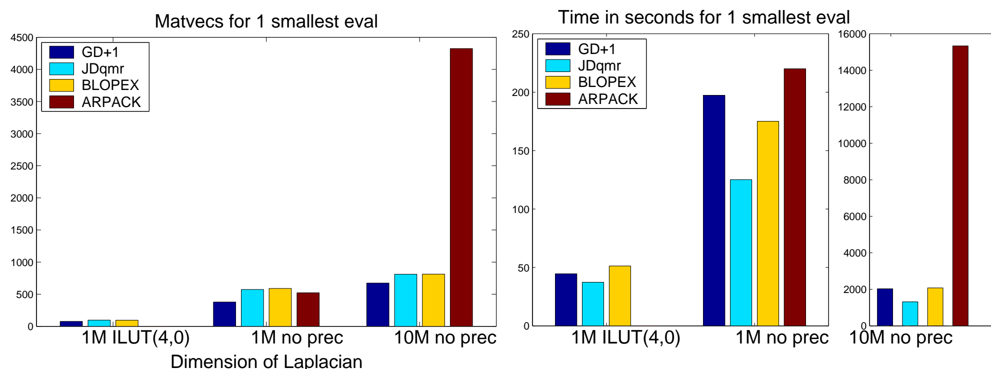

FIG. 5.1. *Finding one eigenvalue for Laplacians of 1 and 10 million size. GD+1 yields minimum iterations, JDQMR is the fastest method, BLOPEX is competitive, and ARPACK does not scale well.*

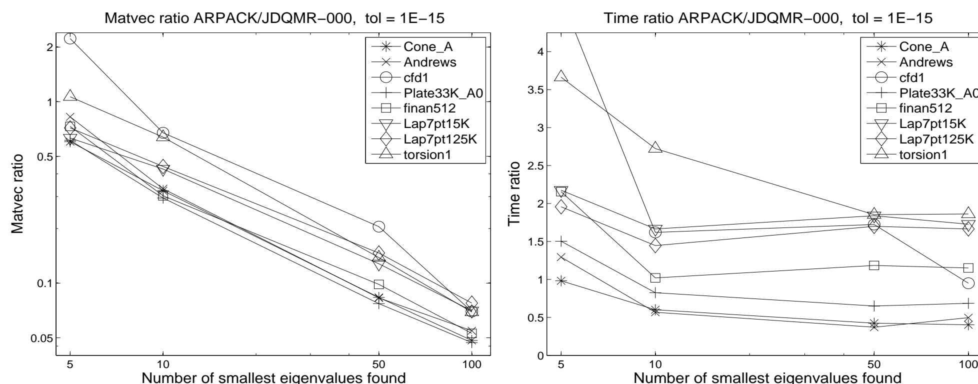

FIG. 5.2. *Relative performance of ARPACK over JDQMR-000 for finding numEvals smallest eigenvalues of 8 matrices. The left graph shows the matvec ratios. The right graph shows time ratios.*

Comparisons with other methods without preconditioning. In the following experiments, we look for the smallest 100 eigenvalues, and ask for tol=1e-15. BLOPEX could not reach the required tol, hence it is not reported. In Figure 5.3, all JD/GD methods converge very similarly including JDBSYM, which means that its stopping criteria work well in this case. Interestingly, most methods and particularly JDQMR-000 are better than ARPACK up to 50 eigenvalues, but for $numEvals = 100$, ARPACK uses a much larger basis which captures some part of the spectrum that smaller bases could not. ARPACK takes fewer matvecs to find 100 eigenvalues than 50, and it matches the time of JDQMR-000.

In Figure 5.4, we consider the Laplacian matrix of dimension 125K, whose eigenvalues are all of multiplicity 3 or 6. JDBSYM cannot converge in tractable time for this matrix with strategy = 1, and strategy = 0 performed worse. The sparsity of this Laplacian makes JDQMR-000 significantly faster than all other methods. Also, in all four examples, we see GD+1 always taking the minimum number of matvecs among JD methods, yet it loses timewise because of its more expensive iteration.

Figure 5.5 shows results from the cfd1 matrix, but with tol=1e-7. BLOPEX with block size of 10 is significantly slower than all PRIMME methods. We also observe a significant deterioration of the performance of ARPACK over the tol =1e-15 case in Figure 5.3. Closer scrutiny of the iterations between the two figures reveals they are about the same. ARPACK

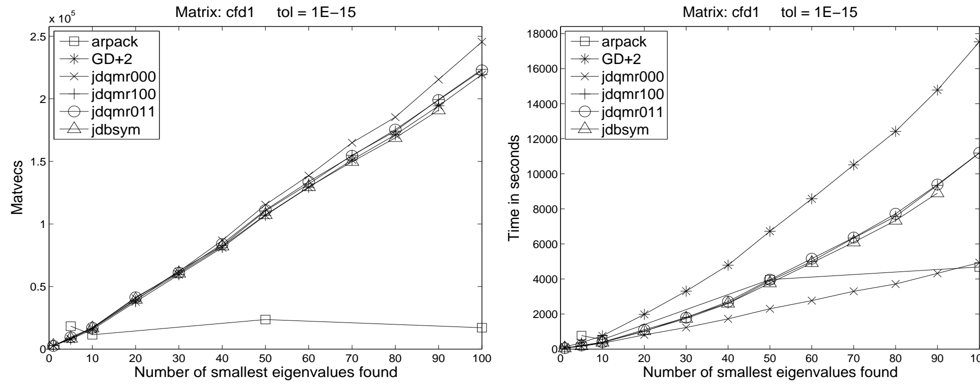

FIG. 5.3. *Matvecs (left graph) and time (right graph) of six methods for numEvals smallest eigenvalues.*

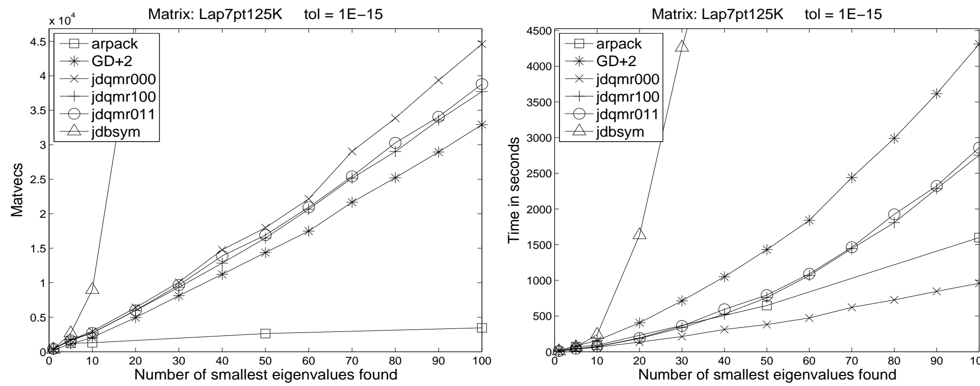

FIG. 5.4. *Matvecs (left graph) and time (right graph) of six methods for numEvals smallest eigenvalues.*

does not benefit from the higher threshold, still computing almost all 100 eigenpairs in full accuracy. We observed this behavior of ARPACK with high tolerances in the majority of our experiments. Surprisingly, JDBSYM is much slower for tol=1e-7 than with full accuracy (compare with Figure 5.3).

Figure 5.6 reports similar results for the Lap7pt125K matrix. Both BLOPEX and JDBSYM cannot converge in tractable time, and ARPACK does not benefit from the lower threshold. The JDQMR and GD+k methods are consistent both in robustness and their relative behavior. Experiments with seeking $numEvals = 500$ largest eigenvalues have confirmed a similar behavior of all methods.

Comparisons with other methods with preconditioning. For our preconditioning experiments, we use the the ILUT preconditioner from the SPARSKIT library [58].

In Figure 5.7, all preconditioned methods improve over their unpreconditioned versions and become much better than ARPACK. Notice that all JDQMR-100/111/011 variants converge identically, but the 111 takes more time for larger *numEvals*. JDBSYM also improves but not as much as JDQMR-100. Because of large fill-in, the computed ILUT factors are expensive and therefore the method with smallest matvecs wins, i.e., GD+1. The number of matvecs for ARPACK is large and out of scale.

Despite the use of preconditioning, BLOPEX was not able to reach convergence to tol =1e-15, in any of our test matrices. Therefore, we conclude this section with one ex-

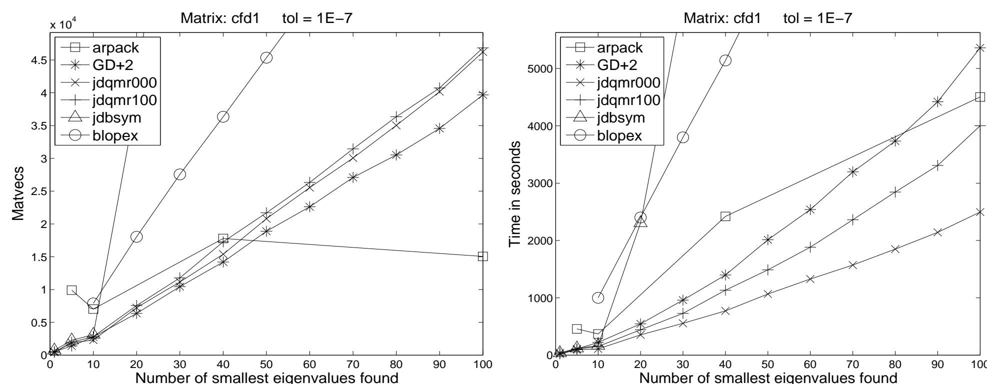

FIG. 5.5. *Matvecs (left) and time (right) of six methods with tol=1e-7, for smallest eigenvalues.*

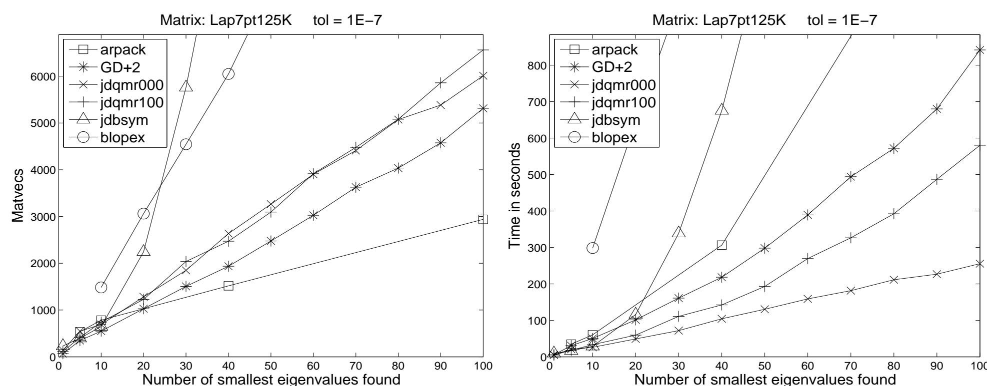

FIG. 5.6. *Matvecs (left) and time (right) of six methods with tol=1e-7, for smallest eigenvalues.*

periment that includes preconditioning but uses a higher tolerance, 1e-7.

In Figure 5.8, with an ILUT(20,1e-6) neither JDBSYM nor BLOPEX were competitive. The graphs also include the unpreconditioned JDQMR-000, whose performance is identical to preconditioned JDQMR-100.

6. Current and future work. We have motivated and described the theory that gives rise to the near optimal methods GD+k and JDQMR that constitute the basis of PRIMME. We have also described the many algorithmic, implementation, and interface features present in PRIMME. Our sample experiments demonstrate that our methods at least match, and typically improve significantly the fastest methods available. Even without preconditioning, PRIMME should be considered the method of choice for a small number of eigenvalues.

PRIMME is currently in a stable state, which means that no known bugs exist in the code at this time. The software and its documentation, however, evolve continuously. The following is a list of on-going and future projects, ordered by expected completion time.

1. Generalized eigenvalue programs. The current distribution of PRIMME includes an interface for generalized eigenproblems, but the functionality is not implemented yet. Traditionally, it is suggested that JD is based on a B inner product with the mass matrix. For stability reasons, we are working on a 2-norm orthogonality implementation, which is similar to JDQZ but exploits symmetry.
2. A Matlab interface to PRIMME. One of our collaborating groups in Europe is near-

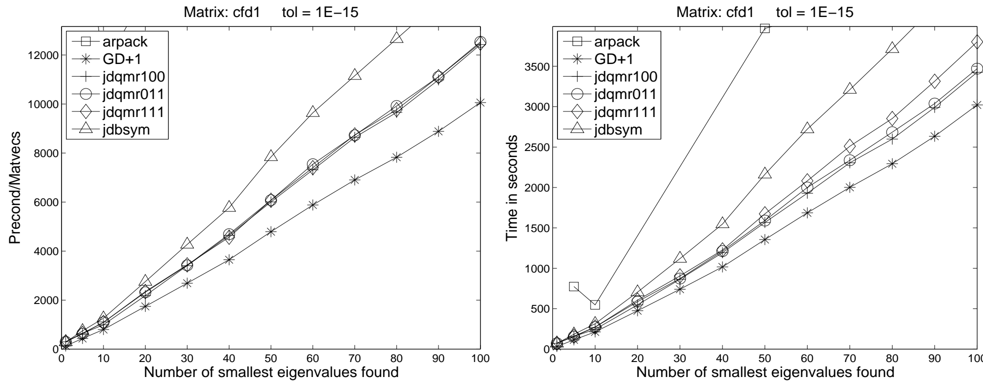

FIG. 5.7. *Matvecs (left) and time (right) with ILUT(80,1e-4) preconditioner. Smallest numEvals.*

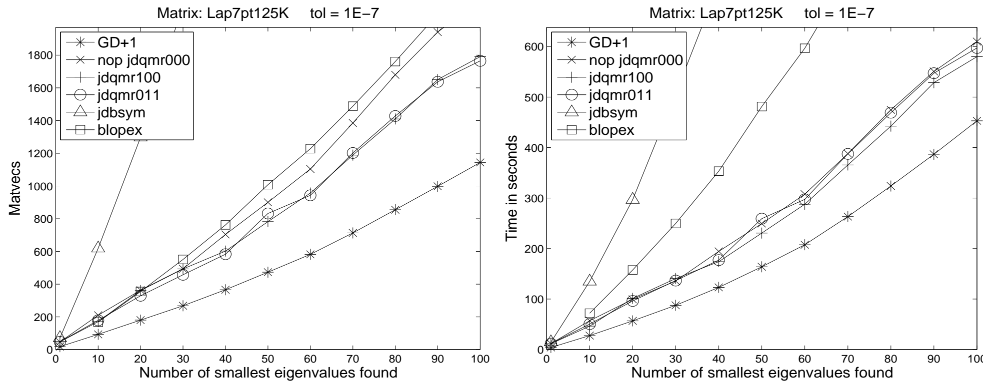

FIG. 5.8. *Matvecs (left) and time (right) with ILUT(20,1e-6). Smallest numEvals and tol=1e-7.*

ing the completion of such an interface, which will significantly improve the potential impact of the package.

3. A front-end that calls PRIMME to compute singular triplets of large sparse matrices. PRIMME's functionality allows both the A^TA approach, and the augmented matrix $[0\ A^T; A\ 0]$ approach. Factorized preconditioners of A, or other problem specific preconditioners can be readily used.
4. The current distribution of PRIMME implements only a subset of the Iterative Validation of Eigensolvers (IVE) algorithm [47]. We have a fully functional IVE working with an older version of PRIMME that will be ported to the current distribution. This will also be coordinated with the final Rayleigh-Ritz procedure over all locked vectors that is needed when a locking problem has occurred.
5. Implementation of the block orthogonalization algorithm SVQB [72].
6. A dynamic block that adjusts its size depending on the architecture, but also according to the clustering or multiplicity of the eigenvalues targeted at every step.
7. Further comparisons on more applications and matrices, and with additional software including ANASAZI and SLEPc.
8. The extension of PRIMME to non symmetric problems is a longer term goal. The basic structure is the same, but the lack of near optimal, global methods means that many algorithmic choices must rely upon heuristics.

REFERENCES

[1] P.-A. Absil, C. G. Baker, and K. A. Gallivan. A truncated-CG style method for symmetric generalized eigenvalue problems. *J. Comput. Appl. Math.*, 189(1–2):274–285, 2006.

[2] P.-A. Absil, R. Mahony, R. Sepulchre, and P. Van Dooren. A grassmann-rayleigh quotient iteration for computing invariant subspaces. *SIAM Review*, 44(1):57–73, 2002.

[3] M. F. Adams. Evaluation of three unstructured multigrid methods on 3d finite element problems in solid mechanics. *International Journal for Numerical Methods in Engineerin*, 55:519–534, 2002.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1992.

[5] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro. A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods. *International Journal of Numerical Methods in Engineering*, 64:204–236, 2005.

[6] J. Baglama, D. Calvetti, and L. Reichel. IRBLEIGS: A MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix. *ACM Transaction on Mathematical Software*, 29(5):337–348, 2003.

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

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[9] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, Philadelphia, PA, 1995.

[10] M. Clint and A. Jennings. The evaluation of eigenvalues and eigenvectors of a real symmetric matrix by simultaneous iteration. *Computer J.*, 13:76–80, 1970.

[11] M. L. Cohen and J. R. Chelikowsky. *Electronic Structure and Optical Properties of Semiconditors*. Springer-Verlag, New York, Berlin, Heidelberg, 2nd edition, 1989.

[12] J. Cullum and W.E. Donath. A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, symmetric matrices. In *Proc. 1974 IEEE Conference on Decision and Control*, pages 505–509, 1974.

[13] J. Cullum and R. A. Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations*, volume 2: Programs of *Progress in Scientific Computing; v. 4*. Birkhauser, Boston, 1985.

[14] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Math. Comp.*, 30(136):772–795, October 1976.

[15] E. R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Comput. Phys.*, 17:87–94, 1975.

[16] T. Davis. University of florida sparse matrix collection. Technical report, University of Florida. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 19.

[17] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact newton methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[18] E. G. D'yakonov. Iteration methods in eigenvalue problems. *Math. Notes*, 34:945–953, 1983.

[19] A. Edelman, T. A. Arias, and S. T. Smith. The geometry of algorithms with orthogonality constraints. *SIAM Journal on Matrix Analysis and Applications*, 20(2):303–353, 1999.

[20] C. F. Fischer. *The Hartree-Fock Method for Atoms: A numerical approach*. J. Wiley & Sons, New York, 1977.

[21] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1), 1998.

[22] Justin Foley et al. Practical all-to-all propagators for lattice qcd. *Comput. Phys. Commun.*, 172:145–162, 2005.

[23] R. W. Freund and N. M. Nachtigal. A new Krylov-subspace method for symmetric indefinite linear systems. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1994.

[24] G. Gambolati, F. Sartoretto, and P. Florian. An orthogonal accelerated deflation technique for large symmetric eigenproblems. *Comp. Methods App. Mech. Eng.*, 94:13–23, 1992.

[25] R. Geus. JDBSYM. http://people.web.psi.ch/geus/software.html.

[26] R. Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*. PhD thesis, ETH, 2002. Thesis. No. 14734.

[27] P. H. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1986.

[28] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. In J. R. Rice, editor, *Mathematical Software III*, pages 361–377, New York, 1977. Academic Press.

[29] G. H. Golub and Q. Ye. Inexact inverse iteration for generalized eigenvalue problems. *BIT*, 40(4):671–684, 2000.

[30] G. H. Golub and Q. Ye. An inverse free preconditioned krylov subspace methods for symmetric generalized eigenvalue problems. *SIAM J. Sci. Comput.*, 24:312–334, 2002.

[31] R. G. Grimes, J. G. Lewis, and H. D. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl.*, 15(1):228–272, 1994.

[32] Martin H. Gutknecht. Block Krylov space solvers: A survey. http://www.sam.math.ethz.ch/ mhg/talks/bkss.pdf.

[33] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. A survey of software for sparse eigenvalue problems. Technical Report SPEPc STR-6, Unversidad Politecnica de Valencia, October, 2006.

[34] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, sep 2005.

[35] U. Hetmaniuk and R. B. Lehoucq. Basis selection in LOBPCG. *J. Comp. Phys.*, to appear, 2006.

[36] Z. Jia. A refined iterative algorithm based on the block Arnoldi process for large unsymmetric eigenproblems. *Linear Algebra and Its Applications*, 270:171–189, 1998.

[37] S. G. Johnson and J. D. Joannopoulos. Block-iterative frequency-domain methods for maxwell's equations in a planewave basis. *Opt. Express*, 8(3):173–190, 2001.

[38] A. V. Knyazev. BLOPEX. http://www-math.cudenver.edu/ aknyazev/software/BLOPEX.

[39] A. V. Knyazev. Convergence rate estimates for iteratime methods for symmetric eigenvalue problems and its implementation in a subspace. *International Ser. Numerical Mathematics*, 96:143–154, 1991. Eigenwertaufgaben in Natur- und Ingenieurwissenschaften und ihre numerische Behandlung, Oberwolfach, 1990.

[40] A. V. Knyazev. Preconditioned eigensolvers - an oxymoron? *Electr. Trans. Numer. Anal.*, 7:104–123, 1998.

[41] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[42] Y.-L. Lai, K.-Y. Lin, and W.-W. Lin. An inexact inverse iteration for large sparse eigenvalue problems. *Num. Lin. Alg. Appl.*, 4:425–437, 1997.

[43] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–325, 1979.

[44] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1998.

[45] B. Liu. Numerical algorithms in chemistry: Algebraic methods, eds. c. moler and i. shavitt. Technical Report LBL-8158, Lawrence Berkeley Laboratory, 1978.

[46] E. Lundström and L. Eldén. Adaptive eigenvalue computations using Newton's method on the Grassmann manifold. *SIAM Journal on Matrix Analysis and Applications*, 23(3):819–839, July 2002.

[47] J. R. McCombs and A. Stathopoulos. Iterative validation of eigensolvers: A scheme for improving the reliability of hermitian eigenvalue solvers. *to appear in SISC*, (Tech. report WM-CS-2005-02).

[48] R. B. Morgan. Computing interior eigenvalues of large matrices. *Lin. Alg. Appl.*, 154–156:289–309, 1991.

[49] R. B. Morgan and D. S. Scott. Generalizations of Davidson's method for computing eigenvalues of sparse symmetric matrices. *SIAM J. Sci. Comput.*, 7:817–825, 1986.

[50] C. W. Murray, S. C. Racine, and E. R. Davidson. Improved algorithms for the lowest eigenvalues and associated eigenvectors of large matrices. *J. Comput. Phys.*, 103(2):382–389, 1992.

[51] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer-Verlag, New York, 1999.

[52] Y. Notay. Combination of Jacobi-Davidson and conjugate gradients for the partial symmetric eigenproblem. *Numerical Linear Algebra with Applications*, 9:21–44, 2002.

[53] Y. Notay. Is Jacobi-Davidson faster than Davidson? *SIAM J. Matrix Anal. Appl.*, 26(2):533–543, 2005.

[54] J. Olsen, P. Jørgensen, and J. Simons. Passing the one-billion limit in full configuration-interaction (FCI) calculations. *Chem. Phys. Lett.*, 169(6):463–472, 1990.

[55] C. C. Paige, B. N. Parlett, and Van der Vorst. Approximate solutions and eigenvalue bounds from Krylov spaces. *Num. Lin. Alg. Appl.*, 2:115–133, 1995.

[56] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, 1998.

[57] A. Ruhe and T. Wiberg. The method of conjugate gradients used in inverse iteration. *BIT*, 12(4):543–554, 1972.

[58] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990. Software currently available at <ftp://ftp.cs.umn.edu/dept/sparse/>.

[59] A. Sameh and Z. Tong. The trace minimization method for the symmetric generalized eigenvalue problem. *J. Comput. Appl. Math.*, 123:155–175, 2000.

[60] R. Shepard, A. F. Wagner, J. L. Tilson, and M. Minkoff. The subspace projected approximate matrix (SPAM) modification of the davidson method. *J. Computational Physics*, 172(2):472–514, 2001.

[61] V. Simoncini and L. Eldén. Inexact Rayleigh quotient-type methods for eigenvalue computations. *BIT Numerical Mathematics*, 42(1):159–182, 2002.

[62] G. L. G. Sleijpen and H. A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, 1996.

[63] P. Smit and M. H. C. Paardekooper. The effects of inexact solvers in algorithms for symmetric eigenvalue problems. *Linear Algebra Appl.*, 287(1-3):337–357, 1999. Special issue celebrating the 60th birthday of

Ludwig Elsner.
[64] D. C. Sorensen. Implicit application of polynomial filters in a K-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13(1):357–385, 1992.
[65] A. Stathopoulos. Locking issues for finding a large number of eigenvectors of Hermitian matrices. *Submitted*, (Technical Report WM-CS-2005-09).
[66] A. Stathopoulos. Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue. *to appear in SISC*, (Technical Report WM-CS-2005-03).
[67] A. Stathopoulos and C. F. Fischer. A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix. *Computer Physics Communications*, 79(2):268–290, 1994.
[68] A. Stathopoulos and J. R. McCombs. Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part II: Seeking many eigenvalues. *Submitted*, (Technical Report WM-CS-2006-02).
[69] A. Stathopoulos and Y. Saad. Restarting techniques for (Jacobi-)Davidson symmetric eigenvalue methods. *Electr. Trans. Numer. Alg.*, 7:163–181, 1998.
[70] A. Stathopoulos, Y. Saad, and C. F. Fischer. Robust preconditioning of large, sparse, symmetric eigenvalue problems. *Journal of Computational and Applied Mathematics*, 64:197–215, 1995.
[71] A. Stathopoulos, Y. Saad, and K. Wu. Dynamic thick restarting of the Davidson, and the implicitly restarted Arnoldi methods. *SIAM J. Sci. Comput.*, 19(1):227–245, 1998.
[72] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirement. *SIAM J. Sci. Comput.*, 23(6):2165–2182, 2002.
[73] H. Thornquist, C. Baker, R. Lehoucq, and U. Hetmaniuk. ANASAZI: block eigensolver package. http://software.sandia.gov/trilinos/packages/anasazi.
[74] K. Wu and H. D. Simon. Thick-restart Lanczos method for symmetric eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22(2):602–616, 2001.
[75] C. Yang, B. W. Peyton, D. W. Noid, B. G. Sumpter, and R. E. Tuzun. Large-scale normal coordinate analysis for molecular structures. *SIAM Journal on Scientific Computing*, 23(2):563–582, 2001.
[76] Q. Ye and J. Money. Algorithm 845: Eigifp: A matlab program for solving large symmetric generalized eigenvalue problems. *ACM Transaction on Mathematical Software*, 31:270–279, 2005.