PRIMME: PReconditioned Iterative MultiMethod Eigensolver: Methods and software description

Andreas Stathopoulos
and
James R. McCombs

This paper describes the PRIMME software package for solving large, sparse Hermitian standard 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. However, the complexity of tuning or even using such methods has kept them outside the reach of many users. Responding to this problem we have developed PRIMME, a comprehensive package that brings state-of-the-art methods from "bleeding edge" to production, with the best possible robustness, efficiency, and a flexible, yet highly usable interface that requires minimal or no tuning. We describe: (1) the PRIMME multimethod framework that implements a variety of algorithms, including the near optimal methods GD+k and JDQMR; (2) a host of algorithmic innovations and implementation techniques that endow the software with its robustness and efficiency; (3) a multilayer interface that captures our experience and addresses the needs of both expert and end users.

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

General Terms: Algorithms

Additional Key Words and Phrases: Davidson, Jacobi-Davidson, Lanczos, conjugate gradient, Hermitian, eigenvalues, eigenvectors, preconditioning, software package, iterative, block, locking

1. INTRODUCTION

PRIMME, or PReconditioned Iterative MultiMethod Eigensolver, is a software package for the solution of large, sparse Hermitian and real symmetric standard eigenvalue problems [Stathopoulos and McCombs 2006]. 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 [Grimes et al. 1994]. However,

Address for Andreas Stathopoulos: Department of Computer Science, College of William and Mary, Williamsburg, Virginia 23187-8795, (andreas@cs.wm.edu).
Address for James R. McCombs: Cobham Inc, 1911 N. Ft. Myer Drive, Arlington VA 22209, (james.mccombs@cobham.com)
Work supported partially by National Science Foundation grants: ITR/DMR 0325218, ITR/AP-0112727, ITR/ACS-0082094, CCF/TF-0728915.

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, specially 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 [Lehoucq et al. 1998]. Yet, the implicitly restarted Lanczos method [Sorensen 1992], on which ARPACK is based, does not converge optimally (i.e., converges slower than the unrestarted method) and can only use preconditioning via iterative, shift-invert spectral transformations. 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.

PRIMME's design philosophy is to provide

(1) preconditioned eigenmethods converging near optimally under limited memory,
(2) the maximum robustness possible without matrix factorization,
(3) flexibility in mixing and matching among most currently known features,
(4) efficiency at all architectural levels,
(5) and 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 presents how PRIMME accomplishes this by integration of state-of-the-art methods, new algorithms and implementation techniques, and a carefully crafted user interface. In Section 2 we describe the problem, its importance and difficulty, and briefly discuss other current eigenvalue software. In Section 3 we show how the algorithmic framework of PRIMME is built on top of the two near optimal methods GD+k and JDQMR, and how other algorithms can be parameterized within this framework. In Section 4 we discuss how the PRIMME software meets its design goals. After a few sample numerical comparisons in Section 5, we conclude in Section 6 with some discussion of on-going work and future extensions.

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 $\mathbf{x}_i$, $i = 1, \ldots, numEvals$ for $Ax_i = \lambda_i x_i$. 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. Examples abound in structural engineering [Grimes et al. 1994], electromagnetics [Geus 2002; Johnson and Joannopoulos 2001], lattice Quantum Chromodynamics (QCD) [Foley. et al. 2005], and electronic structure applications from atomic scale physics [Fischer 1977] to molecular scale materials science [Cohen and Chelikowsky 1989].

The challenge is twofold. First, the matrix size, n, is very large. Second, some applications, e.g., electronic structure calculations, require the computation of hun-

dreds or even thousands of extreme eigenpairs. Often the number of required eigenpairs is described as a small percentage of the problem size. In such cases, orthogonalization of $numEvals$ vectors, an $O(numEvals^2 n)$ task, becomes $O(n^3)$, making the scaling to larger problem sizes practically infeasible.

Iterative methods are the only means of addressing these problems but may converge slowly, especially as the problem size grows. 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 [Edelman et al. 1998; Knyazev 1998; Sleijpen and van der Vorst 1996]. This probably explains the noticeable scarcity of high quality, general purpose software for preconditioned eigensolvers. On the other hand, such applications can have staggering storage demands as the iteration vectors must be stored for computing eigenvector approximations. Restarting techniques can be employed so that approximations are obtained from a search space of limited size, see for example thick restarting [Stathopoulos et al. 1998] and the theoretically equivalent implicit restarting [Sorensen 1992]. However, this is at the expense of convergence.

Recently, iterative methods have been developed [Sleijpen and van der Vorst 1996; Stathopoulos and Saad 1998; Knyazev 2001; Simoncini and Eldén 2002; Notay 2002; Stathopoulos 2007; Stathopoulos and McCombs 2007] that can use effectively the large arsenal of preconditioners for linear systems and converge near optimally to an eigenpair under limited memory requirements. When seeking many eigenpairs, it is unclear whether optimality can be achieved under limited memory. Restarting techniques cannot maintain near optimal convergence for more than a small window of eigenvalues. Similarly, preconditioned methods typically correct for a particular eigenvector and so repeat the work for each required eigenvalue.

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. Without preconditioning it still offers a practical alternative to the Lanczos method. Lanczos requires storage for all iteration vectors or a second pass to recompute them. Moreover, in machine arithmetic, it produces spurious eigenvalues and must resort to expensive orthogonalization or post-processing techniques [Parlett 1998; Cullum and Willoughby 1985]. Finally, the computational expenses for the tridiagonal matrix grow significantly, and involved locking or block implementations are needed for multiple eigenvalues (see [Cullum and Donath 1974; Golub and Underwood 1977; Gutknecht 2005] and [McCombs and Stathopoulos 2006] for extensive bibliography on block methods).

2.1 Current state of Hermitian eigenvalue software

Iterative methods have gained notoriety as very difficult to include in general purpose software, not so much because of library interfaces and data structures, but because different problems may require different iterative solvers and preconditioners for robustness and/or efficiency. This has led a group of experts to produce the popular series of "Templates for the solution of linear systems" [Barrett et al. 1994], and "eigenvalue problems" [Bai et al. 2000]. Since then, the area of symmetric eigenproblems has seen some remarkable progress [Stathopoulos 2007; Stathopoulos and McCombs 2007; Notay 2005; Absil et al. 2006; Notay 2002; Geus 2002; Lundström and Eldén 2002; Golub and Ye 2002; Knyazev 2001; Baglama et al. 2003].

This recent progress is reflected in the large number of codes for symmetric eigenproblems. The survey of eigenvalue codes, [Hernandez et al. 2006], lists 20 eigenvalue codes that have become available since 1998. Eighteen of these are for symmetric eigenproblems. We are aware of only one additional eigensolver code produced since the survey, JADAMILU [Bollhöfer and Notay 2007]. There are 13 codes that implement preconditioned eigensolver methods. Only one of these was publicly available before 1998; the Davidson code in [Stathopoulos and Fischer 1994] which is now superseded by PRIMME. Most of these codes do not provide the robustness and efficiency required in a general purpose software, and therefore we refer the reader to [Hernandez et al. 2006] for more details. Instead, we discuss three notable software efforts that relate to some key characteristics of PRIMME.

Anasazi [Baker et al. 2009] is a well engineered package, with several features that enhance robustness and efficiency. Developers can use the Anasazi framework to develop their own eigensolvers. Anasazi also implements three methods: A version of the LOBPCG method [Knyazev 2001] with orthogonalization to avoid stability issues [Hetmaniuk and Lehoucq 2006; Stathopoulos and McCombs 2007]; A block Davidson method (what we refer to as Generalized Davidson) for solving standard and generalized real symmetric and Hermitian eigenvalue problems; An implicit Riemannian Trust-Region method [Absil et al. 2006] has been added recently which shares some of the properties of our JDQMR. Block methods are used to increase robustness for obtaining multiple eigenvalues [McCombs and Stathopoulos 2006] and to take advantage of the increased data locality in block matrix-vector, preconditioning, and BLAS operations. Although the total number of matrix-vector multiplications increases, for appropriate block sizes this effect is usually balanced by better cache performance [Arbenz et al. 2005]. Anasazi is part of the Trilinos framework that includes highly optimized linear algebra operations, although users may define their own matrix, preconditioner, or basic multivector operations.

Currently, Anasazi does not include the near-optimal GD+k method and the trust-region method cannot be used for interior eigenvalues. The difference in convergence of GD+k over LOBPCG and GD can be substantial, if the preconditioner is not powerful enough to converge in a few iterations [Stathopoulos 2007; Stathopoulos and McCombs 2007; Stathopoulos and Saad 1998]. This is also evident in the comparisons in Section 5. Also, despite the 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, or because some algorithmic options need to be implemented by the end user. PRIMME offers the alternative of a stand-alone design with a multitude of ready-to-use features and a choice of several near-optimal methods.

SLEPc [Hernandez et al. 2005] is a library rather than an implementation of a single method. As an extension to the popular PETSc toolkit [Balay et al. 1999], SLEPc inherits a variety of tuned data structures, multivector operations, matrix-vector and preconditioning operators, but it cannot run as stand-alone with applications that do not use PETSc. SLEPc implements several basic and some advanced methods for solving standard and generalized Hermitian and non-Hermitian eigenvalue problems. An interesting 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. When these features are included, SLEPc can be a powerful experimentation testbed.

JADAMILU is a FORTRAN77 implementation of JDCG [Notay 2002] that extends it with elements of our JDQMR method and couples it with the MILU preconditioner [Bollhöfer and Saad 2006]. JADAMILU appeared shortly after PRIMME and was the second package to implement a nearly optimal method. It is distributed in a pre-compiled library format for certain Unix architectures. Its distinct feature is that it is tightly coupled with the MILU preconditioner which is computed dynamically as the iteration accrues more eigen-information. While the combination can be powerful, it is also less general as some applications may require different preconditioners. It also makes it difficult to compare against other software.

The Anasazi and SLEPc packages come close to a robust, efficient, general purpose code, but they do not offer all the state-of-the-art methods and could be better viewed as platforms for further development. In JADAMILU the near-optimal method comes constrained by a provided preconditioner and a distribution which is far less general purpose. Our goal in this software project has been to bring state-of-the-art methods and expertise from "bleeding edge" to production.

3. DEVELOPING ROBUST, NEAR-OPTIMAL METHODS AND SOFTWARE

We first outline the ideas behind the two near-optimal methods that form the basis for the PRIMME framework. Theoretical details and further comparative discussion can be found in [Stathopoulos 2007; Stathopoulos and McCombs 2007].

3.1 The JDQMR and GD+k methods

For eigenvalue problems, the Newton method for minimizing the Rayleigh quotient on the Grassmann manifold is equivalent to Rayleigh Quotient Iteration (RQI) [Edelman et al. 1998]. When the linear system at every step is solved iteratively beyond some level of accuracy, the overall RQI time increases. Inexact Newton methods attempt to balance good convergence of the outer Newton method with an inexact solution of the Hessian equation, but this is not straightforward for RQI (see [Simoncini and Eldén 2002] and references therein).

A more appropriate representative of the inexact Newton minimization is the Jacobi-Davidson method [Sleijpen and van der Vorst 1996]. 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:

$$(I - \mathbf{u}^{(m)}\mathbf{u}^{(m)T})(A - \eta I)(I - \mathbf{u}^{(m)}\mathbf{u}^{(m)T})\mathbf{t}^{(m)} \;=\; -\mathbf{r}^{(m)} = \theta^{(m)}\mathbf{u}^{(m)} - A\mathbf{u}^{(m)}, \quad (1)$$

where η is a shift close to the wanted eigenvalue. The next Newton iterate is then $\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} + \mathbf{t}^{(m)}$. This pseudoinverse is preferable because it avoids stagnation $\mathbf{t}^{(m)} = \mathbf{u}^{(m)}$ when $\eta = \theta^{(m)}$, and singularity when $\eta \approx \lambda$. Moreover, it applies not on $\mathbf{u}^{(m)}$ but on the residual which is the gradient of the Rayleigh quotient. See [Absil et al. 2006] for some theoretical differences between Newton variants.

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

The challenge in JD is to identify the optimal accuracy to solve each correction equation. In [Notay 2002], 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 [Stathopoulos 2007], we proposed JDQMR that extends JDCG by using symmetric QMR (QMRs) [Freund and Nachtigal 1994] 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 argued that JDQMR converges less than three times and typically significantly less than two times slower than the optimal method. As optimal we consider the unrestarted GD method with the same preconditioner — without preconditioning this is the Lanczos method in exact arithmetic. JD methods are typically used with subspace acceleration, where the iterates $\mathbf{t}^{(m)}$ are accumulated in a search space from which eigenvector approximations are extracted through some projection technique [Paige et al. 1995; Morgan 1991; Jia 1998]. This further improves convergence, especially when looking for many eigenpairs. Overall, JDQMR has proved one of the fastest and most robust methods for $numEvals = 1$.

When seeking many eigenvalues, applying the Newton method on the $numEvals$ dimensional Grassmann manifold, computes directly the invariant subspace [Sameh and Tong 2000; Absil et al. 2002]. Practically, this is just a block JD method [Stathopoulos and McCombs 2007]. 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 [Geus 2002]. In our experience with block JD methods, single-vector versions outperform their block counterparts both in execution time and matvecs. Only the block JDQMR seems to be close to its 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 several unnecessary matvecs.

Yet, a block size larger than one may be needed for robustness, especially in the presence of multiplicities, because JDQMR may converge out of order. Alternatively, to avoid stopping early and thus missing eigenvalues, one may ask for a few more eigenvalues than needed.

With large $numEvals$, the near-optimal convergence of the JDQMR has to be repeated $numEvals$ times; each for an independent inverse iteration run. Larger subspace acceleration does not improve the convergence rate of QMR but approximates better the nearby eigenpairs to be targeted next. In the extreme, the optimal unrestarted GD relies only on subspace-acceleration and no inner iterations. This can be considered a full-memory quasi-Newton method which is computation-

ally impractical. Instead, variants of the non-linear Conjugate Gradient (NLCG) method have been popular for decades [Edelman et al. 1998].

However, it is natural to consider a method that minimizes the Rayleigh quotient not only along the conjugate direction but on three dimensions. The method: $\mathbf{u}^{(m+1)} = \text{RayleighRitz}\left(\{\mathbf{u}^{(m-1)}, \mathbf{u}^{(m)}, \mathbf{r}^{(m)}\}\right)$ is often called locally optimal Conjugate Gradient [D'yakonov 1983; Knyazev 1991], or LOBPCG if used with multivectors, and seems to consistently outperform other NLCG type methods.

Because of the non-linearity of the eigenproblem, LOBPCG is not optimal. In [Stathopoulos 2007] we argued that subspace acceleration speeds up LOBPCG similarly to the way that quasi-Newton methods speed up NLCG by using the iterates to incrementally construct an approximation to the Hessian [Gill et al. 1986]. In the extreme, the need for a three-term recurrence is obviated if all iterates are stored. Therefore, we could view the LOBPCG recurrence not as driving the iteration but as a means to remember the appropriate subspace information when thick restarting methods such as Lanczos or GD. This is the basis for the GD+k method [Murray et al. 1992; Stathopoulos and Saad 1998; Stathopoulos 2007].

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 required) Ritz values and their Ritz vectors, $\mathbf{u}_i^{(m)}, i = 1, m_{min}$, and also k of the corresponding Ritz vectors from step $m-1$: $\mathbf{u}_i^{(m-1)}, i = 1, k$. A basis for this set of $m_{min} + k$ vectors, which can be made orthonormal in negligible time, becomes the restarted basis. A JD+k implementation is identical. If we use GD/JD with block size b, it is advisable to keep $k \geq b$ to maintain good convergence for all block vectors. Note also that the special case of block GD(b, $3b$)+b is mathematically equivalent to LOBPCG with block size b.

As we showed in [Stathopoulos 2007], convergence of the GD+k is appreciably faster than LOBPCG for one eigenpair, even with a small subspace, *and often indistinguishable from optimal.* For large *numEvals* the convergence gains over LOBPCG are even greater [Stathopoulos and McCombs 2007]. Yet, higher costs per iteration 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/preconditioning operations. 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 Building a flexible multimethod framework in PRIMME

In [Stathopoulos 2007] 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 I 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 eigenvalues, 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

Table I. 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 (array $n \times m_{max}$)
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
$blockConv$	the number of vectors converged in the current block
$numConv$	the number of vectors converged since the last iteration
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
H	$H = V^T AV$ the projection matrix in Rayleigh-Ritz
S, Θ	the eigenvectors and eigenvalues of H
X	on input any initial guesses, on output the eigenvectors $\mathbf{x}_i$

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 bookkeeping. PRIMME includes a host of other features and handling of special cases for basis size, block size, irregular eigenvalue convergence, the handling of multiple user-defined shifts, and many others which are impractical to describe in one algorithm. Section 3.3 describes the most important and innovative ones.

Algorithms 3.2–3.4 describe the PRIMME implementation of three important components of GD+k: convergence checking and forming the target block, restarting, and locking converged eigenpairs. The restart procedure in particular (Algorithm 3.3) combines thick restarting [Stathopoulos et al. 1998] 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 $k \leq b$ is not only sufficient but also obviates the use of larger m_{min} [Stathopoulos and Saad 1998].

What characterizes Algorithm 3.1 is its flexibility. It allows for complete freedom on 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 returns $\mathbf{t}^{(m)} = \mathbf{r}^{(m)}$ (and with $k = 0$), it is equivalent to an expensive IRL implementation. With $k \geq 0$, however, GD+k should be preferable to 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 [Baglama et al. 2003; Parlett 1998], subspace iteration (without preconditioning) [Clint and Jennings 1970], and preconditioned subspace iteration [Bai et al. 2000]. We also derive the classical

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

/ Initialization */*
Initial guesses are in X. Let $m = \min(m_{min}, size(X,2))$, $\mathbf{v}_{0:m-1} = X_{0:m-1}$
Build ($m_{min} - m$) Lanczos vectors to get basis $V = [\mathbf{v}_0, ..., \mathbf{v}_{m_{min}-1}]$
Set $W = AV$, $H = V^T W$, $m = n_{mv} = m_{min}$, $q = l = 0$
Compute eigendecomposition $H = S\Theta S^T$ with $\theta_0, \theta_1, \ldots, \theta_{m-1}$ sorted according to user defined criteria (smallest, largest, interior)
/ Repeat until convergence or max number of matvecs */*
while ($l < numEvals$ **and** $n_{mv} < max\_num\_matvecs$)
  / Repeat until basis reaches max size or it spans the whole space */*
  while ($m < m_{max}$ **and** $m < n - l$)
    Reset b, if needed, so that $m + b \leq m_{max}$
    $\mathbf{u}^{(m)} = Vs_{0:b-1}$, $\theta^{(m)} = diag(\theta_{0:b-1})$, $\mathbf{w}^{(m)} = Ws_{0:b-1}$, $\mathbf{r}^{(m)} = \mathbf{w}^{(m)} - \mathbf{u}^{(m)}\theta^{(m)}$
    Check convergence and determine target block. See ALGORITHM 3.2
    if ($l + q \geq numEvals$), **break**
    Precondition the block of residuals: $\mathbf{t}^{(m)} = Prec(\mathbf{r}^{(m)})$
    Orthonormalize $\mathbf{t}^{(m)}$ among themselves and against $\mathbf{v}_{0:m-1}$ and $\mathbf{x}_{0:l-1}$
    Update $\mathbf{v}_{m:m+b-1} = \mathbf{t}^{(m)}$, $\mathbf{w}_{m:m+b-1} = A\mathbf{v}_{m:m+b-1}$, $n_{mv} = n_{mv} + b$
    Update $H_{i,m:m+b-1} = \mathbf{v}_i^T \mathbf{w}_{m:m+b-1}$ for $i = 0, \ldots, m + b - 1$
    Store Ritz vector coefficients: $s_i^{old} = s_i$, $i = 0, \ldots, \max(b, k) - 1$
    $m = m + b$
    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)
  end while
  Restart the basis and reset variables. See ALGORITHM 3.3
  Lock the q flagged Ritz pairs into λ and X. See ALGORITHM 3.4
end while

ALGORITHM 3.2. *Check convergence and determine target block*

numConv = 0
repeat
  blockConv = Number of converged vectors in block $\mathbf{r}^{(m)}$ consider also practically converged vectors (see ALGORITHM 3.5*)*
  numConv = numConv + blockConv
  q = q + blockConv. Flag these Ritz vectors as converged
  Find the next blockConv unconverged Ritz vectors $\mathbf{u}_i^{(m)}$ and their residuals $\mathbf{r}_i^{(m)}$ to replace the blockConv ones in the block
  Swap converged vectors after the block to maintain block locality
until *(b unconverged residuals are in the block* **or** *not enough available)*
if *(not enough unconverged residuals) reduce block size*

block GD and the block GD+k methods [Liu 1978; Stathopoulos and Fischer 1994; Geus 2002; Stathopoulos and Saad 1998]. GD(b,3b)+b is a numerically stable implementation of LOBPCG that maintains full orthogonality of the basis for only 5% more floating point operations per iteration [Stathopoulos 2007; Hetmaniuk and Lehoucq 2006]. A smaller b can be chosen independently from $numEvals$ to obtain locking implementations of LOBPCG, while variations such as GD(b, mb)+b are

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 $H_{sub} = (s^{old})^T H s^{old}$ ($k \times k$ 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) $H = \mathbf{0}$. *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* $W_{m-g_q:m-1} = A v_{m-g_q:m-1}$, $n_{mv} = n_{mv} + g_q$
(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,* $q = 0$

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 this preconditioner can be written as:

$$\left((I - QQ^T)K(I - QQ^T)\right)^+ \ = \ K^{-1}(I - Q(Q^T K^{-1} Q)^{-1} Q^T K^{-1})(I - QQ^T). \quad (2)$$

The above is known as Olsen's method [Olsen et al. 1990]. "Robust shifting"

[Stathopoulos et al. 1995] 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 (2) is used in an iterative method on (1), we obtain the classical inner-outer JD variants. In [Fokkema et al. 1998; Sleijpen and van der Vorst 1996] it is shown that JD methods can be implemented with one projection with Q per iteration. If the inner iteration solves (1) accurately, we obtain 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 (1) is solved approximately.

For the reasons described in Section 3.1, at step (10) of the underlying GD+k outer method, PRIMME calls the symmetric, right preconditioned QMRs as a robust inner solver for all JD variants. The JDQMR's particular scalar recurrences for monitoring the eigenvalue residual and the dynamic stopping criteria have been described in detail in Algorithms 3.1 and 3.2 in [Stathopoulos 2007].

3.3 Special techniques in PRIMME

The state-of-the-art algorithms and the myriad of their combinations are complemented by several techniques that provide additional efficiency and robustness.

3.3.1 *Avoiding the JD oblique projectors.* The classical JD requires extra storage for $K^{-1}Q$ to avoid doubling the number of preconditioning operations. In [Stathopoulos and McCombs 2007] we have shown that the pseudoinverse (2) with only $Q = [\mathbf{u}^{(m)}]$, without X, is sufficient. Intuitively, projecting out $\mathbf{u}^{(m)}$ from a very accurate preconditioner helps avoid the problem where the correction is almost completely in the direction of $\mathbf{u}^{(m)}$ [Notay 2005]. By projecting out X one would hope for a better conditioned correction equation. Instead, our analysis showed that the correction equation without the X-projected pseudoinverse is often better.

For large $numEvals$, avoiding the X projection yields significant storage savings, effectively halving the memory required by JD. 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 in the notation of A and K. Define the orthogonal projector $P = I - QQ^T$, and for any matrix B the skew projector:

$$P_B \ = \ (I - BQ(Q^TBQ)^{-1}Q^T), \tag{3}$$

and note that the correction equation preconditioned with (2) can be written as:

$$PAP(PKP)^+ \ = \ PAP_{K^{-1}}K^{-1} = PAK^{-1}P^T_{K^{-1}}. \tag{4}$$

Table II 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 a right skew projector, replacing it with P. For example, the case $PAP_{K^{-1}}K^{-1} = PAK^{-1}P^T_{K^{-1}}$, which corresponds to notation (111), requires one orthogonal and one skew projection per QMR step. The suggested implementation by the JD authors is $AP_{K^{-1}}K^{-1} = AK^{-1}P^T_{K^{-1}}$, or (011). Our default strategy with preconditioning is (100), or PAK^{-1}, with only one left orthogonal projection.

An important benefit occurs when K has the same eigenvectors as A (e.g., if K is a polynomial of A or $K = I$). Then, the QMR iterates stay in $X^\perp$ invariant

Table II. Projection alternatives to the classical JD 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 JD 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}

space without orthogonalization. Floating point arithmetic and the approximate computation of X eventually introduce X components that QMR has to remove by additional iterations. However, this is a small price to pay for removing the limiting factor $O(numEvals^2 n)$ of orthogonalization. In our experience, unpreconditioned JDQMR-000 achieves an almost linear scaling with $numEvals$, both in convergence and in time, which is the best we can currently achieve under limited memory.

3.3.2 *Avoiding stagnation because of locking.* Locking is a stable form of deflation, where an eigenvector $\mathbf{x}$ is removed from the search space of an eigensolver and all subsequent vector operations are performed orthogonally to $\mathbf{x}$. Locking usually provides a better mechanism than non-locking for identifying eigenvalues that are highly clustered or of very high multiplicity [McCombs and Stathopoulos 2006].

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 report [Stathopoulos 2005]. 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. In [Stathopoulos 2005] we have provided an algorithm that identifies the problem when it occurs. Its variant as implemented in PRIMME appears in Algorithm 3.5. The interesting theoretical result is that a "practically converged" eigenvector can still be locked, because enough of its missing components are in X and can be recovered by a single Rayleigh-Ritz projection at the end. The resulting Ritz vector will have residual norm less than the required tolerance.

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 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: the choice between GD+k and JDQMR. GD+k converges

ALGORITHM 3.5. *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 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

almost identically to the optimal method, while JDQMR may repeat some 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], but the cost per iteration of JDQMR is significantly lower than GD+k. The crossover point between the two methods depends on the expense of the matrix and preconditioner operators, on $numEvals$, and on the slowdown of the JDQMR convergence relative to GD+k.

In [Stathopoulos and McCombs 2007], 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 flop counts. Such components include the operators, 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 enables a dynamic prediction of the execution time of GD+k and JDQMR based on runtime measurements 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 3.6 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.

[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.

First, by running solely with GD+k we cannot measure the cost or predict the convergence rate of JDQMR. By running solely with JDQMR, we can still update the GD+k cost but not its convergence rate. 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 JD 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 converges to an eigenpair in a small number of outer iterations (often less than $m_{max} - m_{min}$). 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 an expensive matrix operator, our dynamic algorithm should not solve another correction equation. Therefore, we must evaluate JDQMR at every outer step, just before calling the correction equation.

Fourth, in addition to the above, 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 similar problems need to be solved repeatedly. Algorithm 3.6 summarizes these decisions for dynamic method switching.

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 may even stagnate or produce spurious "eigenpairs". PRIMME uses a variation of the classical Gram-Schmidt with iterative reorthogonalization [Daniel et al. 1976]. If after two iterations an additional orthogonalization is needed, the vector has lost all significant components of the original direction and we replace it with a random

ALGORITHM 3.6. *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
        endif
```

vector. This also guards against vectors that are zero or close to machine precision.

On parallel computers, one of the factors limiting scalability is the presence of several dot products in Gram-Schmidt. We have implemented a not so well known strategy that typically removes one dot product per Gram-Schmidt iteration. Similar techniques have been used for Gram-Schmidt in various contexts [Stathopoulos and Fischer 1993; de Sturler and der Vorst 1995; Hernandez et al. 2007]. Let $\mathbf{r}$ be the vector to be orthogonalized against V and $s_0 = \|\mathbf{r}\|$. After orthogonalization $\mathbf{r}' = (I - VV^T)\mathbf{r}$, we reorthogonalize if $s_1 = \|\mathbf{r}'\| < 0.7071 s_0$. It is possible to avoid

the expense of the dot product to compute s_1 and its synchronization. Note that:

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

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 $s_1 \geq 0.7071 s_0$, the resulting vector can be normalized as $\mathbf{r}'/s_1$, 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 second reorthogonalization test to fail and return a vector that is neither unit-norm nor orthogonal. A simple error analysis of the computation provides the following interesting (and to our knowledge new) result:

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

where $\hat{s_1}$ is the floating point representation of 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 $s_1 > s_0\ \sqrt{\epsilon_{machine}}$. 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 that the numerical danger is real and that our test restores robustness and results in improved efficiency. Having ensured that Gram-Schmidt produces a numerically orthonormal V, V will stay orthonormal throughout PRIMME. This is because the only other place that modifies V is restarting, $V = Vs$, which involves orthonormal matrices and thus it is stable.

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 they converged, so that they are excluded from the target block. Still, they participate in the Rayleigh-Ritz at every step and therefore improve as additional information is gathered in V. Typically overall convergence is faster. However, it is possible that a Ritz vector $\mathbf{x}_i$ that was flagged converged becomes unconverged during later iterations. This could occur if eigenpairs converge out of order or 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, $W = AV$ and H are recomputed, 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 Computational requirements

3.4.1 *Memory.* The memory requirements for the GD, JD, and symmetric QMR methods are established in the literature [Barrett et al. 1994; Bai et al. 2000]. The combination of these basic methods in PRIMME through various parameter choices determines the actual memory requirements. PRIMME requires that the user provides an array where the computed eigenvectors will be placed. Without

locking, and for those methods 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 significantly reduces storage. For general users, we do not yet recommend this undocumented feature. The table below outlines the memory footprint of the basic outer, GD+k method:

$2m_{max}n$	storage for V and W
$2m_{max}^2$	storage for H and S
m_{max}k	storage for the s^{old}
$\max(\text{k}^2, m_{max}\text{k}, 2b(numEvals + m_{max}))$	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 a small work space of $2numEvals + m_{max} + 2b$. JDQMR with skew projection on $\mathbf{u}^{(m)}$ (i.e., methods JDQMR-000, -100, -001, and -101 in Section 3.3.1) requires one additional vector (n). JDQMR with skew projector on both $\mathbf{u}^{(m)}$ and X (i.e., methods JDQMR-110, -011, and -111) 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 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, classical RQI requires $m_{max} = 2$ for a total memory of $O(9n)$. Similarly, implementing LOBPCG as GD(b,3b)+b requires a total memory of $O(6bn)$.

3.4.2 *Complexity.* The complexity of any method in PRIMME is the sum of the complexities of the outer and the inner iteration components. The only additional parameter is the relative frequency that a method spends in each component. A detailed complexity analysis was carried out in [Stathopoulos 2007; Stathopoulos and McCombs 2007]. For completeness, we present the resulting model based on floating point operation counts for GD+k and JDQMR. Considering the complexity of each PRIMME component at some basis size m, we average over $m = m_{min} \ldots m_{max}$ and by setting $\mu = \frac{m_{min}}{m_{max}}$, we have the average cost per outer step:

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

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. Clearly, for a large number of locked vectors, l, the orthogonalization dominates the iteration costs.

Currently, the correction equation for each block vector is solved independently, so it suffices to obtain the cost for each QMR step, that includes also two projections with $\mathbf{u}^{(m)}$. For JDQMR-000 the cost is $25n$, and for JDQMR-100 ($25n + 4nl$).

Note that the QMRs Algorithm has $24n$ 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. 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 [Notay 2002] would cost $24n$ operations, but the slightly additional cost of JDQMR is justified by the increased robustness, flexibility, and ability to derive better stopping criteria.

4. THE PRIMME SOFTWARE

Our target is an eigenvalue code that is as close as possible to "industrial strength" standards. We have described a long list of methods, techniques, and specialized algorithms that have been implemented in a unified framework in PRIMME. In this section, we address implementation efficiency, rich functionality, and a flexible but simple user interface.

4.1 Choice of language and implementation

The underlying ideas for the basic structure of PRIMME have evolved starting from the FORTRAN 77 code DVDSON [Stathopoulos and Fischer 1994] and are loosely based on our early FORTRAN version of GD+k/Jacobi-Davidson, DJADA, which we circulated in 2000. A project of this proportion needed to be engineered around a more flexible language, for this reason we have chosen the C language.

Our primary reasons for choosing C are: the flexibility it allows for the user interface and parameter passing, its interoperability, its cleaner memory management, and its efficiency. A PRIMME type structure contains 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 interprocess communication primitives, 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. Being the primary language for systems programming, C easily interoperates with C++, FORTRAN 77, Fortran 90, Python, and many other scripting languages and environments (such as MATLAB and Mathematica), and thus could help PRIMME achieve a broader impact in the community. We have opted not to use C++ which might have been a better choice if PRIMME were tightly coupled with a broader problem solving environment, as opposed to being a stand alone, general purpose package. Finally, if properly implemented, C codes are as efficient as any other language, although the brunt of computation in PRIMME is handled by calls to BLAS and LAPACK functions which are typically machine optimized.

On the technical side of the implementation, users can poll PRIMME for the required memory and provide their own workspace or let PRIMME allocate memory internally. To avoid allocate/free overheads, PRIMME allocates all required memory as one contiguous work array upon initialization and subdivides the work array by assigning pointers to different portions of the array. 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 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 when allocating small amounts of data it may waste large fractions of a page. In the case of PRIMME 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 performing the 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. To minimize communication and the number of parallel primitives the user has to provide, we assume a homogeneous parallel machine where identical computations on each processor are expected to yield exactly the same floating point number. Heterogeneous processors would have required in addition a broadcast function. PRIMME can also benefit from multithreading or special purpose hardware by linking to BLAS and LAPACK libraries that are optimized appropriately. Finally, the user must 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 – 9.7.1 on PowerPC and Intel, 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.

4.2 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 parameters, but also of the names of the methods. More experienced users should be able to use additional functionality to match their specific needs. PRIMME caters also to expert users who might also use the code to experiment with new techniques, combinations of methods, etc. Such combinations involve only adjusting an extensive set of parameters rather than implementing new components, which would be primarily the case with the Anasazi/SLEPc frameworks.

Figure 1 shows a minimal interface required by PRIMME. Users must declare a parameter of type `primme_params` that holds all solver information and is used for input and some output. Although not strictly required, a call to our initializa-

```
#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. 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.

tion function is strongly recommended to avoid undefined parameters. 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 non-expert users, we provide three generic method choices `DEFAULT_MIN_MATVECS` (which defaults to GD+k), `DEFAULT_MIN_TIME` (defaults to JDQMR_ETol), and `DYNAMIC` (dynamically finds the best of the first two). If a preconditioning operator is available, it should be set before setting the method as:

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

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

```
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. The multivectors store individual vectors consecutively in memory. To allow a common interface across C and FORTRAN languages and between different precisions we use by reference (`void *`) arguments.

The minimal PRIMME interface makes heavy use of defaults. For example, the above segment 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+k and JDQMR, using $m_{min} = 6$, $m_{max} = 15$, $b = 1$, and $k = 1$. We emphasize that, despite the simplicity of the interface, the defaults and the methods reflect expertly tuned, near-optimal methods. In fact, the above code segment for finding the smallest eigenvalue of difficult problems has matched or outperformed all other software we are aware of.

Users can have far more control over the eigenvalue problem than this minimal interface provides. This control is achieved through a detailed interface that allows the user to specify parameters for the problem to be solved: the number of eigenvalues, where these eigenvalues are located (extreme or interior), the exact

residual norm convergence tolerance, the number of available initial guesses, the maximum number of matvecs, and the operators. None of the above parameters determines any algorithmic features, so these are parameters 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.

PRIMME provides several preset methods, including various versions of Jacobi and Generalized Davidson, RQI, Inverse iteration, LOBPCG, and subspace iteration. The different versions of the methods differ on how they employ preconditioning, deflation, subspace acceleration, locking, and block size. We briefly describe the default methods that set PRIMME apart from other eigensolvers.

The method `GD_Olsen_plusK`, which serves as the `DEFAULT_MIN_MATVECS` method, 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. The `JDQMR_ETol` method, which serves as the `DEFAULT_MIN_TIME` method [Stathopoulos 2007; Stathopoulos and McCombs 2007], improves performance over `JDQMR` by terminating the solve of the correction equation when the eigenvalue residual (not the linear system residual) is reduced by an order of magnitude. This method avoids recomputing QMRs information between outer steps which often causes an increase in run time of the original JDQMR solver. The `DEFAULT_MIN_MATVECS` method typically yields the smallest number of matvec operations over all methods, and the `DEFAULT_MIN_TIME` method spends the least amount of time in the eigensolver while its number of matvecs is usually within a factor of 1.5 of `DEFAULT_MIN_MATVECS`. The `DYNAMIC` method alternates between the above two according to Algorithm 3.6.

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

To facilitate portability and usability, we have provided a FORTRAN 77 interface that allows FORTRAN users to set all the members of the primme structure, set methods, and call the PRIMME interface functions. For a detailed explanation of the solver parameters, supported methods, and the FORTRAN interfaces, we refer the reader to the distributed documentation.

4.3 Additional special features

We briefly mention a few features that improve robustness and 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. Users may select to target the smallest or largest eigenvalues, or they may target interior eigenvalues in one of three ways with respect to a set of shifts: closest in absolute value to the shifts, closest to and greater than or equal to the shifts ($\geq$), or closest to and less than or equal to the shifts ($\leq$). User provided shifts are also used in the JD correction equation as approximations to the target eigenvalues until additional eigenvalue in-

formation is obtained. We note that the functionality of the last two options ($\geq$ and $\leq$), which is useful in certain applications, cannot be obtained from the "Smallest Magnitude" option that is common in packages such as Anasazi. In principle, these options could be programmed in Anasazi through its abstract sorting interface, but the implementation burden would be on the user.

Second, the `primme` structure supplies an array of shifts (the robustly shifted Ritz values $\tilde{\lambda}_i, i = 1, b$) corresponding to the vectors in the block to be preconditioned. Many applications can afford to invert the preconditioner at every step. Examples include diagonal matrix preconditioners, such as the FFT transform of the Laplace operator in planewave space, or when the preconditioner is an iterative method. In these cases, it is preferable to use $(K - \tilde{\lambda}_i I)^{-1} \approx (A - \tilde{\lambda}_i I)^{-1}$. Providing robust shifts to the user to improve the preconditioner is a unique feature of PRIMME.

Third, PRIMME explicitly provides for a set of orthogonality constraints, i.e., it can solve for the nonzero eigenvalues of $(I - QQ^T)A(I - QQ^T)$, where Q are previously computed eigenvectors. PRIMME assumes Q is included in the first vectors of the `evecs` input array. Computed eigenvectors will be placed after Q. This is cleaner and more stable than expecting the user to use a deflated matrix.

Finally, we mention that PRIMME includes two simultaneously callable libraries for real and complex matrices, 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. Such a level of reporting is required for meeting the standards of industrial strength software.

5. SAMPLE EXPERIMENTAL RESULTS

In [Stathopoulos 2007] and [Stathopoulos and McCombs 2007] we have presented performance results for one of the most extensive comparisons between sparse Hermitian and real symmetric eigensolvers in the literature. PRIMME was compared with three other software packages: JDBSYM, BLOPEX, and ARPACK's `dsaupd` function, significantly outperforming them when looking for a few eigenvalues. For several hundreds of eigenvalues, PRIMME even outperformed ARPACK, the default benchmark for unpreconditioned cases, when the matrix is sufficiently sparse. The results testified not only for the quality of the algorithms but also for the efficiency of the implementation.

In this paper, we provide only a sample update of experimental results by comparing with the Anasazi package, as a representative of another state-of-the-art code, and showing the effectiveness of two of PRIMME's unique features: finding interior eigenvalues ($\geq$ shift) and the dynamic method. We do not perform comparisons with SLEPc methods as they do not natively implement any near-optimal methods, nor with JADAMILU because it is tightly coupled with the MILU preconditioner. We plan to make such comparisons in the future.

5.1 Anasazi comparisons for exterior and interior eigenvalues

We use the DEFAULT_MIN_TIME (JDQMR) and DEFAULT_MIN_MATVECS (GD+k) methods with the default parameters provided by PRIMME and no other fine tuning. This yields a basis size of 15 for both methods. We look for five algebraically smallest eigenvalues. For convergence we require a residual tolerance

Table III. The matrices used in the experiments.

Matrix	n	nonzero	Source	Matrix	n	nonzero	Source
Lap7p1M	941192	6530720		nd3kf	9000	3279690	[Yang et al. 2001]
Andrews	60000	760154	[Davis]	or56f	9000	2890150	[Yang et al. 2001]
finan512	74752	596992	[Davis]	Fillet13K_A	13572	632146	[Adams 2002]
cfd1	70656	1825580	[Davis]	Cone_A	22032	1433068	[Adams 2002]
cfd2	123440	3085406	[Davis]	Plate33K_A	39366	914116	[Adams 2002]

Table IV. PRIMME (top two) and Anasazi (bottom four) methods for finding 5 smallest eigenvalues without preconditioning. For each matrix, matrix vector multiplications and run time is reported. LOBPCG uses $b = 5$ and LOBPCG[1] and BD[1] use $b = 1$. [2] block size of 2 was needed to avoid missing one of the three multiple eigenvalues. [3] stopped with orthogonalization problems.

matrix:	Andrews		Cone_A		Lap7p1M		Plate33K_A		finan512	
method	MV	Sec	MV	Sec	MV	Sec	MV	Sec	MV	Sec
JDQMR	768	4.1	793	3.7	2442	171	1431	6	1109	5
GD+k	568	7.8	588	4.3	2237	[2]418	1065	10	936	15
IRTR	2620	25.3	3140	11.9	4630	582	4800	27	5340	56
LOBPCG	1360	22.6	1765	11.8	2850	644	2945	31	2260	45
LOBPCG[1]	2291	28.2	2435	17.6	4478	793	4256	36	5228	72
BD[1]	2742	34.6	3012	21.6	5680	908	41556	509	3538	49

matrix:	cfd1		cfd2		nd3kf		or56f	
method	MV	Sec	MV	Sec	MV	Sec	MV	Sec
JDQMR	4650	35	23107	299	7821	70	7616	60
GD+k	4383	81	21221	668	8124	90	6760	68
IRTR	29090	320	107140	1852	31415	147	39810	170
LOBPCG	10610	209	59085	2045	—[3]		14570	88
LOBPCG[1]	17724	282	62289	1768	100009	1081	27913	273
BD[1]	65762	1033	294101	7783	294100	3486	294100	3197

of $\|A\|_F 10^{-10}$, where $\|A\|_F$ is the Frobenius norm of A. For Anasazi we use three methods: IRTR, LOBPCG, and BD (block Davidson). Convergence thresholds are also $\|A\|_F 10^{-10}$ without a relative tolerance requirement. We can only control the basis size of BD which is set to 50 vectors. LOBPCG and BD use locking, LOBPCG uses full orthogonalization, and restarting is performed in-situ for BD. We run experiments on a 2.8 GHz Intel Core 2 Duo MacBook Pro with 6MB of L2 cache and 4 GB of memory. The suite of gcc-mp-4.2, g++-mp-4.2 and gfortran-mp-4.2 compilers is used for both packages with -O3 optimization flag. We link with Apple's vecLib library which includes optimized versions of BLAS/LAPACK. We experiment with ten matrices shown in Table III. Lap7p1M is a 7 point finite difference Laplacian on the unit 3-dimensional cube with Dirichlet conditions.

Table IV shows the results from finding five algebraically smallest eigenvalues of nine of the above matrices without preconditioning. Even though the IRTR is based on the same optimality principles as JDQMR, its implementation details as well as the fact that it requires $b \geq numEvals$ make it substantially slower in convergence. The PRIMME methods yield also by far the shortest times and make the solution of even very tough problems feasible. Because of fast convergence, the default PRIMME methods may find exact multiple eigenvalues out of order. Although this is rare, we include one example (Lap7p1M) when it occurs for GD+k. In this case, a block size of two is sufficient to resolve the triplet eigenvalue.

Table V. Finding interior eigenvalues on the right of σ for two diagonal matrices.

	JDQMR		GD+k		IRTR	
Diagonal with 1-D Laplace evals	MV	Sec	MV	Sec	MV	Sec
5 evals ≥ 0.01	34702	198	27355	1292	N/A	
10 abs smallest of $(A-0.01I)$	34452	206	39069	1932	N/A	
10 smallest of $(A-0.01I)^2$	749557	2080	835724	16892	4469920	68370
Diagonal with denser evals on left	MV	Sec	MV	Sec	MV	Sec
4 evals ≥ 10.015	50521	47	24771	169	N/A	
14 abs smallest of $(A-10.015I)$	146734	122	57161	438	N/A	
14 smallest of $(A-10.015I)^2$	292418	70	282474	641	679742	1022

In the second experiment, we look for interior eigenvalues of two artificial diagonal matrices. The first contains the eigenvalues $(4\sin(\frac{k\pi}{(2(n+1))}))$ of the 1-D Laplacian of dimension 100000. We look for five eigenvalues on the right of $\sigma = 0.01$. For PRIMME, we can use the option for $\geq \sigma$. Alternatively, the five required eigenvalues can be obtained if we find the 10 eigenvalues closest in absolute value to σ. Finally, we can look for the 10 smallest eigenvalues of $(A-\sigma I)^2$, but with tolerance $\|A\|_F 10^{-14}$ which is needed to distinguish the squared eigenvalues.

The second matrix contains diagonal entries: $A_{ii} = i0.01$, if $i \leq 1000$ and $A_{ii} = 1+(i-1000)0.03$, if $1000 < i \leq 10000$. We look for four eigenvalues on the right of $\sigma = 10.015$. Because of the asymmetrical eigenvalue distribution on either side of σ, we need 14 eigenvalues if we instead opt to find the ones closest in absolute value to σ or the smallest of $(A-\sigma I)^2$. For this matrix, $(A-\sigma I)^2$ has the additional complication that all required eigenvalues coincide as multiples with certain ones from the left side. As before, we require $\|A\|_F 10^{-15}$ convergence tolerance except for IRTR which overconverges and thus a tolerance of $\|A\|_F 10^{-11}$ yields comparable results to PRIMME. The results are shown in Table V.

We observe that when the eigenvalues on one side of σ are denser than on the other side (which is often the case in some problems in materials science), our interior $\geq \sigma$ option is better than the traditional `'SM'` option; otherwise it is competitive. We also confirm that the matvec optimality of GD+k does not fully carry over to interior eigenvalues. Yet, both JDQMR and GD+k are significant improvements over IRTR on the $(A-\sigma I)^2$ matrix — the only way we could use Anasazi for this problem as IRTR could not be used to find interior eigenvalues, and the LOBPCG and BD methods did not converge with the `'SM'` option.

5.2 Dynamic method behavior

We tested the dynamic method in Algorithm 3.6 for switching between the GD+k and JDQMR-000 without preconditioning on the `Fillet_13K` matrix. In Figure 2 the results are shown in three graphs to keep the plots from being compressed by scale. For a small number of eigenvalues, GD+k and JDQMR-000 cross over numerous times, but the dynamic method adapts and remains close to the best-performing method at all times. For larger numbers of eigenvalues, the times for GD+k and JDQMR-000 begin to diverge and the dynamic method performs consistently with the best one.

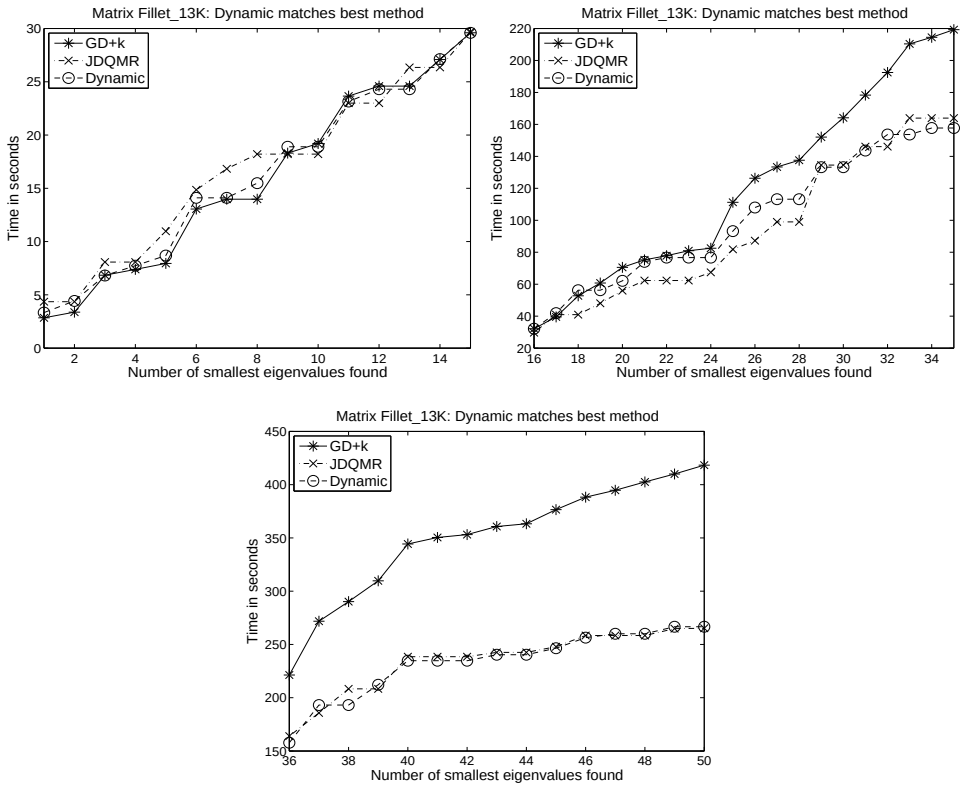

Fig. 2. Results with dynamic method switching between GD+k and JDQMR-000.

6. CONCLUSIONS AND FUTURE WORK

The PRIMME software is a unique implementation that incorporates numerous methods and features for computing selected eigenvalues of sparse Hermitian matrices. In particular, PRIMME features the state-of-the-art near-optimal methods JDQMR and GD+k. The success of these methods to perform near-optimally has been well documented. This paper has focused on the research that was necessary to realize this in a highly efficient and robust way without compromising ease of use. We have detailed the multimethod framework and described several new algorithmic techniques that contribute equally to achieving robustness and efficiency. A new locking technique avoids a classic stagnation problem, a new orthogonalization scheme reduces synchronization and computation in a provably stable way, a verification component ensures that converged eigenvectors are returned, and a new algorithm dynamically identifies and adapts to the fastest method for a particular problem. Finally, we have outlined the design and features of a multi-layer user interface that addresses the needs of both end-users and experts. Supplementing our previous extensive list of numerical evidence, a few new experiments confirm the efficiency and rich functionality of PRIMME.

PRIMME version 1.1 was publicly released in October 2006 and distributed with

a Lesser GPL license. After more than 200 unique downloads, no bugs have been reported in the library code. A list of on-going and future projects includes generalized eigenvalue problems, an SVD interface, implementing the full functionality of the Iterative Validation algorithm [McCombs and Stathopoulos 2006], and incorporating a promising new algorithm from [Stathopoulos and Orginos 2007].

Acknowledgement

The authors are indebted to the referees for their extensive, thoughtful, and helpful suggestions, and to Profs. Bai and Gladwell for their excellent editorial work.

REFERENCES

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

Absil, P.-A., Mahony, R., Sepulchre, R., and Dooren, P. V. 2002. A Grassmann-Rayleigh quotient iteration for computing invariant subspaces. *SIAM Review 44,* 1, 57–73.

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

Arbenz, P., Hetmaniuk, U. L., Lehoucq, R. B., and Tuminaro, R. S. 2005. 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.

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

Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H., Eds. 2000. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Software Environ. Tools 11, SIAM, Philadelphia.

Baker, C. G., Hetmaniuk, U. L., Lehoucq, R. B., and Thornquist, H. K. 2009. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software 36,* 3. http://trilinos.sandia.gov/packages/anasazi.

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. 1999. PETSc home page. http://www.mcs.anl.gov/petsc.

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. 1994. *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, Philadelphia, PA.

Bollhöfer, M. and Notay, Y. 2007. JADAMILU: a software code for computing selected eigenvalues of large sparse symmetric matrices. *Comput. Phys. Commun. 177,* 12, 951–964.

Bollhöfer, M. and Saad, Y. 2006. Multilevel preconditioners constructed from inverse-based ilus. *SIAM J. Sci. Comput. 27,* 5, 1627–1650.

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

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

Cullum, J. and Donath, W. 1974. 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*. 505–509.

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

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

Davis, T. University of Florida sparse matrix collection. Tech. rep., 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, 1997.

de Sturler, E. and der Vorst, H. V. Oct 1995. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics 18,* 4, 441–59.

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

Edelman, A., Arias, T. A., and Smith, S. T. 1998. The geometry of algorithms with orthogonality constraints. *SIAM Journal on Matrix Analysis and Applications 20,* 2, 303–353.

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

Fokkema, D. R., Sleijpen, G. L. G., and van der Vorst, H. A. 1998. Jacobi-Davidson style QR and QZ algorithms for the reduction of matrix pencils. *SIAM J. Sci. Comput. 20,* 1, 94–125.

Foley., J., Juge, K. J., O'Cais, A., Peardon, M., Ryan, S., and Skullerud, J.-I. 2005. Practical all-to-all propagators for lattice qcd. *Comput. Phys. Commun. 172*, 145–162.

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

Geus, R. 2002. The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities. Ph.D. thesis, ETH, Zurich, Switzerland. Thesis. No. 14734.

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

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

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

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

Gutknecht, M. H. 2005. Block Krylov space solvers: A survey. `http://www.sam.math.ethz.ch/~mhg/talks/bkss.pdf`.

Hernandez, V., Roman, J. E., and Tomas, A. 2007. Parallel arnoldi eigensolvers with enhanced scalability via global communications rearrangement. *Parallel Computing 33*, 521–540.

Hernandez, V., Roman, J. E., Tomas, A., and Vidal, V. October, 2006. A survey of software for sparse eigenvalue problems. Tech. Rep. SLEPc STR-6, Unversidad Politecnica de Valencia.

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

Hetmaniuk, U. and Lehoucq, R. B. 2006. Basis selection in LOBPCG. *J. Comp. Phys. 218*, 324–332.

Jia, Z. 1998. A refined iterative algorithm based on the block Arnoldi process for large unsymmetric eigenproblems. *Lin. Alg. Appl. 270*, 171–189.

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

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

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

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

Lehoucq, R. B., Sorensen, D. C., and Yang, C. 1998. *ARPACK User's guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA.

Liu, B. 1978. Numerical algorithms in chemistry: Algebraic methods, eds. C. Moler and I. Shavitt. Tech. Rep. LBL-8158, Lawrence Berkeley Laboratory.

Lundström, E. and Eldén, L. 2002. Adaptive eigenvalue computations using Newton's method on the Grassmann manifold. *SIAM J. Matrix Anal. Appl. 23,* 3, 819–839.

McCombs, J. R. and Stathopoulos, A. 2006. Iterative validation of eigensolvers: A scheme for improving the reliability of Hermitian eigenvalue solvers. *SIAM J. Sci. Comput. 28,* 6, 2337–2358.

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

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

Notay, Y. 2002. Combination of Jacobi-Davidson and conjugate gradients for the partial symmetric eigenproblem. *Numer. Lin. Alg. Appl. 9*, 21–44.

Notay, Y. 2005. Is Jacobi-Davidson faster than Davidson? *SIAM J. Matrix Anal. Appl. 26,* 2, 522–543.

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

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

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

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

Simoncini, V. and Eldén, L. 2002. Inexact Rayleigh quotient-type methods for eigenvalue computations. *BIT 42,* 1, 159–182.

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

Sorensen, D. C. 1992. Implicit application of polynomial filters in a K-step Arnoldi method. *SIAM J. Matrix Anal. Appl. 13,* 1, 357–385.

Stathopoulos, A. 2005. Locking issues for finding a large number of eigenvectors of Hermitian matrices. Technical Report WM-CS-2005-09.

Stathopoulos, A. 2007. Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue. *SIAM Journal on Scientific Computing 29,* 2, 481–514.

Stathopoulos, A. and Fischer, C. F. 1993. Reducing synchronization on the parallel Davidson method for the large,sparse, eigenvalue problem. In *Supercomputing '93*. IEEE Comput. Soc. Press, Los Alamitos, CA, 172–180.

Stathopoulos, A. and Fischer, C. F. 1994. A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix. *Computer Physics Communications 79,* 2, 268–290.

Stathopoulos, A. and McCombs, J. R. 2006. PRIMME: PReconditioned Iterative Multimethod Eigensolver. http://www.cs.wm.edu/∼andreas/software/.

Stathopoulos, A. and McCombs, J. R. 2007. Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part II: Seeking many eigenvalues. *SIAM Journal on Scientific Computing 29,* 5, 2162–2188.

Stathopoulos, A. and Orginos, K. June 12, 2008 (original version July 1, 2007). Computing and deflating eigenvalues while solving multiple right hand side linear systems with an application to quantum chromodynamics. Tech. Rep. arXiv.org 0707.0131v2. http://arxiv.org/pdf/0707.0131v2, accepted in SIAM J. Sci. Comput.

Stathopoulos, A. and Saad, Y. 1998. Restarting techniques for (Jacobi-)Davidson symmetric eigenvalue methods. *Electr. Trans. Numer. Anal. 7*, 163–181.

Stathopoulos, A., Saad, Y., and Fischer, C. F. 1995. Robust preconditioning of large, sparse, symmetric eigenvalue problems. *J. Comput. Appl. Math. 64*, 197–215.

Stathopoulos, A., Saad, Y., and Wu, K. 1998. Dynamic thick restarting of the Davidson, and the implicitly restarted Arnoldi methods. *SIAM J. Sci. Comput. 19,* 1, 227–245.

Yang, C., Peyton, B. W., Noid, D. W., Sumpter, B. G., and Tuzun, R. E. 2001. Large-scale normal coordinate analysis for molecular structures. *SIAM J. Sci. Comput. 23,* 2, 563–582.