JADAMILU: a software code for computing selected eigenvalues of large sparse symmetric matrices

Matthias Bollhöfer [1]

Institut "Computational Mathematics", Technische Universität Braunschweig, D–38106 Braunschweig, Germany.

Yvan Notay [2]

Service de Métrologie Nucléaire, Université Libre de Bruxelles (C.P. 165/84), 50 av. F.D. Roosevelt, B-1050 Brussels, Belgium.

Abstract

A new software code for computing selected eigenvalues and associated eigenvectors of a real symmetric matrix is described. The eigenvalues are either the smallest or those closest to some specified target, which may be in the interior of the spectrum. The underlying algorithm combines the Jacobi–Davidson method with efficient multilevel incomplete LU (ILU) preconditioning. Key features are modest memory requirements and robust convergence to accurate solutions. Parameters needed for incomplete LU preconditioning are automatically computed and may be updated at run time depending on the convergence pattern. The software is easy to use by non-experts and its top level routines are written in FORTRAN 77. Its potentialities are demonstrated on a few applications taken from computational physics.

Program summary

Title of program: JADAMILU
Catalogue identifier: ...
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/...
Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland
Licensing provisions: free for non-commercial applications
Processors & compilers: Intel or AMD with g77 & pgf; Intel EM64T or Itanium with ifort; AMD Opteron with g77, pgf & ifort; Power (IBM) with xlf90
Operating systems under which the program has been tested: Linux, AIX
Programming language used: Fortran 77
Memory required to execute with typical data: problem dependent
No. of bytes in a word: real:8; integer: 4 or 8, according to user's choice
No. of processors used: 1
No. of bytes in distributed program, including test data, etc.: 7493144
Distribution format: tar.gz
Nature of physical problem: Any physical problem requiring the computation of a few eigenvalues of a symmetric matrix.
Method of solution: Jacobi–Davidson combined with multilevel ILU preconditioning
Additional comments: We supply binaries rather than source code because JADAMILU uses the following external packages.
– MC64. This software is copyrighted software and not freely available. COPYRIGHT (c) 1999 Council for the Central Laboratory of the Research Councils
– AMD. Copyright (c) 2004-2006 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All Rights Reserved. Source code is distributed by the authors under the GNU LGPL license.
– BLAS. The reference BLAS is a freely-available software package. It is available from netlib via anonymous ftp and the World Wide Web.
– LAPACK. The complete LAPACK package or individual routines from LAPACK are freely available on netlib and can be obtained via the World Wide Web or anonymous ftp.

For maximal benefit to the community, we added the sources we are proprietary of to the tar.gz file submitted for inclusion in the CPC library. However, as explained in the README file, user willing to compile the code instead of using binaries should first obtain the sources for the external packages mentioned above (email and/or web addresses are provided).
Typical run time: problem dependent; the test examples provided with the code only take a few seconds to run; timing results for large scale problems are given in Section 5.

Key words: Eigenvalue computation, Sparse matrices, Jacobi–Davidson, Preconditioning, Numerical software, Discrete energy states
PACS: 02.60.Cb, 02.60.Dc

1. Introduction

Recently much progress has been made in developing novel algorithms for linear algebra problems involving large sparse matrices. With respect to eigenvalue problems

$$A\mathbf{x} = \lambda\mathbf{x},$$

the Jacobi–Davidson (JD) method [1,2] combined with efficient preconditioning represents a significant advance. It has been shown to be successful in several practical applications [3–13], confirming the potentialities revealed by its theoretical analysis [14,15,1,2].

Unfortunately, to date there is no publicly available general purpose software allowing non-expert users to benefit from these advances. Some JD codes are available in Matlab (e.g., [16,17]) which can be combined with Matlab built-in preconditioners. However, this is not always efficient (see Section 5) and, in addition, not so convenient for use by physicists and chemists whose applications are predominantly written in FORTRAN or C. On the other hand, some codes like PRIMME [18] or JDBSYM [19] implement JD in one of these languages, but little attention has been paid to providing an efficient preconditioner that is robust in practical applications. Thus, application scientists often rely on methods that may be much slower, such as the Lanczos method [20] or the Davidson method [21]. The latter is popular with chemists and physicists because an easy-to-use code [22] is available which is efficient for some applications in computational chemistry, in which the matrix is close to diagonal form (in some sense). However the Davidson method is not well established outside this specific context, and can be actually slow, see Section 5 below (see also [23] for a more theoretically oriented comparison).

Creating a general purpose JD code is a difficult task.

- The main JD algorithm has a complicated structure, see Section 2 below. It combines an outer iteration of the Davidson type with inner iterations to solve auxiliary linear systems. These inner systems are solved iteratively using Krylov subspace methods [24,25] and optimal performance requires a dedicated stopping test that takes into account the potential progress in the outer loop [14]. For increased robustness, the outer loop uses harmonic Ritz extraction [26] when searching for eigenvalues in the interior of the spectrum. Additionally, a deflation is needed when computing several eigenvalues [27].
- Preconditioning plays a key role in the speed of execution. Without preconditioning, JD can be as slow as the Lanczos method. For a given matrix C, a good preconditioner is a matrix P that is cheap to construct and invert, while still being a good approximation of the original matrix. This means $P^{-1}C$ is “close” to the identity matrix, whereas “cheap to invert” means that solving a system $P\mathbf{x} = \mathbf{y}$ should not cost more than a few multiplications by C. Much progress has recently been made in the development of robust algebraic preconditioning techniques suitable for general sparse matrices, see [28] for a comprehensive survey. As the methods become more robust, they also become more involved and their efficiency is more dependent on the quality of the implementation. They are then even harder to program than the JD main algorithm. In addition, specific difficulties arise because the matrix C to be approximated within JD depends on a shift parameter which may change as the convergence proceeds, see Section 3 for details.

The software developed here provides application scientists with JADAMILU, a package which is not only efficient but is also easy to use, with top level routines written in FORTRAN 77 and user defined parameters kept to a minimum. Attention is also paid to post processing, and each computed eigenvalue has an associated error bound, see Section 4 for details.

Note that, instead of JD, we could have selected the preconditioned Lanczos method [29], a Generalized Davidson method [30,31], and in particular the “GD+1” variant from [32,33], or a method based on nonlinear conjugate gradient [34], in particular LOBPCG [35] which satisfies an optimality property. All of these methods share with JD the characteristic that they are not easy to implement (especially taking into account stability issues [36]) and require preconditioning for efficiency. On average, the best alternatives perform similarly to JD when computing the smallest eigenvalue [37,32]. We selected JD because the inner-outer scheme makes it potentially more robust, in particular for computing interior eigenvalues, a case for which other approaches are not so well suited.

URLs: `http://www-public.tu-bs.de/~bolle` (Matthias Bollhöfer), `http://homepages.ulb.ac.be/~ynotay` (Yvan Notay).
[1] Part of this research was undertaken when this author was visiting the Université Libre de Bruxelles with the financial support of the Belgian FNRS
[2] Supported by the Belgian FNRS (“Maître de recherches”).

The remainder of this paper is organized as follows. In Section 2 we present a short description of the JD method. Some issues related to preconditioning are addressed in Section 3. The control of the convergence and error estimates are discussed in Section 4. Numerical results obtained on a few applications are reported in Section 5. More details on how to use the code are given in Section 6.

2. The Jacobi–Davidson method

Here we present a short review of the JD method, stressing those aspects that are relevant for non-experts. For general references about eigenvalue methods, including extensive discussions on Jacobi–Davidson, see, e.g., [38,39,26,40].

We describe the method for computing a single eigenvalue and the corresponding eigenvector. When more eigenvectors are sought, the code uses a simple deflation process: the algorithm is restarted but restricted to the subspace orthogonal to converged eigenvectors. Experiments show that this can be done stably: that is, the later eigenvectors can be computed with similar accuracy. This approach implies that memory requirements are kept to a minimum: the work space needed to compute, say, 200 eigenvectors is the same as that needed for a single eigenvector, augmented with the space needed to store 199 converged eigenvectors.

The JD method is a subspace method. That is, it builds a subspace of increasing dimension (the *search space*), in which an approximation to the desired eigenvector is sought. This desired eigenvector is, according to user choice, either the eigenvector corresponding to the smallest eigenvalue [3] or the eigenvector corresponding to the eigenvalue closest to some specified target σ. The *extraction* of an approximation to this eigenvector is performed using the standard Rayleigh–Ritz method when searching for the smallest eigenvalue. Since this procedure may give poor approximate eigenvectors for interior eigenvalues, we use Harmonic Ritz extraction when searching for the eigenvalue closest to σ. We refer to, e.g., [26] for details and algorithms, and mention only a few important issues. In both cases, one solves a small-dimensional eigenvalue problem to obtain the coefficients of the linear combination of the basis vectors of the search space that yield the approximate eigenvector of interest. Let $\mathbf{u}$ be this approximate eigenvector. The corresponding eigenvalue approximation is given by the Rayleigh quotient

$$\theta \;=\; \frac{(\mathbf{u}\,,\,A\,\mathbf{u})}{(\mathbf{u}\,,\,\mathbf{u})}\ . \tag{1}$$

The small eigenvalue problem has the same size as that of the search space and solving it yields as many solutions. From among these, the method selects the one for which the corresponding eigenvalue approximation satisfies the acceptability criterion, i.e., is either the smallest or the one closest to σ, according to requirements. This allows concentration on the convergence of the *desired* eigenvalue. Note that one can never be sure that θ effectively converges towards the desired eigenvalue. So-called *misconvergence* takes place when the search space is nearby orthogonal to the corresponding eigenvector, favoring convergence towards, say, the second smallest or second closest-to-σ eigenvalue. In practice, this happens very infrequently when computing a single eigenvalue, but slightly more frequently when computing several eigenvalues, especially when some of them are multiple or tightly clustered. Then, it may happen that one or two eigenvalues are missed in the interval between the smallest and the largest computed eigenvalues. In general, missed eigenvalues are close to the boundary of this interval, and an obvious safeguard consists in asking for a few more eigenvalues than are actually needed.

As for any subspace method, the search space has to be reduced periodically in order to avoid excessive numerical cost and storage requirements. In our code, the maximal dimension of the search space is a user specified parameter (we suggest using 20 if sufficient memory is available [4]). When the current search space reaches this dimension, its size is cut by a factor of about two, in such a way that the new search space, of dimension, say, m, contains the m most promising eigenvector approximations (according to the appropriate acceptability criterion).

These features are, of course, shared by most subspace methods, for instance, the Davidson method. The JD method is original in the way it expands the current subspace at each outer iteration step. This is done by computing an approximate solution $\mathbf{t}$ to the so-called *correction equation*

$$(I-\mathbf{u}\,\mathbf{u}^*)(A-\eta I)(I-\mathbf{u}\,\mathbf{u}^*)\mathbf{t} = -\mathbf{r} \qquad \text{where } \mathbf{t}\perp\mathbf{u}\,, \tag{2}$$

where η is an approximation to the sought eigenvalue, and where

$$\mathbf{r} \;=\; A\,\mathbf{u}-\theta\,\mathbf{u}$$

is the residual corresponding to the pair $(\theta\,,\,\mathbf{u})$. If this equation is solved exactly, then [1]

$$\mathbf{t} \;=\; -\,\mathbf{u}+\alpha(A-\eta I)^{-1}\,\mathbf{u}\ ,$$

where $\alpha=(\mathbf{u}\,,\,(A-\eta I)^{-1}\mathbf{u})^{-1}$ is such that $\mathbf{t}\perp\mathbf{u}$. This solution is used to expand the search space which already contains $\mathbf{u}$. The expansion vector is thus effectively $(A-\eta I)^{-1}\mathbf{u}$, which is the same as for inverse iteration. Thus if η is close to the sought eigenvalue rapid convergence is anticipated, and if $\eta=\theta$ very fast convergence of the Rayleigh quotient iteration ensues [39]. The latter choice is the standard one in JD, except when far from convergence, where it is better to use a value of η equal to some fixed approximation of the target eigenvalue, if available. This means

[3] The eigenvalues are real since the matrix is assumed to be symmetric and real. For technical reasons, computing the largest eigenvalue(s) is not proposed; users interested in the largest eigenvalue(s) of A are invited to call the package supplying the matrix $-A$.

[4] This number does not need to be increased with the number of requested eigenvalues, since converged eigenvectors are stored separately from the search space; the workspace supplied to JADAMILU has to take into account both the search space and the storage for converged eigenvectors, see user guide for details.

using $\eta = \sigma$ when searching for the eigenvalue closest to σ ; when searching for the smallest eigenvalue, the code relies on a user supplied eigenvalue estimate. Note, however, that the user may just indicate that no estimate is available, in which case $\eta = \theta$ is used throughout, entailing in general a slightly slower convergence rate.

The key idea in the JD method is to solve the correction equation (2) only approximately with a preconditioned Krylov subspace method. Following [14], we use the conjugate gradient (CG) method [41] when searching for the smallest eigenvalue, whereas, following [32], we use the symmetric version of QMR (sQMR) [42] when searching for the eigenvalue closest to σ . Thanks to the fact that the iterative solver is restricted to the subspace of vectors orthogonal to $\mathbf{u}$, the system matrix in (2) does not become nearly singular as θ converges to an eigenvalue. Moreover, it is positive definite when searching for the smallest eigenvalue, and the convergence analysis developed in [14,15] shows that solving the eigenvalue problem is then only slightly more costly than solving a linear system with the matrix A shifted so that it is positive definite. Note that this requires a proper stopping criterion for the inner solution of the correction equation (2); we use the one proposed in [43], which combines the advantages of those proposed in [14,32].

From a practical point of view, the projected matrix $(I-\mathbf{u}\,\mathbf{u}^*)(A-\eta I)(I-\mathbf{u}\,\mathbf{u}^*)$ is never formed explicitly. CG or sQMR require only a multiplication by this matrix at each iteration. This product is performed by sequentially applying each of three factors. If the user passes the matrix to the code, this is done internally. Alternatively, the user may provide their own routine to implement the multiplication by A , which is then called through a reverse communication protocol (see user guide for details).

3. Preconditioning

Preconditioning within JD is applied to accelerate the inner iterative solution of the correction equation (2). Since this is an intermediate step embedded in an outer subspace iteration, preconditioning has a minor impact on the global behavior of the algorithm, including its accuracy and stability. However, solving these inner systems represents the most expensive part of the algorithm, and, as for any iterative linear system solution, good preconditioning is mandatory for fast convergence.

Most work on JD consider a projected preconditioner to approximate the projected matrix $(I-\mathbf{u}\,\mathbf{u}^*)(A-\eta I)(I-\mathbf{u}\,\mathbf{u}^*)$ [44]. Here, following [45], we use a somewhat simpler and cheaper approach, in which projections are omitted from the construction of the preconditioner, which is then a simple and easy-to-invert approximation, P , to $(A-\eta I)$. To this end, we provide a robust preconditioner based on a multilevel incomplete LU (ILU) factorization of the matrix, following the algorithm described in [46]. This incomplete LU method is based on threshold techniques, that is, the matrix is factorized but entries whose absolute values fall below some specified threshold are "dropped", i.e., set to zero. This strategy is combined with an a priori reordering technique (cf. [47]) in order to save on fill-in. There are two major ways in which this novel ILU approach differs from classical ILU methods [24].

– The use of so-called "matchings".
– The special pivoting strategy which keeps the inverse triangular factors bounded.

"Matchings" can be thought of as a preprocessing step prior to factorization. Here the initial matrix is rescaled and the rows are reordered so that, in the resulting system, the diagonal entries are 1 in modulus while all off-diagonal entries are less than or equal to 1 in modulus. The impact of using "matchings" for the solution of linear systems was first observed in [48] and it turned out to be helpful for incomplete LU factorization also [49], especially when the matrix to be approximated is highly indefinite, as occurs for $A-\eta I$ when η is in the interior of the spectrum. A symmetrized version of "matchings" was presented first in [50]. This method yields a permutation matrix Π and a diagonal matrix Δ such that

$$\Pi^T \Delta A \Delta \Pi = B + E,$$

where B is a block diagonal matrix with diagonal blocks of size 1×1 and 2×2 and E denotes the remaining entries of the matrix. The entries of B and E are less than or equal to 1 in modulus. Experiments in [50,51] have indicated that, except for unusual cases, the entries in B are typically largest in modulus and can be used as static block diagonal pivots in symmetric indefinite LU decomposition methods in which a factorization of type LDL^T is computed such that D is block diagonal and L is unit lower triangular. Since then "symmetric matchings" have turned out to be an efficient preprocessing tool when applied to highly symmetric indefinite systems [50,52–54].

The multilevel incomplete LU algorithm implemented in JADAMILU uses these block diagonal pivots obtained from the symmetric matching step whenever they are "satisfactory"; that is, whenever the norm of their inverse is below a prescribed bound. Whenever they are "unsatisfactory", they are moved to the lower right part of the matrix by a further symmetric permutation. This strategy ensures that the inverse triangular factors are kept bounded. (If a diagonal pivot or a block diagonal pivot has a large or infinite inverse, then so would be the inverse triangular factor L^{-1}.) Simultaneously, this strategy creates the multilevel structure: the next level consists of the rejected (block) pivots. This multilevel structure itself is then exploited following the ideas in [55]. The robustness of this approach is illustrated in [46].

Within JD, an additional difficulty arises from the fact that the shift η varies from step to step as indicated in Section 2. Fortunately, a good preconditioner remains efficient if the matrix is shifted slightly. Hence, one may select a representative shift, compute the preconditioner for that shift, and use it during the whole computation [27,44]. When searching for the eigenvalue(s) closest to σ , we choose σ as the representative shift.

The situation is more complicated when searching for the smallest eigenvalue(s).

– Some a priori information on the location of the smallest eigenvalue is needed to define a shift.
– Knowing the lower end of the spectrum does not guarantee that one will define the best shift. Indeed, when searching for the smallest eigenvalue, better performances are obtained with positive definite preconditioners, even when η is in the interior of the spectrum (for instance, when η is equal to the Rayleigh quotient (1)) [14]. Unfortunately, incomplete LU preconditioning of a positive definite matrix is not always positive definite, which means in practice that it is often advisable to use a shift slightly smaller than the smallest eigenvalue [56]. How much smaller it should be is, however, problem dependent, although one often obtains good results by setting the gap between the shift and the smallest eigenvalue approximately equal to the gap between the smallest eigenvalue and the second smallest eigenvalue.

These difficulties are addressed as follows.

– The user is given the opportunity to pass an initial shift to the code. This may come either from a priori information about the spectrum (applying then the rule of thumb mentioned above), or from a previous execution of the code on a similar application (the code returns as output parameter a shift that is likely to be good).
– As explained above, such a shift may nevertheless be too large for the computed preconditioner to be positive definite. In this case, the following procedure is used. First, recall that the preconditioner has the form $P = LD\,L^T$, where L is lower unit triangular and D is a block diagonal matrix with small diagonal blocks of size 1×1 and 2×2 . These small diagonal blocks are called "pivots", and the preconditioner is positive definite if and only if all pivots are positive definite. The number of non-positive definite pivots is checked. If this value falls below a specified threshold (1% of the total number), all of these pivots are converted to a positive definite block, taking the absolute values in the case of 1×1 pivot, and replacing the eigenvalues of the 2×2 pivots block by their absolute values. This ensures the positive definiteness of the preconditioner. However, converting many pivots spoils the quality of the factorization. Thus, if the number of negative pivots is above the threshold, the factorization is recomputed with a smaller shift. This is done carefully to avoid spoiling the factorization with a too small shift. Several factorizations may then be needed, but this was not observed in the experiments. Note that a user can decide to recompute the factorization until all pivots are positive (this is left as an option in the code). However, this is often not cost effective. The resulting preconditioner is in general slightly better, but this may be not sufficient to compensate for the cost of the extra factorization(s).
– If no shift is specified by the user, a tentative shift is computed with a heuristic rule inspired by Gerschgorin's theorem [26, Theorem 3.2, p. 39]. This may be relatively inaccurate, but may be improved as follows. If the shift is too large, negative pivots occur, and the procedure sketched above is applied. If the shift is too small, this is detected as execution proceeds, by comparing the shift with eigenvalue approximations obtained from the extraction procedure (see Section 2). Then, a new factorization with a better shift is a possibility. Whether or not this is done depends on heuristic rules that take into account the amount of change in the shift – which should be significant – and the ratio between the factorization time and the expected remaining solution time (estimated from the previous convergence record) [5] .

With the exception of this optional shift, the factorization is parameter free, except that the user is asked to indicate which "drop tolerance" should be be used (that is, the threshold below which entries are "dropped" as indicated above). This tolerance is interpreted in a relative sense. Hence it need not be adapted according to the scaling of the matrix. It is proposed that 10^{-3} be used as a default value. Note that some adaptivity is also foreseen with respect to this parameter. First of all, recall that a smaller drop tolerance implies more fill-in, that is, more nonzero entries in L ; as a result the factors occupy more memory and are more costly to compute, i.e., the factorization time increases. But the preconditioner is likely to be more accurate, resulting in a reduction in solution time. Thus if, when computing L , it transpires that it will not fit in available memory (see user guide for details), the drop tolerance is increased and the computation restarted. With fewer nonzero entries, the new factorization is indeed more likely to fit in memory. If not, the process is repeated until a factorization is successfully computed. On the other hand, as execution proceeds, the solution time is monitored and compared to the factorization time. If the convergence appears to be slow, and if enough memory appears to be available, a new factorization is computed with smaller drop tolerance, producing, it is hoped, a better preconditioner. After a complete run, the code also computes and returns as output parameter a new value for the drop tolerance, that can be used in subsequent runs. This value is different from the input value either if the adaptive procedure sketched above has been applied or if the factorization time is small or large compared to the solution time for 1 eigenvalue. Indeed it has been observed that, in general, optimal performances are obtained when there is a good balance between both.

Finally, the user does not need to supply the *exact* matrix when computing the preconditioner. Since the latter will be only an approximation to the shifted matrix anyway, the matrix supplied to the code may itself only be an approximation if this more convenient (then, of course, the user should provide their own routine to implement multi-

[5] The *factorization time* is the CPU time needed to compute the matrix L and the *solution time* is the CPU time actually spent to compute the eigenvectors; that is, the total time minus the factorization time

plication by A). This is useful in some applications, where matrix entries are not explicitly available; one might then compute, for instance, only the dominant part of A, filtering out small entries. One may also supply only the diagonal of the matrix, in which case a diagonal preconditioner is returned instead of the multilevel incomplete LU.

4. Control of convergence and error analysis

The control of the convergence is based on the residual norm

$$\|\mathbf{r}\| = \|A\,\mathbf{u} - \theta\,\mathbf{u}\| , \tag{3}$$

where the approximate eigenvector $\mathbf{u}$ has been normalized: $\|\mathbf{u}\| = 1$. The computation is stopped when $\|\mathbf{r}\|$ becomes smaller than the user specified tolerance.

Since the expansion of the search space is based on $\mathbf{r}$ (see (2)), the residual norm can be effectively decreased as long as the computation of $\mathbf{r}$ is not marred by rounding errors. This occurs when $\|\mathbf{r}\|$ is of the order $\varepsilon_{\mathrm{mach}}\|A\|$, where $\varepsilon_{\mathrm{mach}}$ is the unit roundoff of the machine, and where $\|A\|$ (the norm of A) is here its largest eigenvalue in modulus. Continuing to that point gives eigenvectors whose accuracy is close to the best attainable [26, pp. 61–62]. If the user specifies a tolerance below this limit, the code detects it and adapts the tolerance accordingly. Using very small tolerances should however be considered with care. This adaptation requires several iterations, incurring a computing time penalty. It may also fail if the specified tolerance is significantly smaller than the attainable accuracy, in which case the algorithm stagnates until the iteration limit is reached.

Except for situations where this adaptation is necessary, the convergence criterion will be satisfied if sufficient iterations are permitted.

The code also prints an error bound for the converged eigenvalues. In most cases it is based on the inequality [57, Corollary 3.4]

$$|\theta - \lambda_i| \leq \frac{\|\mathbf{r}\|^2}{\delta_i} , \tag{4}$$

where λ_i is the eigenvalue of A that is closest to θ, and where $\delta_i = \min_{j\neq i} |\theta - \lambda_j|$ is the distance to the remainder of the spectrum.

It is important to note that this bound is valid only when θ is the *exact* Rayleigh quotient (1), and that the code computes it in finite precision. Because the bound is quadratic in $\|\mathbf{r}\|$, it often happens that *the error bound returned is untrustworthy because it is swapped by the rounding errors incurred in the Rayleigh quotient computation.* In such cases, a trustworthy error estimate should be based on a rounding error analysis. Roughly speaking, rounding errors on the Rayleigh quotient are expected of the order $\varepsilon_{\mathrm{mach}}\|A\|$. However, this might be pessimistic. In fact, the Rayleigh quotient is exact for a nearby matrix in which each nonzero entry differs from that of A by a perturbation of order $\varepsilon_{\mathrm{mach}}$ (in relative value) [58]. An accurate analysis depends on the structure of the matrix and is the responsibility of the user. Note that the computed error is useful in any case if the Rayleigh quotient is recomputed in extended precision (using the eigenvector produced by the code).

Now, the bound (4) depends on δ_i, which requires, in principle, knowledge of all eigenvalues but the computed one. When computing a single eigenvalue, we may obtain, as a byproduct of the Rayleigh–Ritz or Harmonic Ritz extraction, an estimate of the gap between the computed eigenvalue and the remainder of the spectrum. The latter is used to derive the error estimate, which should therefore be considered as indicative, since this gap has not been computed safely. When computing several eigenvalues, we use the same approach for the least of these, that is, for the one largest or farthest from σ, according the problem being addressed. For this eigenvalue, the same restriction applies. For the others however, δ_i is known sufficiently accurately, *assuming no misconvergence occurred*, as defined in Section 2. Then, reported error bounds are trustworthy if not contaminated by rounding errors as discussed above.

On the other hand, the bound (4) is not so useful when λ_i is multiple or tightly clustered with other eigenvalue(s) (the code cannot distinguish between those situations). In these cases, better error bounds may be obtained from the results in [59] by considering clusters of eigenvalues. Essentially, the same sort of quadratic bound holds, but the gap δ_i is now the distance of θ to the eigenvalues that are not in the cluster. This approach has been implemented, with a few limitations that need to be addressed. Firstly, an additional Rayleigh–Ritz computation needs to be performed with a basis that has the size of the cluster. Thus, the code is only able to detect clusters whose size does not exceed the maximal dimension of the search space, as specified by the user (see Section 2). Next, the least computed eigenvalue, as defined above, is also excluded from possible clusters. Otherwise, for any eigenvalue in the cluster, the reported error estimate would depend on the gap to the uncomputed part of the spectrum, which is not computed accurately.

Eventually, when δ_i is larger than $\|\mathbf{r}\|$ (because, for instance, clusters cannot be resolved), we use the alternative bound [57, Corollary 3.4]

$$|\theta - \lambda_i| \leq \|\mathbf{r}\| .$$

5. Numerical experiments

Computing environment. Experiments where performed either (Configuration 1) on an Intel XEON 32 bit processors at 3.05 GHz, and 2 GB of SDRAM memory, or (Configuration 2) an Intel Xeon EM64T at 3.66 GHz and 8 GB of SDRAM memory. The operating system is Linux in both cases, with compiler `g77` for the first configuration and `gfortran` for the second. The code was used throughout with default and suggested options & parameters (see user guide for details), except for some runs performed to illustrate the influence of some of them. In particular, the maximal dimension of the search space mentioned in Section 2 was always set to 20, and the initial drop tolerance for incomplete LU preconditioning discussed in Section 3 was always set to 10^{-3} , except in the third example where the influence of this parameter is discussed.

3D Discrete Laplacian. The first example is the three-dimensional Laplacian on the unit cube discretized by seven point finite differences. This problem is relatively hard to precondition by general purpose methods although many specialized techniques have been developed. From the point of view of eigenvalue computation, it is interesting because the discrete eigenvalues are known, allowing the accuracy of the computed ones to be determined. It is also challenging because most eigenvalues are multiple, some of them with multiplicity 6.

Here we are interested in the smallest eigenvalues. Because it is well known that the matrix is positive definite, zero is supplied as the initial eigenvalue estimate (optional parameter discussed page 4) and zero is supplied as the initial shift (optional parameter discussed page 5).

JADAMILU was first executed with the tolerance on the residual norm (3) set to 10^{-10} , requesting successively 1, 10 and 200 eigenvalues. The results are reported in Table 1. All eigenvalues were found with correct multiplicity and with maximal error less than $\varepsilon_{\mathrm{mach}}\|A\|$.

	$40 \times 40 \times 40$ grid			$80 \times 80 \times 80$ grid		
n	5939			493039		
$nzau$	$3.8\,n$			$3.9\,n$		
$\varepsilon_{\mathrm{mach}}\|A\|$	2.8e-12			1.1e-11		
#Eigs	1	10	200	1	10	200
Mem. ILU	3.6 *nzau*	3.6 *nzau*	3.6 *nzau*	3.8 *nzau*	10.7 *nzau*	10.7 *nzau*
Mem. eig.	$62.0\,n$	$71.0\,n$	$261.0\,n$	$62.0\,n$	$71.0\,n$	$261.0\,n$
Max. Err.	5.9e-13	8.1e-13	1.0e-12	1.6e-12	2.1e-12	4.0e-12
#Matvec	45	471	16568	75	438	16194
Fact. time	2.2	2.2	2.1	2.2e1	1.3e2	1.3e2
Total time	4.8	3.8e1	4.1e3	6.2e1	5.9e2	4.0e4

Table 1
Results obtained with JADAMILU for the Laplace problem (Configuration 1); *nzau* denotes the number of nonzero entries in the upper triangular part of the matrix; "Mem. ILU" refers to the memory needed to store the preconditioner (automatically allocated within JADAMILU); "Mem. eig." refers to the additional memory needed to compute and store the eigenvectors (to be addressed as workspace by the calling program); "Err." refers to the absolute value of the difference between the exact and the computed eigenvalue, checked for each of the computed eigenvalues (with multiple eigenvalues checked as many times as they occur); "#Matvec" stands for the number of matrix vector multiplications performed; timings refer to CPU time and are reported in seconds.

As expected, the preconditioner is not scalable; that is, the number of iterations increases with the problem size. Better results would have been obtained with specific preconditioning, for instance based on multigrid (e.g., [60]), but this would require a problem dependent approach whereas JADAMILU is a general purpose code (note that it is also possible to use JADAMILU with a user defined preconditioner). Nevertheless, a degree of scalability is obtained thanks to the adaptivity of the drop tolerance: when asking for 10 or 200 eigenvalues on the largest grid, the preconditioner is recomputed during execution with a smaller drop tolerance, entailing that the number of matrix vector multiplications remains about the same as for the smaller grid. There is some price to pay, however: more memory is needed (see "Mem. ILU"), which implies that iterations are more costly. Recomputing the preconditioner also incurs a time penalty, which explains why JADAMILU omits this operation when a single eigenvalue is requested. Note also that JADAMILU would have to make do with the original preconditioner if not enough memory was available to compute a better one.

We also executed JADAMILU with varying tolerances, in order to check the relevance of the printed error bound. The results for the first eigenvalue are reported in Table 2. Observe that, following the discussion in Section 4, the error bound is reliable until it becomes smaller than the rounding error incurred when computing the Rayleigh quotient.

JADAMILU was also compared with other eigensolvers publicly available. Firstly, many eigensolvers are available in Matlab, and this is convenient because preconditioners may be set up in Matlab using built-in functions. As an example, JADAMILU was compared with Matlab code JDCG [16], using the preconditioner obtained with the Matlab *CHOLINC* function, which is also a symmetric ILU preconditioner based on threshold techniques. JDCG also implements the Jacobi-Davidson method and shares many features with JADAMILU. With respect to preconditioning, setting the threshold parameter to 10^{-3} results in a preconditioner for which convergence is roughly as fast as for the JADAMILU preconditioner with default drop tolerance 10^{-3} . However, as shown in Table 3, *CHOLINC* requires significantly more memory to achieve this. Consequently, setting up the preconditioner also takes significantly more time, and the Matlab code is much slower for computing a single eigenvalue. Its relative performance improves a little when computing several eigenvalues, but the program fails due to insufficient memory when 200 eigenvalues are requested on the largest grid.

Next ARPACK [61], a package based on implicitly restarted Arnoldi-like methods [62], was considered. Run without adjustment, the package could not find the smallest eigenvalue in an acceptable amount of time. To achieve convergence, it is needed to combine it with inverse iteration; that is, ARPACK is called to compute the largest eigenvalues of A^{-1} . This requires the implementation of an interface with a direct solver providing an LU factorization of A . In Table 3 we report the results obtained with the PARDISO solver [63,51]. The approach is competitive for small problems. However, for large problems, it suffers from the lack of scalability of direct solvers, especially with regard to their excessive memory requirements.

We then used ARPACK implementing the multiplication by A^{-1} via the iterative solution of systems $A\mathbf{x} =$

Computed Eig.	Tolerance	Res. norm	Error bound	True error
$40 \times 40 \times 40$ grid: $\lambda = 29.5935961619708$ (exact)				
29.5939600145405	1.0e-00	2.6e-01	2.4e-03	3.6e-04
29.5935996308861	1.0e-01	5.7e-02	5.9e-05	3.5e-06
29.5935961802297	1.0e-02	3.7e-03	4.6e-07	1.8e-08
29.5935961626138	1.0e-03	9.7e-04	3.2e-08	6.4e-10
29.5935961619931	1.0e-04	8.9e-05	2.7e-10	2.2e-11
29.5935961619717	1.0e-05	9.5e-06	3.1e-12	9.5e-13
29.5935961619714	1.0e-06	5.9e-07	1.2e-14	5.9e-13
29.5935961619714	1.0e-07	4.1e-08	5.8e-17	5.9e-13
29.5935961619714	1.0e-08	9.3e-09	2.9e-18	5.9e-13
29.5935961619714	1.0e-09	4.9e-10	8.2e-21	5.9e-13
29.5935961619714	1.0e-10	5.4e-11	9.8e-23	5.9e-13
$80 \times 80 \times 80$ grid: $\lambda = 29.6050083562378$ (exact)				
29.6070018684169	1.0e-00	8.3e-01	2.4e-02	2.0e-03
29.6050163413501	1.0e-01	9.6e-02	3.1e-04	8.0e-06
29.6050083691372	1.0e-02	4.1e-03	5.8e-07	1.3e-08
29.6050083566645	1.0e-03	9.3e-04	2.9e-08	4.3e-10
29.6050083562483	1.0e-04	9.9e-05	3.3e-10	1.1e-11
29.6050083562393	1.0e-05	7.4e-06	1.8e-12	1.6e-12
29.6050083562393	1.0e-06	5.9e-07	1.2e-14	1.6e-12
29.6050083562393	1.0e-07	8.6e-08	2.5e-16	1.6e-12
29.6050083562393	1.0e-08	5.6e-09	1.1e-18	1.6e-12
29.6050083562393	1.0e-09	4.6e-10	8.2e-21	1.6e-12
29.6050083562393	1.0e-10	4.2e-11	5.8e-23	1.6e-12

Table 2
Results for the Laplace problem (first eigenvalue) with varying tolerance on the residual norm (3) (Configuration 1).

y using the CG method. Without preconditioning, it was not possible to compute any eigenvalue in an acceptable amount of time. We then developed an interface with the ILUPACK package [64], a software package that provides a preconditioner which is somewhat similar to the one implemented in JADAMILU (but without matchings). We stress that *this required a substantial implementation effort which most users would be happy to avoid.* The results are reported in Table 3. The approach is not competitive when few eigenvalues are requested, but it is roughly twice as fast as JADAMILU when 200 eigenvalues are requested. This weaker performance of JADAMILU is due to the projections needed to maintain the computation in the subspace orthogonal to converged eigenvectors. This ensures the robustness of the method, but is relatively costly when many eigenvalues are requested. This explanation is confirmed by checking the number of matrix vector multiplication performed, which is smaller with JADAMILU. In practice, this means that JADAMILU could remain faster even when many eigenvalues are requested in applications

	$40 \times 40 \times 40$ grid			$80 \times 80 \times 80$ grid		
#Eigs	1	10	200	1	10	200
Preconditioners						
JADAMILU						
Mem. ILU	3.6 *nzau*	3.6 *nzau*	3.6 *nzau*	3.8 *nzau*	10.7 *nzau*	10.7 *nzau*
Fact. time	2.2	2.2	2.1	2.2e1	1.3e2	1.3e2
CHOLINC (Matlab)						
Mem. ILU	7.7 *nzau*			8.1 *nzau*		
Fact. Time	3.1e1			1.4e3		
PARDISO (direct LU)						
Mem. LU	58. *nzau*			not enough memory		
Fact. time	2.9e1					
ILUPACK						
Mem. ILU	6.3 *nzau*			7.0 *nzau*		
Fact. time	6.2			6.9e1		
Solvers						
JADAMILU						
#Matvec	45	471	16568	75	438	16194
Total time	4.8	3.8e1	4.1e3	6.2e1	5.9e2	4.0e4
JDCG & CHOLINC (Matlab)						
#Matvec	40	449	17616	74	775	–
Total time	4.3e1	1.5e2	7.2e3	1.6e3	3.0e3	–
ARPACK & PARDISO						
#Matvec	61	121	620	–	–	–
Total time	4.9e1	7.3e1	6.8e2	–	–	–
ARPACK & CG(ILUPACK)						
#Matvec	1521	3012	14900	2623	2940	27200
Total time	9.4e1	1.8e2	1.3e3	1.4e3	1.6e3	1.8e4
BLOPEX & ILUPACK						
#Matvec	26	450	7200	50	710	–
Total time	8.3	5.5e1	8.9e3	1.0e2	7.4e2	–
PRIMME & ILUPACK						
#Matvec	60	418	35783	92	556	29548
Total time	9.4	3.1e1	2.5e3	1.2e2	3.8e2	2.0e4

Table 3
Results for the Laplace problem (Configuration 1); "Mem. (I)LU" refers to the memory needed to store the preconditioner; *nzau* denotes the number of nonzero entries in the upper triangular part of the matrix; "#Matvec" stands for the number of matrix vector multiplications performed; timings refer to CPU time and are reported in seconds.

where the matrix has many more nonzeros per row.

Finally, PRIMME [18] and BLOPEX [65] were considered. Like JADAMILU, PRIMME is based on a Jacobi-Davidson-like method whereas BLOPEX is an implementation of the LOBPCG method [35]. These packages do not provide preconditioning, which is left to the user. Without preconditioning, they were unable to compute the smallest eigenvalue in an acceptable amount of time. We then modified these eigensolvers, making an interface with ILUPACK. Again, we stress that *this required a substantial implementation effort.* The results are reported in Table 3. BLOPEX is always slower than JADAMILU and breaks down on the largest grid when requesting 200 eigenvalues. On the other hand, PRIMME appears to be competitive. Roughly speaking, it is twice as slow when requesting 1 eigenvalue, as fast when requesting 10, and it is twice as fast when requesting 200. As for the combination ARPACK & CG, the superior execution time comes with slower convergence in terms of matrix vector products.

To summarize, JADAMILU performs better than state of the art solvers except when many eigenvalues are requested. In the latter case JADAMILU remains an option if the matrix has relatively many nonzero entries per row, and/or if ease of use is a prime consideration (save user effort), whereas a combination of ARPACK or PRIMME with ILUPACK is advisable if saving on CPU time is the most important consideration.

3D Lagrange mesh calculation. The next example arises in the context of a 3D Lagrange mesh calculation of three-body atoms and molecules [66,67]. This method uses a Lagrange polynomial basis to approximate wave functions of three body systems in perimetric coordinates. After discretization, a matrix is obtained whose smallest eigenvalues give the discrete energies of the quantum states of the system. Using a Lagrange basis allows highly accurate energies to be obtained with moderately large mesh sizes. Despite the use of a Gauss quadrature formula to approximate the potential, the method is in fact often as accurate as a complete variational calculation with Lagrange bases [66]. While this Gauss approximation simplifies the computation of the discretization matrix, solving the eigenvalue problems remains expensive because the matrix is sparse, but not overly so.

To illustrate the potentialities of JADAMILU, a configuration that allows the computation of the vibrational spectrum of the H_2^+ molecule is considered. Note that this spectrum is computed accurately with a complete three-body simulation. The problem is difficult from a numerical point of view, because the smallest energies are tightly clustered: $E_1 = -0.593\,,\ E_2 = -0.583\,,\ E_3 = -0.574\,,\ E_4 = -0.565\,,$ which means that the gap between consecutive eigenvalues is about 10^{-2}, very small compared to the largest diagonal elements in the matrix, which are of the order of 10^6.

In Table 4, we give the results obtained with both JADAMILU and the Davidson code from [68], which was used to produce the numerical results in, e.g., [66,67]. In all cases, the tolerance on the residual norm (3) is set to 10^{-10}. That this tolerance is effectively achieved has been checked and the converged eigenvalue is indeed the smallest one (no misconvergence).

For JADAMILU, both the default multilevel ILU preconditioner, and the diagonal preconditioner obtained when only the diagonal of the matrix is passed to the code are considered. For illustrative purposes, several options have been tested: passing (IS="y") or not passing (IS="n") a initial eigenvalue estimation (see page 4) and initial shift (see page 5) (when IS="y", the values output by the code when running the computation *on a coarser mesh* are initially passed); allowing adaptive preconditioning as discussed in Section 3 (AP="y") or not (AP="n"); replacing negative pivots with positive values (CP="y"), or decreasing the shift until all pivots are positive (CP="n"), as discussed in Section 3. Note that the standard usage (default options) corresponds to IS="n", AP="y" and CP="y" (the first line in the table for each preconditioner type).

Observe that the gains with respect to the Davidson code are impressive even with a diagonal preconditioner; that is, a preconditioner similar to the one used in the Davidson method. This confirms that the inner–outer iteration scheme approach of the JD method is more robust than the pure subspace approach of the Davidson method [23]. With the diagonal preconditioner, it is always better to pass a priori information on the eigenvalues (IS="y"). However, the adaptive procedure works satisfactorily, and the code is nearly as fast without a priori information (IS="y") when adaptivity is enabled (AP="y").

With respect to ILU preconditioning, note that the matrix is relatively dense; for the finest mesh, the matrix passed to the code already occupies about 0.4 GB of memory. Fortunately, using the default drop tolerance (10^{-3}) produces, in all cases, a preconditioner *sparser* than the original matrix. Indeed, the latter contains many nonzeros per row that vary greatly in magnitude, and the smallest are automatically filtered out by the ILU threshold-based process (other memory requirements are not addressed because they correspond only to a small multiple of n).

Observe that the shift adaptivity works well: if the preconditioner is recomputed (AP="y"), convergence is nearly as fast without the provision of any a priori information (IS="n"). However, some price has to be paid because the adaptive procedure results in higher factorization times (the reported factorization time is the sum of the CPU times needed to compute all ILU preconditioners). For the largest mesh, the code is nevertheless faster without a priori information. In fact, the initial shift passed to the code is not particularly good in this case, because the resulting factorization has a few negative pivots that are then made positive using the default option (CP="y"). The adaptive procedure does not suffer from this deficiency because it produces a somewhat smaller shift. A better shift is also generated if, starting with initial shift passed to the code (IS="y"), one enforces recomputation of the factor-

	IS	AP	CP	Mem. ILU	#Matvec	Fact. time	Tot. time
				$20 \times 20 \times 15$ mesh $(n = 6000\,,\ nzau = 385.6\,n)$			
DAVIDSON					21250		20m04s
JAD. (DIA)	n	y	–	–	1245		54s
JAD. (DIA)	y	y	–	–	1212		53s
JAD. (DIA)	n	n	–	–	1764		1m17s
JAD. (DIA)	y	n	–	–	1358		59s
JAD. (ILU)	n	y	y	0.80 *nzau*	175	23s	37s
JAD. (ILU)	y	y	y	0.82 *nzau*	29	12s	15s
JAD. (ILU)	n	n	y	0.11 *nzau*	1067	3s	54s
JAD. (ILU)	y	n	y	0.82 *nzau*	29	12s	15s
JAD. (ILU)	n	y	n	0.82 *nzau*	175	23s	37s
JAD. (ILU)	y	y	n	0.82 *nzau*	29	12s	15s
				$40 \times 40 \times 20$ mesh $(n = 32000\,,\ nzau = 1180.5\,n)$			
DAVIDSON					66729		13h28m25s
JAD. (DIA)	n	y	–	–	2592		30m48s
JAD. (DIA)	y	y	–	–	2501		29m52s
JAD. (DIA)	n	n	–	–	4103		49m07s
JAD. (DIA)	y	n	–	–	2501		29m45s
JAD. (ILU)	n	y	y	0.62 *nzau*	479	6m08s	12m31s
JAD. (ILU)	y	y	y	0.64 *nzau*	519	4m45s	17m13s
JAD. (ILU)	n	n	y	0.01 *nzau*	1928	1m12s	22m40s
JAD. (ILU)	y	n	y	0.64 *nzau*	519	4m45s	17m13s
JAD. (ILU)	n	y	n	0.62 *nzau*	479	6m10s	12m36s
JAD. (ILU)	y	y	n	0.64 *nzau*	129	8m41s	11m35s

Table 4
Results for the Lagrange mesh calculation, computing the smallest eigenvalue (Configuration 1); *nzau* denotes the number of nonzero entries in the upper triangular part of the matrix; "Mem. ILU" refers to the memory needed to store the preconditioner; "#Matvec" stands for the number of matrix vector multiplications performed.

ization until all pivots are positive instead of converting them (CP="n"). This latter option has no effect on the smallest mesh (there are no negative pivots), but improves the preconditioner significantly on the largest mesh at the price, however, of a relatively large factorization time.

To summarize: the default options (IS="n", AP="y" and CP="y") offer robust and cost effective convergence, and tuning is not required; in particular, the adaptive shift setting (a novel feature of JADAMILU) works well. Other options are left for those who experience convergence problems: they may try to improve the situation by passing an initial shift to the code and/or by enforcing the computation of a new factorization until all pivots are positive. Note that JADAMILU reports how many negative pivots have been converted.

In Table 4 we illustrate the computation of the smallest eigenvalue. It is, however, especially interesting to compute a few more, since this allows the deduction of the vibrational frequencies. Using the Davidson code for this purpose would be prohibitively slow. The results with JADAMILU (with default options) are given in Table 5. As has already been seen in Table 4, for the "small" problem, Diagonal and ILU preconditioning result in roughly comparable performances. More precisely, ILU preconditioning significantly improves the convergence rate, but this is not really advantageous, because of the additional costs (factorization time and more costly iterations). For the largest mesh however, ILU preconditioning dramatically improves the performance and allows the computation of more eigenvalues with only a small additional cost.

	1 eigenvalue		5 eigenvalues	
	#Matvec	Tot. time	#Matvec	Tot. time
		$20 \times 20 \times 15$ mesh		
JAD. (DIA)	1245	54s	2611	1m54s
JAD. (ILU)	175	37s	1011	2m10s
		$40 \times 40 \times 20$ mesh		
JAD. (DIA)	2592	30m48s	7892	1h34m15s
JAD. (ILU)	479	12m31s	686	18m14s

Table 5
Results for the Lagrange mesh calculation, computing the smallest or the five smallest eigenvalue(s) (Configuration 1); *nzau* denotes the number of nonzero entries in the upper triangular part of the matrix; "#Matvec" stands for the number of matrix vector multiplications performed.

For this example JADAMILU has been compared with the Davidson code from [68] since this is the code used till now for the related research project [66,67]. As for the Laplace problem, the comparison could, in principle, include state of the art solvers built on ARPACK, PRIMME or BLOPEX interfaced with some preconditioning package. However, the matrix in this case is not positive definite. Hence, it is not sufficient just to construct this interface. In addition, it is requested to shift the matrix before calling the preconditioning routine, and choosing a proper shift is not easy – see the discussion in Section 3. The results in Table 5 for JADAMILU also display the sensitivity to the shift when adaptivity is turned off (AP="n"). Because it is difficult to anticipate how a non-expert user willing to use ARPACK, PRIMME or BLOPEX would define a shift, we concluded that a fair comparison was not possible.

The Anderson Model. The Anderson model of localization [69] describes the electronic transport properties in disordered quantum systems. Mathematically, the discretized problem is described by a large sparse real symmetric eigenvalue problem $A\mathbf{x} = \lambda\mathbf{x}$, where the eigenvectors close to zero correspond to quantum wave functions. A is obtained from 7-point stencil discretization in three spatial dimensions similar to the discretization of the Laplacian operator with periodic boundary conditions, except that the diagonal entries are randomly chosen from an interval $[-w/2, w/2]$. One of the most physically interesting cases is the choice $w = 16.5$ which reflects a metal–insulator transition.

To obtain significant results, large $m \times m \times m$ grids are required, resulting in huge matrices of size $n = m^3$. As already mentioned the eigenvectors and eigenvalues closest to zero are of interest. Since zero is roughly in the middle of the spectrum, this problem is very challenging from a numerical point of view: the requested eigenvalues are really in the interior of the spectrum. This challenge was already considered in [54], where a specific code was developed which shares many features with JADAMILU: it uses the JD method as implemented in JDBSYM [19] (enhanced with the stopping criterion from [14]) in combination with an ILU preconditioner similar to the one in JADAMILU. This code, however, is not publicly available and required much hand tuning to achieve convergence for $m > 160$. In particular, after trial and error, it was found useful to set the drop tolerance for ILU preconditioning to $n^{-1/2}$.

On the basis of this code, significant advances have been made in Anderson model computations: using a computer with 112 GB of SDRAM memory, problems for m up to 350 could be solved; other standard software packages could not cope for $m \geq 100$. This is further illustrated in Table 6 where JADAMILU is compared with ARPACK in combination with PARDISO to implement the multiplication by A^{-1} . Note that neither PRIMME nor BLOPEX can compute eigenvalues in the interior of the spectrum, whereas there is no Matlab built-in preconditioner that works for indefinite matrices and preserves symmetry. Hence this combination of ARPACK and PARDISO represents the best publicly available alternative to JADAMILU.

	Mem. (I)LU	Fact. time	Tot. time
$40 \times 40 \times 40$ mesh $(n = 64,000\ ,\ nzau = 4\,n)$			
JADAMILU	11. $nzau$	1.2e1	3.0e1
ARPACK & PARDISO	130. $nzau$	3.9e2	9.8e2
$70 \times 70 \times 70$ mesh $(n = 343,000\ ,\ nzau = 4\,n)$			
JADAMILU	11. $nzau$	8.6e1	2.7e2
ARPACK & PARDISO	280. $nzau$	1.8e4	3.7e4

Table 6
Results for the Anderson model: computation of the 5 eigenvalues closest to zero (Configuration 1); $nzau$ denotes the number of nonzero entries in the upper triangular part of the matrix; ; "Mem. (I)LU" refers to the memory needed to store the triangular factors of A (PARDISO) or the preconditioner (JADAMILU); timings refer to CPU time and are reported in seconds.

Compared with the code from [54] we cannot hope to improve it with JADAMILU since both codes are based on similar algorithms. However, it is interesting to enquire if a general purpose code allows comparable results to be obtained without hand tuning. To this end, both the code from [54] and JADAMILU were executed. JADAMILU was tested with default options and parameter; that is, starting with drop tolerance 10^{-3} (but allowing for its reduction) and the computation of a new preconditioner if the convergence is deemed to be too slow. For illustrative purposes, JADAMILU was also tested with the latter option disabled (AP="n"), which means that the initial preconditioner is used throughout. This option was considered in combination with two drop tolerances: 10^{-3} and $n^{-1/2}$.

The results are reported in Table 7. The largest value for m chosen was 190 because only 8 GB of SDRAM memory is available. However, this is sufficient to show the trend.

	AP	Drop Tol.	Mem. ILU	#Matvec	Fact. time	Tot. time
$130 \times 130 \times 130$ mesh $(n = 2,197,000\ ,\ nzau = 4\,n)$						
JAD.	y	1.0e-3	11. $nzau$	386	10m31s	51m09s
JAD.	n	1.0e-3	11. $nzau$	386	10m31s	51m09s
JAD.	n	6.7e-4	13. $nzau$	275	13m09s	44m42s
[54]	n	6.7e-4	13.1 $nzau$	232	14m40s	47m48s
$160 \times 160 \times 160$ mesh $(n = 4,096,000\ ,\ nzau = 4\,n)$						
JAD.	y	2.5e-4	17. $nzau$	308	1h04m23s	2h17m51s
JAD.	n	1.0e-3	11. $nzau$	760	18m53s	2h52m30s
JAD.	n	4.9e-4	14. $nzau$	418	29m58s	2h09m19s
[54]	n	4.9e-4	14. $nzau$	319	35m00s	2h00m44s
$190 \times 190 \times 190$ mesh $(n = 6,859,000\ ,\ nzau = 4\,n)$						
JAD.	y	2.5e-4	17. $nzau$	301	1h48m20s	3h55m04s
JAD.	n	1.0e-3	11. $nzau$	>5000	32m00s	>29h
JAD.	n	3.8e-4	15. $nzau$	376	59m16s	3h36m25s
[54]	n	3.8e-4	15. $nzau$	311	1h00m00s	3h23m26s

Table 7
Results for the Anderson model: computation of the 5 eigenvalues closest to zero (Configuration 2); $nzau$ denotes the number of nonzero entries in the upper triangular part of the matrix; "Drop Tol." is the drop tolerance for the least computed ILU; "Mem. ILU" refers to the memory needed to store the preconditioner; "#Matvec" stands for the number of matrix vector multiplications performed; timings refer to CPU time and are reported in seconds.

Observe first that some adaptivity of the drop tolerance is imperative as m increases. If the preconditioner obtained with 10^{-3} is used throughout, convergence becomes slow for $m = 190$: after 5000 matrix vector multiplications (the maximum allowed), only 4 eigenvectors have been computed to the required accuracy, and the computation of the fifth was still in progress. In contrast, JADAMILU performs similarly to the application-specific code from [54] when using the same drop tolerance $n^{-1/2}$. Moreover, JADAMILU is about as fast with default options. More precisely, JADAMILU converges as fast, but more execution time is needed, because the ILU preconditioning is computed twice.

Note that [54] is the result of a joint effort of numerical analysts. With JADAMILU, similar advances could have been achieved directly by researchers working on application problems like the Anderson model, without deep knowledge of the underlying numerical methods.

6. User guide and test run output

JADAMILU is distributed as a single file called `jadamilu.tar.gz`. Expanding it in the standard way creates a directory `./JADAMULU/` and several subdirectories. In `./JADAMULU/Doc/`, a user guide is provided that describes in detail how to call top level JADAMILU routines in an application program. These routines are also documented in the source code available in file `./JADAMULU/src/PJD.f`. In addition, in `./JADAMULU/`, a `README` file explains how to compile and run an application that uses the JADAMILU routines, for instance the examples provided in `./JADAMULU/samples/`. The test run output for these examples is reproduced in the user guide.

The basic usage of JADAMILU consists in calling the top level routine PJD from a FORTRAN 77 or FORTRAN 90 program, supplying the matrix and basic parameters. These tell which and how many eigenvalues are requested, what information is to be printed and the tolerance on the residual norm. In addition, there are the parameters discussed in the preceding sections, which may be set to the suggested default values.

Because of the symmetry, only the upper triangular part of the matrix has to be supplied. The required format is the compressed sparse row (CSR) format [70]. Note that all diagonal entries must be referenced, even when zero. For the strictly upper triangular part, only nonzero entries need to be referenced. The CSR format requires three arrays. `A` is a double precision array containing the diagonal entries and the nonzero offdiagonal entries in the strictly upper triangular part, stored row-wise; that is, `A` first contains the entries of the first row (in any order), next those of the second row, etc. `JA` is a integer array such that `JA`(i) contains the column index of the entry stored in `A`(i). `IA` is an integer array of length $n{+}1$ such that `IA`(i), $1 \le i \le n$, points to the beginning of row i in arrays `A` and `JA`, while `IA`$(n{+}1)-1$ points to the last entry in the last row. As an example, consider the matrix

$$A = \begin{pmatrix} 0 & 5 & & & \\ 5 & 1 & 5 & & \\ & 5 & 2 & 5 & \\ & & 5 & \ddots & \ddots \\ & & & \ddots & n-1 \end{pmatrix} \in \mathbb{R}^{n,n} .$$

With $n = 5$, the arrays to be supplied are

$$\begin{aligned} \texttt{A} &= \left[\, 0.0\ 5.0 \,\middle|\, 1.0\ 5.0 \,\middle|\, 2.0\ 5.0 \,\middle|\, 3.0\ 5.0 \,\middle|\, 4.0 \,\right] \\ \texttt{JA} &= \left[\, 1\ 2 \,\middle|\, 2\ 3 \,\middle|\, 3\ 4 \,\middle|\, 4\ 5 \,\middle|\, 5 \,\right] \\ \texttt{IA} &= \left[\, 1\ 3\ 5\ 7\ 9\ 10 \,\right] . \end{aligned}$$

The sample program below calls JADAMILU to compute the five eigenvalues closest to zero of the above matrix when $n = 1000$. Parameters are defined as follows. `N=1000` is the order of the matrix. `EIGS`, `X` and `RES` are double precision arrays in which will be returned, respectively, the eigenvalues, the corresponding eigenvectors and the associated residual norms. These arrays need not be initialized in the calling program. `LX` is the size of `X` whose minimal value is indicated in the user guide (the latter size exceeds the space needed to store eigenvectors because `X` is also used as workspace). `NEIG=5` is the number of requested eigenvalues. `SIGMA=0.0d0` together with `ISEARCH=2` indicate that the eigenvalues closest to zero are requested. `NINIT=0` indicates that no initial approximation for the eigenvectors is supplied in `X`. `MADSPACE=20` is the size of the search space mentioned in Section 2 (20 is the suggested default). `ITER=1000` is the maximal number of matrix vector product allowed. `TOL=1.0d-10` is the tolerance on the residual norm (see (3)). `SHIFT` is the optional shift parameter discussed in Section 3; it is not initialized here because it is only meaningful when requesting the smallest eigenvalue(s). `DROPTOL=1.0d-3` is the drop tolerance discussed in Section 2 (10^{-3} is the suggested default). `MEM=20.0d0` indicates that the preconditioner can have 20 times more nonzero entries than the original matrix, if needed. Note that the corresponding memory is dynamically allocated by JADAMILU. `ICNTL` is an integer array of length 4 all of whose entries are set to zero except when fine tuning of preconditioning parameters is desired. `IPRINT=6` indicates that the user wants a standard report on standard output. `INFO` and `GAP` are the least important output parameters.

Note that all of these parameters are described in detail in the user guide, and also in the comments of the FORTRAN source PJD.f (which is reproduced in the Appendix of the user guide).

Listing 1. Simple driver to compute inner eigenvalues

```
      PROGRAM EXAMPLE
      INTEGER    N, NEIG, MADSPACE, LX
      PARAMETER (N=1000, NEIG=5, MADSPACE=20,
     +   LX=N*(3*MADSPACE+NEIG+1)+4*MADSPACE**2)
c
      INTEGER    NEIG, ISEARCH, NINIT, MADSPACE,
     +           ITER, ICNTL(4), IPRINT, INFO
      DOUBLE PRECISION EIGS(NEIG), RES(NEIG),
     +                 X(LX), SIGMA, TOL, SHIFT,
     +                 DROPTOL, MEM, GAP
      INTEGER IA(N+1),JA(2*N-1), I, K
      DOUBLE PRECISION A(2*N-1)
c     ... initialize matrix
      K=1
      DO I=1,N
         IA(I)=K
         JA(K)=I
```

```
            A(K)=DBLE(I-1)
            K=K+1
            IF (I .LT. N) THEN
               JA(K)=I+1
               A(K)=5.0D0
               K=K+1
            END IF
         END DO
         IA(N+1)=K
c        ...
         SIGMA=0.0d0
         ISEARCH=2
         NINIT=0
         ITER=1000
         TOL=1.0d-10
         DROPTOL=1.0d-3
         MEM=20.0d0
         ICNTL(1)=0
         ICNTL(2)=0
         ICNTL(3)=0
         ICNTL(4)=0
         IPRINT=6
c
         CALL PJD(N,A,JA,IA,EIGS,RES,X,LX,NEIG,
     +            SIGMA,ISEARCH,NINIT,MADSPACE,
     +            ITER,TOL,SHIFT,DROPTOL,MEM,ICNTL,
     +            IPRINT,INFO,GAP)
         END
```

Upon completion, PJD returns the desired eigenvalues in `EIGS(1)`,..., `EIGS(5)` ; `X(1+N*(I-1):N*I)`, for `I=1`,...,`5` contains the associated eigenvectors.

For this sample program, the output is as follows.

```
**ENTERING PJDREVCOM************************************

Computing the eigenvalue(s) closest to  0.00000E+00
---------------------------------------------------

- Setting up a new ILU preconditioner ------------------
 fill-in factor:     1.18 (=nnz(LU)/nnz(A))
   used droptol:   1.00000E-03
     used shift:  0.00000E+00
  ILU computed:      0.00[sec]
--------------------------------------------------------

        Eigenvalue    Res. norm   Error est.  No(Cluster)
-4.181309490462310E+00     2.2E-11     5.8E-22     -2(---) (*)
-1.882982191624710E+00      7.7E-11     3.0E-21     -1(---)
 1.031502327791123E-01     4.3E-11     1.0E-21      1(---)
 1.877779738954345E+00     3.5E-11     7.4E-22      2(---)
 3.492268220684322E+00     6.7E-11     6.6E-21      3(---)

 (Warning: error estimates are subject to several limitations,
  see documentation for details; for (*), error estimate is
  indicative only)

 Number of MATVEC needed: 93 ; Estimated gap: 7.96E-01 ; INFO: 0

CPU time for eigenvalue computation:     0.05[sec]
     CPU time for preconditioning:     0.00[sec]
                Total CPU time:     0.05[sec]

 (Suggested DROPTOL if restart needed: 1.00E-03)

**LEAVING PJDREVCOM*************************************
```

7. Conclusions

A code based on an up-to-date eigenvalue solver with an integrated multilevel ILU preconditioner has been described. This integration goes deeper than a simple interface between separate libraries. The code includes a novel feature called adaptive preconditioning: parameters needed to build the preconditioner are deduced from the ongoing eigenvalue computation and are updated if necessary. Experiments on large-scale applications show that the approach is robust.

From the user point of view, the complex numerical methods used in the code are now within reach of non-experts. Although some parameters are included to allow fine tuning, such tuning is unnecessary to achieve good performance, and the code works well as a black box solver for all of the examples we tried.

Acknowledgment

We thank M. Havet whose initial experiments with Matlab motivated this work. We also thank D. Baye and M. Hesse who provided us with their Lagrange mesh calculation code. Both referees are acknowledged for numerous suggestions based on a careful reading of our manuscript.

References

[1] G. Sleijpen, H. A. van der Vorst, A Jacobi–Davidson iteration method for linear eigenvalue problems, SIAM J. Matrix Anal. Appl. 17 (1996) 401–425.

[2] G. Sleijpen, H. A. van der Vorst, A Jacobi–Davidson iteration method for linear eigenvalue problems, SIAM Review 42 (2000) 267–293.

[3] L. Bergamaschi, G. Pini, F. Sartoretto, Computational experience with sequential and parallel, preconditioned Jacobi–Davidson for large, sparse symmetric matrices, J. Computational Physics 188 (2003) 318–331.

[4] A. Booten, D. Fokkema, G. Sleijpen, H. A. van der Vorst, Jacobi–Davidson methods for generalized MHD-eigenvalue problems, Z. Angew. Math. Mech. 76 (1996) 131–134.

[5] S. J. Cooke, B. Levush, Eigenmode solution of 2-d and 3-d electromagnetic cavities containing absorbing materials using the Jacobi–Davidson algorithm, J. Computational Physics 157 (2000) 350–370.

[6] S. Cooke, R. Shtokhamer, A. Mondelli, B. Levush, A finite integration method for conformal, structured-grid, electromagnetic simulation, J. Computational Physics 215 (2006) 321–347.

[7] T.-M. Hwang, W.-W. Lin, W.-C. Wang, W. Wang, Numerical simulation of three dimensional pyramid quantum dot, J. Computational Physics 196 (2004) 208–232.

[8] C. Liu, J.-F. Lee, Jacobi–Davidson algorithm and its application to modeling RF/microwave detection circuits, Comp. Meth. Appl. Mech. Engng. 169 (1999) 359–375.

[9] A. V. Mitin, New methods for calculations of the lowest eigenvalues of the real symmetric generalized eigenvalue problem, J. Computational Physics 161 (2000) 653–667.

[10] J. L. M. van Dorsselaer, Computing eigenvalues occurring in continuation methods with the Jacobi–Davidson QZ method, J. Computational Physics 138 (1997) 714–733.

[11] H. Voss, Iterative projection methods for computing relevant energy states of a quantum dot, J. Computational Physics 217 (2006) 824–833.

[12] W. Wang, T.-M. Hwang, W.-W. Lin, J.-L. Liu, Numerical methods for semiconductor heterostructures with band nonparabolicity, J. Computational Physics 190 (2003) 141–158.

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

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

[15] Y. Notay, Convergence analysis of inexact Rayleigh quotient iteration, SIAM J. Matrix Anal. Appl. 24 (2003) 627–644.

[16] Y. Notay, JDCG, JDRPCG and JDRPCG_GEP codes, available via `http://homepages.ulb.ac.be/~ynotay/`.

[17] G. Sleijpen, JDQR and JDQZ codes, available via `http://www.math.uu.nl/people/sleijpen`.

[18] A. Stathopoulos, PRIMME: PReconditioned Iterative MultiMethod Eigensolver, available via `http://www.cs.wm.edu/~andreas/software/`.

[19] R. Geus, Jdbsym version 0.14, http://www.inf.ethz.ch/personal/geus/software.html.

[20] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, J. Res. Nat. Bur. Standards 45 (1950) 255–282.

[21] E. R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, J. Computational Physics 17 (1975) 87–94.

[22] A. Stathopoulos, Y. Saad, C. F. Fischer, Robust preconditioning of large, sparse, symmetric eigenvalue problems, J. Comput. Appl. Math. 64 (1995) 197–215.

[23] Y. Notay, Is Jacobi-Davidson faster than Davidson ?, SIAM J. Matrix Anal. Appl. 26 (2005) 522–543.

[24] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, 2003, second ed.

[25] H. A. van der Vorst, Iterative Krylov Methods for Large Linear systems, Cambridge University Press, Cambridge, 2003.

[26] G. W. Stewart, Matrix algorithms. Vol. II, SIAM, Philadelphia, PA, 2001.

[27] D. Fokkema, G. Sleijpen, H. A. van der Vorst, Jacobi–Davidson style QR and QZ algorithms for the reduction of matrix pencils, SIAM J. Sci. Comput. 20 (1999) 94–125.

[28] M. Benzi, Preconditioning techniques for large linear systems: A survey, J. Computational Physics 182 (2002) 418–477.

[29] R. B. Morgan, D. S. Scott, Preconditioning the Lanczos algorithm for sparse symmetric eigenvalue problems, SIAM J. Sci. Comput. 14 (1993) 585–593.

[30] R. B. Morgan, D. S. Scott, Generalizations of Davidson's method for computing eigenvalues of sparse symmetric matrices, SIAM J. Sci. Statist. Comput. 7 (1986) 817–825.

[31] C. W. Murray, S. C. Racine, E. R. Davidson, Improved algorithms for the lowest few eigenvalues and associated eigenvectors of large matrices, J. Computational Physics 103 (1992) 382–389.

[32] A. Stathopoulos, Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue, preprint, Department of Computer Science, College of William & Mary, Williamsburg, USA (2005).

[33] A. Stathopoulos, Y. Saad, Restarting techniques for the (Jacobi–)Davidson symmetric eigenvalue methods, Electronic Trans. Numer. Anal. 7 (1998) 163–181.

[34] L. Bergamaschi, G. Gambolati, G. Pini, Asymptotic convergence of conjugate gradient methods for the partial symmetric eigenproblem, Numer. Lin. Alg. Appl. 4 (1997) 69–84.

[35] A. V. Knyazev, Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method, SIAM J. Sci. Comput. 23 (2001) 517–541.

[36] U. Hetmaniuk, R. Lehoucq, Basis selection in lobpcg, J. Computational Physics 218 (2006) 324–332.

[37] R. B. Morgan, Preconditioning eigenvalues and some comparison of solvers, J. Comput. Appl. Math. 123 (2000) 101–115.

[38] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. A. van der Vorst (Eds.), Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, Philadelphia, PA, 2000.

[39] B. N. Parlett, The Symmetric Eigenvalue Problem, SIAM, Philadelphia, PA, 1998, corrected reprint of the 1980 original.

[40] H. A. van der Vorst, Computational methods for large eigenvalue problems, in: Handbook of numerical analysis, Vol. VIII, North-Holland, Amsterdam, 2002, pp. 3–179.

[41] M. R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, J. Res. Nat. Bur. Standards 49 (1952) 409–436.

[42] R. W. Freund, N. M. Nachtigal, Software for simplified Lanczos and QMR algorithms, Appl. Num. Math. 19 (1995) 319–341.

[43] M. E. Hochstenbach, Y. Notay, Controling inner iterations in the Jacobi–Davidson method, Tech. Rep. GANMN 06–03, Université Libre de Bruxelles, Brussels, Belgium, `http://homepages.ulb.ac.be/~ynotay` (2006).

[44] G. Sleijpen, H. A. van der Vorst, E. Meijerink, Efficient expansion of subspaces in the Jacobi–Davidson method for standard and generalized eigenproblems, Electronic Trans. Numer. Anal. 7 (1998) 75–89.

[45] Y. Notay, Inner iterations in eigenvalue solvers, Tech. Rep. GANMN 05–01, Université Libre de Bruxelles, Brussels, Belgium, `http://homepages.ulb.ac.be/~ynotay` (2005).

[46] M. Bollhöfer, Y. Saad, Multilevel preconditioners constructed from inverse–based ILUs, SIAM J. Sci. Comput. 27 (5) (2006) 1627–1650.

[47] P. Amestoy, T. A. Davis, I. S. Duff, An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Appl. 17 (4) (1996) 886–905.

[48] I. S. Duff, J. Koster, The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, SIAM J. Matrix Anal. Appl. 20 (1999) 889–901.

[49] M. Benzi, J. C. Haws, M. Tůma, Preconditioning highly indefinite and nonsymmetric matrices, SIAM J. Sci. Comput. 22 (4) (2000) 1333–1353.

[50] I. S. Duff, S. Pralet, Strategies for scaling and pivoting for sparse symmetric indefinite problems, SIAM J. Matrix Anal. Appl. 27 (2) (2005) 313–340.

[51] O. Schenk, K. Gärtner, On fast factorization pivoting methods for sparse symmetric indefinite systems, Elect. Trans. Numer. Anal 23 (2006) 158–179.

[52] O. Schenk, K. Gärtner, On fast factorization pivoting methods for sparse symmetric indefinite systems, Electr. Trans. Num. Anal. 23 (2006) 158–179.

[53] M. Hagemann, O. Schenk, Weighted matchings for preconditioning symmetric indefinite linear systems, SIAM J. Sci. Comput. 28 (2) (2006) 403–420.

[54] O. Schenk, M. Bollhöfer, R. A. Römer, On large scale diagonalization techniques for the Anderson model of localization, SIAM J. Sci. Comput. 28 (3) (2006) 963–983.

[55] Y. Saad, ILUM: a parallel multi-elimination ILU preconditioner for general sparse matrices, SIAM J. Sci. Comput. 17 (1996) 830–847.

[56] T. A. Manteuffel, An incomplete factorization technique for positive definite linear systems, Math. Comp. 34 (1980) 473–497.

[57] Y. Saad, Numerical Methods for Large Eigenvalue Problems, Halstead Press, New York, 1992.

[58] N. J. Higham, Accuracy and Stability of Numerical Algorithms, SIAM, Philadelphia, PA, 1996.

[59] R. Mathias, Quadratic residual bounds for the Hermitian eigenvalue problem, SIAM J. Matrix Anal. Appl. 19 (1998) 541–550.

[60] U. Trottenberg, C. W. Oosterlee, A. Schüller, Multigrid, Academic Press, London, 2001.

[61] R. B. Lehoucq, D. C. Sorensen, C. Yang, ARPACK: software package based on implicitly restarted Arnoldi methods, available via `http://www.caam.rice.edu/software/ARPACK/`.

[62] R. B. Lehoucq, D. C. Sorensen, C. Yang, ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods, SIAM, Philadelphia, PA, 1998.

[63] O. Schenk, K. Gärtner, Solving unsymmetric sparse systems of linear equations with PARDISO, Journal of Future Generation Computer Systems 20 (3) (2004) 475–487.

[64] M. Bollhöfer, Y. Saad, O. Schenk, ILUPACK — preconditioning software package, volume 2.1 (January 2006).
URL `http://www.math.tu-berlin.de/ilupack/`

[65] A. Knyazev, BLOPEX: Block Locally Optimal Preconditioned Eigenvalue Xolvers, available via `www-math.cudenver.edu/~aknyazev/software/BLOPEX/`.

[66] D. Baye, M. Hesse, M. Vincke, The unexplained accuracy of the Lagrange–mesh method, Phys. Rev. E 65 (2002) 026701(1–8).

[67] M. Hesse, D. Baye, Lagrange–mesh calculations of three–body atoms and molecules, At. Mol. Opt. Phys. 32 (1999) 5605–5617.

[68] A. Stathopoulos, C. F. Fischer, A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix, Comput. Phys. Comm. 79 (1994) 268–290.

[69] P. Anderson, Absence of diffusion in certain random lattices, Phys. Rev. 109 (1958) 1492–1505.

[70] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, SIAM Publications, 2003.