Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Computational Physics 188 (2003) 318–331

JOURNAL OF COMPUTATIONAL PHYSICS

www.elsevier.com/locate/jcp

Computational experience with sequential and parallel, preconditioned Jacobi–Davidson for large, sparse symmetric matrices

Luca Bergamaschi [a], Giorgio Pini [a], Flavio Sartoretto [b,*]

[a] *Dipartimento di Metodi e Modelli Matematici per le Scienze Applicate, Università di Padova, Via Belzoni, 7, 35131 Padova PD, Italy*
[b] *Dipartimento di Informatica Università di Venezia, Via Torino 155, 30171 Mestre VE, Italy*

Received 7 February 2002; received in revised form 22 November 2002; accepted 3 March 2003

Abstract

The Jacobi–Davidson (JD) algorithm was recently proposed for evaluating a number of the eigenvalues of a matrix. JD goes beyond pure Krylov-space techniques; it cleverly expands its search space, by solving the so-called *correction equation*, thus in principle providing a more powerful method. Preconditioning the Jacobi–Davidson correction equation is mandatory when large, sparse matrices are analyzed. We considered several preconditioners: Classical block-Jacobi, and IC(0), together with *approximate inverse* (AINV or FSAI) preconditioners. The rationale for using approximate inverse preconditioners is their high parallelization potential, combined with their efficiency in accelerating the iterative solution of the correction equation. Analysis was carried on the sequential performance of preconditioned JD for the spectral decomposition of large, sparse matrices, which originate in the numerical integration of partial differential equations arising in physical and engineering problems. It was found that JD is highly sensitive to preconditioning, and it can display an irregular convergence behavior. We parallelized JD by data-splitting techniques, combining them with techniques to reduce the amount of communication data. Our own parallel, preconditioned code was executed on a dedicated parallel machine, and we present the results of our experiments. Our JD code provides an appreciable parallel degree of computation. Its performance was also compared with those of PARPACK and parallel DACG.
© 2003 Elsevier Science B.V. All rights reserved.

AMS: 65F15; 65F50

Keywords: Eigenvalues; Sparse approximate inverses; Parallel algorithms; Jacobi–Davidson method

* Corresponding author. Fax: +39-049-827-5995.
E-mail address: sartoret@dmsa.unipd.it (F. Sartoretto).

0021-9991/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0021-9991(03)00190-6

1. Introduction

An important task in many scientific applications is the computation of a small number of the leftmost eigenpairs (the smallest eigenvalues and corresponding eigenvectors) of the problem $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, where $\mathbf{A}$ is a large, sparse, symmetric positive definite (SPD) matrix. Several techniques for solving this problem have been proposed: Subspace iteration [3,31], Lanczos method [14,23,30]; more recently, the restarted Arnoldi–Lanczos algorithm [24], the Jacobi–Davidson method [34], hereafter labeled JD, and optimization methods by Conjugate Gradient schemes [5,21,33].

In this paper we examine the performance of JD method, which was analyzed in [34] by Sleijpen and Van der Vorst. This paper is not intended to tune JD parameters, a common task in literature, that we accomplished for our problems e.g. in [10]. Our main aim is to show the difficulties that even an expert in eigenvalue computations, not familiar with JD, can face with. The main contribution of the paper is to report on computational experience with various preconditioners on both sequential and parallel computers, when solving real-life problems. Computational experiences with Jacobi–Davidson on parallel computers are not yet widespread in literature. Our paper is a contribution to this important topic.

JD is a Rayleigh–Ritz method which relies upon the solution of the so-called *correction equation* [34] to enlarge its search space. When large, sparse matrices are considered, an iterative, preconditioned Krylov solver is a suitable choice for computing a solution. We exploit four types of preconditioners, classical block-Jacobi, Incomplete Cholesky, IC(0) [27], the factorized approximate inverse AINV [4], and FSAI [22].

Using our optimization method, called DACG [7], we found that in sequential runs IC(0) is the most effective preconditioner for solving shift-invert equations involving our large, sparse matrices (see [10]), thus one can guess it is a suitable choice also for JD. However, in view of developing a parallel code, it must be remembered that IC(0) preconditioning is not easy to parallelize efficiently. Therefore we also consider Block-Jacobi, AINV and FSAI preconditioners, which are much more efficiently parallelized. We studied the performance of sequential Block-Jacobi-JD, ILU(0)-JD, AINV–JD, and FSAI–JD algorithms in the eigenanalysis of Finite Element, Mixed Finite Element, and Finite Difference 2D and 3D models arising in real-life flow problems.

The JD technique, being oriented to large, sparse matrices, is especially suitable to a parallel environment. Some recent references on parallel (or parallelizable) variants of JD are [17,28]. We implemented a parallel version of the JD algorithm via a data-parallel approach, allowing preconditioning by any given approximate inverse. Ad hoc data-distribution techniques were also implemented, in order to reduce the amount of communication among processors, which could spoil the parallel performance of the ensuing code. An efficient routine for performing parallel matrix–vector products was designed and carried out. These parallelization techniques were successfully exploited in parallelizing our DACG code for the eigensolution of sparse matrices [8].

Our parallel preconditioned JD code was used in the solution of several problems, arising in physics and engineering. Numerical tests on a Cray T3E Supercomputer show the appreciable degree of parallelization attainable by our code.

This paper is organized as follows. Section 2 gives an outline of the JD algorithm. Section 3 describes the preconditioners. Numerical results are discussed in Section 4. Conclusions are drawn in Section 5.

2. Preconditioned Jacobi–Davidson algorithm

Let us consider the eigenproblem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}, \tag{1}$$

where $\mathbf{A}$ is a large, sparse SPD, $N \times N$ matrix. The eigenvalues and corresponding eigenvectors are denoted by $\lambda_1 \leqslant \lambda_2 \leqslant \cdots \leqslant \lambda_N$, and $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N$, respectively.

It is assumed that we need to compute $k_{\max}$ eigenpairs, $(\lambda_i, \mathbf{u}_i)$, $i = 1, \dots, k_{\max}$, which are close to a given value θ.

The JD algorithm computes each eigenvalue, λ_j, by taking some Ritz values in a search space which is iteratively expanded by search vectors, $\mathbf{t}$. They are the solutions of the so called *correction equation* [34]

$$\mathbf{P}(\mathbf{A} - \tilde{\lambda}\mathbf{I})\mathbf{P}\mathbf{t} = -\mathbf{r}, \quad \mathbf{P} = (\mathbf{I} - \mathbf{Q}\mathbf{Q}^{\mathrm{T}}), \quad \mathbf{t} \perp \mathrm{span}\{\mathbf{Q}\}. \tag{2}$$

Table 1
JD Algorithm (unpreconditioned Jacobi–Davidson) for computing $k_{\max}$ exterior eigenpairs

```
begin
  Set $\mathbf{V}_{:,0}$ starting vector and $\theta$ reference value; Set $\tau, n_{it,\max}, m_{\min}, m_{\max}$;
  $\mathbf{t} := \mathbf{V}_{:,0}, \quad k := 0, \quad m := 0, \quad \mathbf{U} := [\,], \quad n_{it} := 0;$
  while $(k < k_{\max})$ and $(n_{it} < n_{it,\max})$ do $n_{it} := n_{it} + 1;$
    for $i := 1, m$ do
      $\mathbf{t} := \mathbf{t} - (\mathbf{V}_{:,i}^{\mathrm{T}}\mathbf{t})\mathbf{V}_{:,i};$
    endfor
    $m := m + 1, \quad \mathbf{V}_{:,m} = \mathbf{t}/\|\mathbf{t}\|_2, \quad \mathbf{V}_{:,m}^{A} = \mathbf{A}\mathbf{V}_{:,m};$
    for $i := 1, m$ do
      $\mathbf{M}_{i,m} := \mathbf{V}_{:,i}^{\mathrm{T}}\mathbf{V}_{:,m}^{A};$
    end for
    Compute the eigenpairs $(\mu_i, \mathbf{s}_i)$ of the matrix $\mathbf{M}$, $\|\mathbf{s}_i\|_2 = 1$;
    Sort the pairs $(\mu_i, \mathbf{s}_i)$ such that $|\mu_i - \theta| \geqslant |\mu_{i-1} - \theta|$;
    $\mathbf{u} := \mathbf{V}\mathbf{s}_1, \quad \mathbf{u}^A := \mathbf{V}^A\mathbf{s}_1, \quad \mathbf{r} := \mathbf{u}^A - \mu_1\mathbf{u};$
    while $(\|\mathbf{r}\|_2 \leqslant \tau \cdot \mu_1)$ and $(k < k_{\max})$ do
      $\tilde{\lambda}_{k+1} := \mu_1, \quad \mathbf{U} := [\mathbf{U} \mid \mathbf{u}], \quad k := k + 1, \quad n_{it} := 0;$
      if $k < k_{\max}$
        then $m := m - 1, \quad \mathbf{M} := 0;$
          for $i := 1, m$ do
            $\mathbf{V}_{:,i} := \mathbf{V}\mathbf{s}_{i+1}, \quad \mathbf{V}_{:,i}^{A} := \mathbf{V}^A\mathbf{s}_{i+1}; \mathbf{M}_{i,i} := \mu_{i+1}, \quad \mathbf{s}_i := \mathbf{e}_i, \quad \mu_i := \mu_{i+1};$
          endfor
          $\mathbf{u} := \mathbf{V}_{:,1}, \quad \mathbf{r} := \mathbf{V}_{:,1}^{A} - \mu_1\mathbf{u};$
      endif
    endwhile
    if $k < k_{\max}$
      then
        if $m \geqslant m_{\max}$
          then $\mathbf{M} = 0;$
            for $i := 2, m_{\min}$ do
              $\mathbf{V}_{:,i} := \mathbf{V}\mathbf{s}_i, \quad \mathbf{V}_{:,i}^{A} := \mathbf{V}^A\mathbf{s}_i, \quad \mathbf{M}_{i,i} := \mu_i;$
            endfor
            $\mathbf{V}_{:,1} := \mathbf{u}, \quad \mathbf{V}_{:,1}^{A} := \mathbf{u}^A, \quad \mathbf{M}_{1,1} := \mu_1, \quad m := m_{\min};$
        endif
        $\mu := \mu_1, \quad \mathbf{Q} := [\mathbf{U} \mid \mathbf{u}], \mathbf{P} = (\mathbf{I} - \mathbf{Q}\mathbf{Q}^{\mathrm{T}});$
        Approximately solve for $\mathbf{t} \perp \mathrm{span}\{\mathbf{Q}\}$ the problem $\mathbf{P}(\mathbf{A} - \mu\mathbf{I})\mathbf{P}\mathbf{t} = -\mathbf{r};$
    endif
  endwhile
  if $k < k_{\max}$
    then print 'less than $k_{\max}$ eigenpairs computed';
  endif
end
```

Here, $\tilde{\lambda}$ is a Ritz value, $\mathbf{r}$ is the current residual, $\mathbf{Q}$ is the matrix whose columns are the already computed eigenvectors, $\mathbf{u}_1, \ldots, \mathbf{u}_{j-1}$, together with a suitable Ritz vector, $\mathbf{u}$. In principle, expanding by $\mathbf{t}$ provides better eigenvector guesses than the classical Krylov space. Each vector $\mathbf{t}$ can also be interpreted as an approximated Newton correction step [39]. We implemented the JD algorithm shown in Table 1. The given pseudo-code is essentially a well-structured casting of algorithm 4.17 in [2], which is suited for computing either the smallest eigenpairs (setting $\theta \leqslant \lambda_1$) or the largest ones (setting $\theta \geqslant \lambda_N$). Note that for a given matrix $\mathbf{M}$, the notation $\mathbf{M}_{:,i}$ stands for its ith *column*, while $[\mathbf{M}|\mathbf{v}]$, $\mathbf{v}$ being a column vector, means columnwise catenation.

Since the convergence of JD is not guaranteed, we inserted a parameter, $n_{it,\max}$, setting the maximum number of iterations allowed.

We found that in our problems Modified Gram–Schmidt orthogonalization is satisfactorily accurate, and thus no iterative refinement [2] was needed.

The JD algorithm needs to assess several parameter values. Some of them are common to each iterative eigensolver: The tolerance, τ, on a suitable measure of accuracy, and the maximum number of iterations, $n_{it,\max}$. Besides these, to avoid unsatisfiable memory requirements, JD needs the maximum and minimum dimensions, $m_{\max}$ and $m_{\min}$, of the search space, S. When $\dim(S) = m_{\max}$, our JD algorithm performs a restart step, keeping those $m_{\min}$ vectors corresponding to the Ritz values which are close to θ [2]. A discussion of the restarting techniques proposed in literature can be found in [36].

System (2) is indefinite. We computed a solution by iterative Krylov solvers, which are well suited to large, sparse matrices, and easily allow for $\mathbf{t} \perp \operatorname{span}\{\mathbf{Q}\}$, which is required by JD scheme [34]. Actually, according to [29], system (2) is positive definite in the subspace orthogonal to $\operatorname{span}\{\mathbf{Q}\}$. Thus, we also exploited JDCG, i.e. JD equipped with a Conjugate Gradient (CG) solver. System (2) *need not* to be *accurately* solved, as pointed out in [35]. We performed at most $n_{it,\max}^{(\mathrm{corr})}$ (not too large a natural number) Krylov iterations, stopping the process as soon as the 2-norm of the relative residual is smaller than a prescribed (not too small) tolerance, $\tau^{(\mathrm{corr})}$. The alternative strategy suggested in [2, p. 93], was also tested. Note that in the sequel the term *inner iterations* denotes the iterations performed to solve the correction equation (2) by Krylov methods.

System (2) is usually ill-conditioned, nearly degenerate, thus preconditioning is mandatory to achieve even poor accuracy, when a small number of iterations is allowed. JD scheme requires that the vector $\mathbf{t}$ is orthogonal to $\operatorname{span}(\mathbf{Q})$, thus projected preconditioning matrices must be considered. To efficiently perform the projections, we implemented Algorithm 4.15 on page 94 in [2]. The preconditioned system can be written

$$\tilde{\mathbf{Y}}\tilde{\mathbf{A}}\mathbf{t} = \mathbf{b}, \quad \tilde{\mathbf{A}} = \mathbf{P}(\mathbf{A} - \tilde{\lambda}\mathbf{I})\mathbf{P}, \quad \mathbf{b} = -\tilde{\mathbf{Y}}\mathbf{r}, \quad \tilde{\mathbf{Y}} = \mathbf{P}\mathbf{Y}\mathbf{P}, \tag{3}$$

where the matrix $\mathbf{Y}$ is the preconditioner.

3. Preconditioners

The matrix $\mathbf{C} = \mathbf{A} - \tilde{\lambda}\mathbf{I}$ in Eq. (2) is frequently nearly singular and indefinite, hence care must be taken in selecting a preconditioner. We computed the preconditioner for the matrix $\mathbf{A}$, rather than the whole $\mathbf{C}$. Being $\mathbf{A}$ positive definite, disregarding $\tilde{\lambda}\mathbf{I}$ allows for safe preconditioners. Moreover, the preconditioner can be computed only at process start, irrespective of the value of $\tilde{\lambda}$, which changes throughout the iterations.

Block-Jacobi, Incomplete Cholesky, AINV, and FSAI preconditioners were used.

Block-Jacobi, in the sequel called Jacobi(k) for shortness, amounts to setting $\mathbf{Y} = \mathbf{D}^{-1}$ in Eq. (3), where $\mathbf{D}^{-1}$ is the inverse of the block-diagonal matrix obtained by selecting the k-size diagonal blocks of $\mathbf{A}$, starting from the upper-left one. We made use of block dimensions 1, 2, and 3. Our experience suggests (see for example [7]) that when solving by iterative methods linear systems involving our test matrices, larger blocks do not provide appreciably faster convergence, in terms of CPU time.

Let $\mathbf{L}$ be the Incomplete Cholesky, IC(0), lower triangular matrix, i.e. the incomplete Cholesky factor of $\mathbf{A}$ [27]. IC(0)-preconditioning corresponds to choosing $\mathbf{Y} = (\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1}$, in Eq. (3).

Parallelizing the preconditioning by IC(0) requires sophisticated strategies, like graph coloring [19,20], graph partitioning and ordering [1,18], or domain decomposition methods [25], whilst approximate inverse preconditioners are more easily parallelizable [12,13]. Among these preconditioning techniques, AINV and FSAI were exploited.

The *approximate inverse preconditioner* AINV [4] and the FSAI preconditioner [22] explicitly compute an approximation to $\mathbf{A}^{-1}$, based on its sparse factorization. The pattern of FSAI was identified by annihilating the elements outside the pattern of $\mathbf{A}$, while the pattern of AINV was build by canceling out those elements whose absolute value is smaller than a drop tolerance, ϵ. The smaller ϵ is, the larger is the number of non-zero elements in the AINV preconditioning factor. A sparse approximate inverse of A, $\mathbf{Y} = \mathbf{Z}\mathbf{Z}^{\mathrm{T}}$, is then identified, where $\mathbf{Z}$ is an upper triangular matrix.

4. Numerical results

Table 2 shows the main characteristics of our test matrices, including their bandwidth, i.e. max $|i - j|$ over all $\mathbf{A}_{ij} \neq 0$, and their condition numbers, $\mathscr{K} = \lambda_N/\lambda_1$, which range from small (10^3) to large (10^7) values. We exploited JD scheme to compute a number of their smallest eigenpairs, by setting $\theta = 0$. The matrices arise from the discretization of parabolic and elliptic equations in two and three dimensions. Problem 1 is the result of Mixed Finite Element (MFE) discretization of the linearized 2D Richard's equation [9]. Problems 2, 3, and 4 arise from FE discretization of 3D flow equations [16]. Problem 5 comes from a standard 3D Laplacean discretization, applying 7-point centered Finite Differences. The latter problem is a theoretical test case, that was added as a reference. These matrices are representative of the larger set of test matrices which was analyzed.

Table 3 sketches the relative gaps,

$$\gamma_i^{(r)} = \min_{j>i,\lambda_j\neq\lambda_i} (\lambda_j - \lambda_i)/\lambda_1,$$

on the smallest $s = 10$ eigenvalues of our test matrices. Note that the $N = 216{,}000$ FD matrix discretizing the Laplacean on a regular grid, has a peculiar gap-distribution, which is different from FE and MFE matrices. In literature, testing JD on such FD matrices is usual, but reading the sequel, one can find out that it does not account for JD performance on our real-life problems.

Table 2
Main characteristics of our sample problems

Nos.	Problem	N	N_{nz}	λ_N/λ_1	Bandwidth
1	2D-MFE	28,600	142,204	7.94×10^5	239
2	3D-FE	42,189	602,085	4.21×10^3	2009
3	3D-FE	80,711	325,835	2.42×10^7	9381
4	3D-FE	152,207	2,211,955	7.26×10^4	1507
5	3D-FD	216,000	1,490,400	1.50×10^3	3600

N, matrix size; N_{nz}, number of non-zero elements in matrix A; FE, Finite Elements; FD, Finite Differences; MFE, Mixed Finite Elements.

Table 3
Relative gaps, $\gamma_i^{(r)}$, between the eigenvalues

i	28,600	42,189	80,711	152,207	216,000
1	2.681	1.881	1.901	0.066	0.999
2	1.484	1.205	0.517	0.106	0.999
3	2.912	1.802	1.240	0.060	0.999
4	3.000	0.834	0.600	0.082	0.999
5	3.478	1.284	1.259	0.165	0.663
6	0.958	0.983	0.439	0.259	0.663
7	2.935	0.690	1.569	0.052	0.663
8	1.753	1.065	0.562	0.050	0.336
9	3.661	1.440	1.605	0.050	0.336
10	2.687	0.243	0.218	0.190	0.336

4.1. Sequential JD

We performed sequential JD runs on a dedicated DEC ALPHA 500 MHz machine, with 1536 MB RAM. Our Fortran code was compiled using DIGITAL Fortran 77 v. X5.2-183 with options `-O -tune=ev6 -arch=ev6`. When only our code was running, the CPU time spent by the JD algorithm was measured, irrespective of the time needed for I/O and for computing the preconditioning factors.

Table 4 shows some features of our preconditioning factors, i.e. the number of non-zero elements and the CPU seconds for computing each factor. The time consumed is negligible with respect to the overall sequential time for computing even a small number of eigenvalues (cf. Table 7).

Concerning the choice of the Krylov solver, experiments with GMRES, CGS [32], and Bi-CGSTAB [38] were performed by our group [6]. Further experiments with TFQMR [15] were also carried out. Analyzing our ill-conditioned matrices, we found that usually when TFQMR, GMRES and CGS are exploited, JD either fails to converge or provides slower convergence than using Bi-CGSTAB. In the sequel, the correction equation in JD will be solved by either (preconditioned) Bi-CGSTAB technique (JDBi-CGSTAB, hereafter called for shortness JD), or CG solver (JDCG).

Table 4
Number of nonzero entries, $N_{\mathrm{nz}}^{(M)}$, in the preconditioning factors

N		IC(0)	AINV(ϵ)		
			$\epsilon = 0.1$	$\epsilon = 0.05$	$\epsilon = 0.025$
28,600	$N_{\mathrm{nz}}^{(M)}$	85,402	201,765	464,705	790,632
	$T(M)$	0.02	0.31	0.54	0.90
42,189	$N_{\mathrm{nz}}^{(M)}$	322,137	141,956	341,523	601,360
	$T(M)$	0.11	0.53	0.69	1.08
80,711	$N_{\mathrm{nz}}^{(M)}$	203,273	1,640,603	1,641,085	1,712,476
	$T(M)$	0.04	1.15	1.16	1.22
152,207	$N_{\mathrm{nz}}^{(M)}$	1,182,081	2,594,714	3,222,199	3,798,763
	$T(M)$	0.30	4.74	6.36	8.38
216,000	$N_{\mathrm{nz}}^{(M)}$	853,200	853,200	853,200	2,732,760
	$T(M)$	0.26	1.94	1.94	5.89

$T(M)$ gives the CPU seconds spent to sequentially compute each preconditioner.

Concerning the choice of the parameters in JD, several combinations of values were tested. Our results suggest that suitable choices for analyzing our test matrices are: tolerance on the 2-norm of the residual, $\tau = 10^{-3}$; search space sizes $m_{\min} = 5$, $m_{max} = 20$; stopping parameters for inner iterations $n_{it\max}^{(\mathrm{corr})} = 26$, $\tau^{(\mathrm{corr})} = 10^{-1}$.

A tolerance $\tau = 10^{-3}$ is much larger than the square root of the *machine precision*, u, which is proposed in [2]. However, such a tolerance is too a stringent requirement, which is far smaller than the one required in real-life problems. In the sequel, further comments will be given.

We did not use *thick* restart [36] ($m_{\min} = 5 < s = k_{\max} = 10$ holds true). Our experiments suggest that in our problems this technique is time consuming, while it does not greatly improve the performance of the method. Concerning the choice of search-space dimensions, note that setting $m_{\min} \geqslant s$, as proposed in the literature (see e.g. [2]) does not appreciably change the convergence.

The starting vector was set to $\mathbf{V}_{:,0} = (1, \ldots, 1)^{\mathrm{T}}$.

Table 7 reports the CPU seconds for computing $s = 10$ eigenpairs. Inspecting Table 7, we see that IC(0)-JD and AINV–JD are more robust than FSAI-JD and Jacobi(k)-JD, which do not converge when $N = 80{,}711$ and $N = 152{,}207$.

Focusing on AINV–JD, let us analyze how its efficiency depend upon the dropping parameter ϵ. We must remember that the number of non-zero elements in AINV increases for smaller ϵ, thus increasing the cost of applying the preconditioner. Table 5 shows the average number of inner iterations per outer iteration, $\mathscr{A}_{in}$. On inspection of this table, we can see that for problems 1, 2, and 3, switching from $\epsilon = 0.1$ to $\epsilon = 0.025$, produces a better preconditioner, since $\mathscr{A}_{\mathrm{in}}$ becomes smaller. On the other hand, the number of non-zero elements usually increases with ϵ (cf. Table 4), thus the computational cost of multiplying the preconditioning factor times a vector *increases*. Such increase is not counterbalanced by the decrease in the number of outer iterations, which is reported in Table 6. In four cases (cf. Table 7), when $\epsilon = 0.025$ the CPU time is larger than for $\epsilon = 0.1, 0.05$. However, when $N = 152{,}207$ the opposite is true; when ϵ increases the efficiency of AINV does not appreciably decrease, since $\mathscr{A}_{\mathrm{in}}$ does not decrease (cf. Table 5), but the convergence of AINV–JD turns out to be slower (cf. Table 6). We found that high dependence on the preconditioner is peculiar to the JD scheme.

Taking these results into consideration, in our parallel computations we set $\epsilon = 0.1$, when computing the AINV factor.

Table 5 shows that when $N = 42{,}189$ $\mathscr{A}_{\mathrm{in}}$ is smaller when Jacobi(3) is exploited. We also found that in the same case the norm of the final residual of the correction equation is slightly smaller than employing the other preconditioners (not shown in this paper). However, Table 6 shows that, again for $N = 42{,}189$, the largest number of outer iterations occurs when Jacobi(3) is the preconditioner. This result shows that a more effective preconditioner, which provides a better solution to the correction equation, does not necessarily give faster convergence of JD.

Table 5
Computation of $s = 10$ eigenpairs using AINV(ϵ), FSAI, Jacobi(k), and IC(0)

N	AINV			FSAI	IC(0)	Jacobi(k)		
	$\epsilon = 0.025$	$\epsilon = 0.05$	$\epsilon = 0.10$			$k = 1$	$k = 2$	$k = 3$
28,600	15.62	15.50	16.07	13.81	13.55	1.09	5.49	7.16
42,189	21.83	21.83	24.45	16.75	19.48	18.90	12.80	12.16
80,711	4.67	10.82	13.07	*	13.32	*	*	*
152,207	1.01	1.00	1.00	*	3.31	*	*	*
216,000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Average number of inner iterations performed in one (outer) iteration, $\mathscr{A}_{\mathrm{in}}$. The symbol "*" means no convergence attained within 1000 iterations.

Table 6
Same as Table 5, concerning outer iterations

N	AINV			FSAI	IC(0)	Jacobi(k)		
	$\epsilon = 0.025$	$\epsilon = 0.05$	$\epsilon = 0.10$			$k = 1$	$k = 2$	$k = 3$
28,600	34	36	43	78	42	1609	250	204
42,189	41	30	31	32	33	40	56	58
80,711	90	44	40	*	37	*	*	*
152,207	375	555	669	*	39	*	*	*
216,000	373	411	411	376	256	460	790	747

Table 7
Same as Table 5, concerning CPU seconds

N	AINV			FSAI	IC(0)	Jacobi(k)		
	$\epsilon = 0.025$	$\epsilon = 0.05$	$\epsilon = 0.10$			$k = 1$	$k = 2$	$k = 3$
28,600	46.10	34.01	28.65	49.83	24.94	129.56	52.70	54.41
42,189	79.16	50.94	55.13	44.82	66.99	36.86	45.25	45.98
80,711	139.35	114.56	120.54	*	64.87	*	*	*
152,207	567.69	767.69	810.65	*	89.17	*	*	*
216,000	522.48	429.62	433.35	388.75	369.10	329.27	725.25	726.28

We also considered changing the stopping criterion in Table 1 to e.g. $\|\mathbf{r}\|_2 \leqslant \tau \cdot \min(\mu_1, 1)$, in order to reduce the tolerance when the matrix has large eigenvalues. This new criterion changes our results only for the $N = 216{,}000$ matrix, leading to more accurate eigenvalues. Once more, the eigenvalues are *not* computed in ascending order.

Alternative stopping criteria for the inner iterations are proposed in [2,29,37]. Table 8 shows the results obtained using the tolerance suggested in [2],

$$\tau^{(\mathrm{corr})} = 2^{-n_{it}^{(\mathrm{corr})}}, \tag{4}$$

which depends on the number of inner iterations, $n_{it}^{(\mathrm{corr})} \leqslant n_{it,\max}^{(\mathrm{corr})} = 26$. Only the $N = 28{,}600$ and $N = 42{,}189$ matrices were considered. Indeed, the $N = 216{,}000$ problem is not interesting, since the result $\mathscr{A}_{\mathrm{in}} = 1$ from Table 5 holds true also using the new criterion. Using four preconditioners, out of our eight ones, JD does not converge when $N = 80{,}711$ and $N = 152{,}207$ (see Table 5), hence these cases gather low information. Inspecting Table 8, one can see that a substantial change in both $\mathscr{A}_{\mathrm{in}}$ and *it* turns out only when Jacobi(k) is used. This result can be ascribed to poor efficiency of this preconditioner, which means higher $n_{it}^{(\mathrm{corr})}$ values than with other preconditioners (see Table 5), and thus a smaller $2^{-n_{it}^{(\mathrm{corr})}}$ tolerance. In any case, from Table 8 one can infer that the total number of inner iterations, $\mathscr{A}_{\mathrm{in}} \times it$, for computing $s = 10$ eigenpairs, does not depart more than 10% from $\mathscr{A}_{\mathrm{in}}^{V} \times it^{V}$. Hence the variable tolerance (4) does not provide substantial changes on JD behavior in our problems.

It is interesting to observe that JD usually computes the eigenvalues in an awkward order, when a large tolerance on eigenvalues is admitted. Let us consider the $N = 216{,}000$ matrix. Setting $\tau = 10^{-3}$, $\tau^{(\mathrm{corr})} < 10^{-1}$, using any of the proposed preconditioners, JD computes first an approximation, $\tilde{\lambda}_1$, to the smallest true eigenvalue, λ_1. However, the 6th eigenvalue approximation, $\tilde{\lambda}_6$, computed by Jacobi(3)-JD, is an approximation to the true eigenvalue λ_{20}; IC(0)-JD gives $\tilde{\lambda}_6 = \lambda_{11}$; AINV(0.1)-JD gives $\tilde{\lambda}_6 = \lambda_5 \equiv \lambda_6 \equiv \lambda_7$. These results do not come from errors in our implementation of JD. Both our JD code

Table 8
Computation of $s = 10$ eigenpairs for the $N = 28,600$ and $N = 42,189$ matrices

	AINV			FSAI	IC(0)	Jacobi(k)		
	$\epsilon = 0.025$	$\epsilon = 0.05$	$\epsilon = 0.10$			$k = 1$	$k = 2$	$k = 3$
$N = 28,600$								
$\mathscr{A}_{\text{in}}$	15.62	15.50	16.07	13.81	13.55	1.09	5.49	7.16
$\mathscr{A}_{\text{in}}^{V}$	15.73	16.38	15.73	14.07	13.41	1.17	8.09	8.93
it	34	36	43	78	42	1609	250	204
it^{V}	33	34	44	70	41	1414	179	148
$N = 42,189$								
$\mathscr{A}_{\text{in}}$	21.83	21.83	24.45	16.75	19.48	18.90	12.80	12.16
$\mathscr{A}_{\text{in}}^{V}$	21.34	21.77	24.53	17.81	19.18	17.81	21.03	21.13
it	41	30	31	32	33	40	56	58
it^{V}	41	30	32	32	34	43	37	39

JD with either fixed tolerance, $\tau^{(\text{corr})}$, or variable tolerance $\tau^{(\text{corr})} = 2^{-n_{it}^{(\text{corr})}}$, are considered. The average number of inner iterations is shown, $\mathscr{A}_{\text{in}}$, together with the number of iterations, it. The superscript "V" pertain to results with variable tolerance.

Table 9
Computation of $s = 10$ eigenpairs using either our IC(0)-JD code or our IC(0)-JDCG code

N	JDCG			JD		
	$\mathscr{A}_{\text{in}}$	it	CPU	$\mathscr{A}_{\text{in}}$	it	CPU
28,600	19.67	69	32.06	13.55	42	24.94
42,189	14.71	59	55.90	19.48	33	66.99
80,711	15.21	67	92.34	13.32	37	64.87
152,207	11.64	53	184.14	3.31	39	89.17
216,000	12.47	104	375.02	1.00	256	369.10
Average			147.89			123.01

The average number of inner iterations performed in one iteration, $\mathscr{A}_{\text{in}}$, number of iterations, it, and CPU seconds are shown.

and JDQR Matlab code by G. Sleijpen [1] compute identical sequences; the same holds for our JDCG code and JDCG code by Y. Notay (hereafter called JDCG-N [2]). On the other hand, setting

$$\tau < 10^{-7}, \quad \tau^{(\text{corr})} < 10^{-7}, \tag{5}$$

the eigenvalues in our problems are identified in ascending order, but as an example when $N = 216,000$, IC(0)-JD requires as many as 15 times the total number of inner iterations raised when $\tau < 10^{-3}$, $\tau^{(\text{corr})} < 10^{-1}$ (see Tables 5 and 6). Observe that in physical and engineering problems usually the settings (5) are too stringent. We conclude that when $\tau \gg \sqrt{u}$, it is advisable to compute at least twice the number of eigenpairs needed, like using Arnoldi and Lanczos techniques one must do.

Let us now consider IC(0)-JDCG (JD using as the inner solver CG technique, preconditioned by IC(0)).

Analyzing Table 9, one can see that IC(0)-JDCG is more CPU time consuming than JD, except when $N = 42,189$. Average CPU times suggest that IC(0)-JDCG is not superior to IC(0)-JD in our problems,

[1] JDQR code was downloaded from the URL http://www.math.uu.nl/people/sleijpen/.
[2] JDCG-N was downloaded from http://mnsgi.ulb.ac.be/pub/docs/jdcg/.

Table 10
Computation of $s = 10$ eigenpairs using JD, JDQR, and JDCG-N

N	JD	JDQR	JDCG-N
28,600	2523	2494	4128
42,189	2734	1307	1579
80,711	2134	2015	2042
152,207	753	1011	618
216,000	1759	1461	1441
Average	1980.6	1657.6	1961.6

IC(0) preconditioning was applied. Number of MVM plus number of applications of the preconditioner.

though CG is expected to be more apt to SPD matrices than Bi-CGSTAB. Note that when $N = 152{,}207$ and $N = 216{,}000$, a large difference between $\mathscr{A}_{\text{in}}$ produced by IC(0)-JDCG, and that one by IC(0)-JD, was recorded. Such difference comes from unmatching convergence criteria. Bi-CGSTAB is stopped on the *preconditioned* residual, while CG on the *true* residual, the latter turns out to be far larger than the former. On the other hand the two stopping criteria are peculiar to standard implementations of each scheme, thus it is reasonable to make the comparison on this ground.

Table 10 reports a measure of the computational costs of three JD variants by showing the number of matrix–vector multiplications (MVM) plus the number of IC(0) preconditioner applications. Their mean values show that, on average, our JD code well compares with JDCG-N by Y. Notay. Incidentally, the latter code performs better on the $N = 216{,}000$ FD matrix, and worse on the $N = 28{,}600$ FE matrix. In his paper [29], Notay tests his code on a similar FD matrix. Though in principle it is not the best choice for SPD problems, on average JDQR code by Sleijpen performs better. Again, theoretical results do not agree with experience on real-life problems.

4.2. Parallel JD

A standard data-parallel implementation of JD was performed.

Note that IC(0) preconditioning is not easily parallelized. Jacobi(k) and FSAI *are not robust enough* to be worth exploiting. Taking these factors into account, and the discussion in the previous section, to analyze all our matrices we run AINV–JD, setting $\epsilon = 0.1$.

We tailored the implementation of the MVM needed ($\mathbf{Av}$, $\mathbf{Zv}$, $\mathbf{Z}^{\mathrm{T}}\mathbf{v}$), for application to sparse matrices, using a technique for minimizing data communication between processors [7,11].

We performed parallel runs on the T3E 1200 machine of CINECA Consortium, which was located in Bologna, Italy. The machine was a *stand alone* system made by a set of DEC-Alpha 21164 processors. They performed at a peak rate of 1200 Mflop/s, and they were interconnected by a 3D toroidal network which had a 480 MB/s payload bandwidth along each of 6 torus directions. There were 256 processing elements, half of them having a 256-MB RAM, the other half a 128-MB RAM.

Again the initial vector was set to $\mathbf{V}_{:,0} = (1, \ldots, 1)^{\mathrm{T}}$.

We performed p-processor runs, when $p = 1, 2, 4, 8, 16, 32$.

Define the *speedup*, S_p, and *efficiency*, E_p, as

$$S_p = T_1/T_p, \quad E_p = S_p/p, \quad p = 1, 2, 4, 8, 16, 32.$$

Table 11 shows the CPU seconds and speed-up recorded when running our AINV–JD code. Fig. 1 displays its efficiency. For a given $p > 4$, it usually increases with the size of the problem. Problem 3 ($N = 80{,}711$) violates this rule; as an explanation, we must remember that with respect to problems 1 and 2,

Table 11
CPU seconds, T_p, spent for computing $s = 10$ eigenpairs on p processors by AINV–JD

N	T_1	T_2	T_4	T_8	T_{16}	T_{32}	S_2	S_4	S_8	S_{16}	S_{32}
AINV–JD											
28,600	68.4	38.0	20.8	12.8	8.6	6.9	1.8	3.3	5.3	8.0	9.9
42,189	111.6	58.5	34.2	19.1	13.0	10.2	1.9	3.3	5.8	8.6	10.9
80,711	365.8	216.3	141.9	85.4	64.1	48.9	1.7	2.6	4.3	5.7	7.5
152,207	3003.5	1594.2	844.5	458.1	276.0	191.9	1.9	3.6	6.6	10.9	15.7
216,000	767.0	402.2	207.8	113.8	66.5	40.2	1.9	3.7	6.7	11.5	19.1
ARPACK											
28,600	245.8	128.7	71.0	41.5	27.9	23.4	1.9	3.5	5.9	8.8	10.5
42,189	203.6	106.8	58.6	33.2	20.7	14.2	1.9	3.5	6.1	9.8	14.3
80,711	997.7	593.9	377.1	234.6	164.2	136.9	1.7	2.6	4.3	6.1	7.3
216,000	787.7	397.7	209.8	114.3	66.7	43.1	2.0	3.8	6.9	11.8	18.3
AINV–DACG											
28,600	122.7	66.5	37.0	21.5	13.6	10.6	1.8	3.3	5.7	9.0	11.6
42,189	106.2	56.0	29.6	17.6	10.5	8.5	1.9	3.6	6.0	10.1	12.5
80,711	379.6	230.1	165.1	101.9	70.1	51.8	1.6	2.3	3.7	5.4	7.3
216,000	644.7	339.4	176.4	95.2	53.5	31.5	1.9	3.7	6.8	12.1	20.5

On four test matrices, the CPU seconds spent by PARPACK and AINV–DACG are shown. The speedups S_p, are also reported. The results for the latter two codes are excerpts from [7].

a higher number of non-zero elements in AINV(0.1) was recorded in problem 3 (see Table 4). Such a feature, combined with the large bandwidth of AINV(0.1) in problem 3, inherited from that of **A** (see Table 2), is likely to deteriorate the parallel performance of our MVM code.

Table 11, demonstrates that a satisfactory speed-up is produced, when the amount of data on each processor is large enough. As an example, for problems 4 and 5 a fair speedup is obtained running on up to $p = 32$ processors.

As a comparison, let us consider PARPACK [3] code [26] (which was equipped with our AINV-CG implementation as the inner solver), and a parallel implementation of our AINV–DACG code [7]. Table 11 reports, after [7], the time and speedup recorded when computing $s = 10$ eigenpairs of four test matrices. One can see that AINV–JD CPU seconds are far smaller than PARPACK ones. These results confirm those in [10], where it was shown that (P)ARPACK is efficient when a large number ($s \geqslant 40$) of eigenpairs is needed. On the other hand, AINV–JD times are substantially larger than AINV–DACG ones only when the FD, $N = 216{,}000$, matrix is analyzed. On average, AINV–JD and AINV–DACG times are quite similar, suggesting that the efficiency of the former is appreciable, since it well compares with the latter one, which was developed for SPD problems. With respect to our FD problem, AINV–DACG seems to be preferred, taking into account that its eigenvalue computation sequence, also in presence of many multiple eigenvalues, usually matches ascending order. Concerning the speed-up values, the three codes display quite similar degrees of parallelism, which can be ascribed to the fact they share the same parallel linear algebra kernel.

5. Conclusions

The following points are worth emphasizing, concerning the performance of JD codes, when analyzing our elliptic problems.

[3] Release 1, 1996.

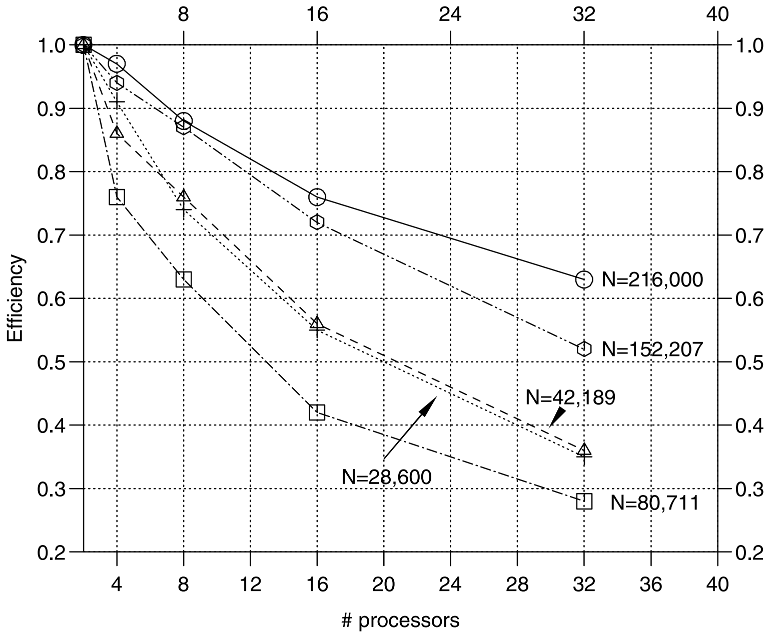

Fig. 1. Efficiency, E_p, of our parallel code, for $p = 2, 4, 8, 16, 32$.

- Our sequential computations show that AINV is usually an *effective preconditioner* for Jacobi–Davidson. When both AINV and FSAI yield convergence, they display similar performances. However, AINV is more robust than FSAI and Block-Jacobi, hence it is our recommended choice for parallel computations.
- With respect to fixed tolerance in the solution of the correction system, the $2^{-n_{it}}$ tolerance proposed in literature, does not improve the performance of JD.
- When the smallest eigenpairs are required, and quite a large tolerance is allowed, which is usual in real-life problems, JD does not necessarily compute the eigenvalues in ascending order. Moreover, the sequence heavily depend on the preconditioning technique.
- A parallel JD implementation, based upon data-parallel techniques, records a satisfactory speedup, when the number of processors is not too large with respect to the amount of data to be managed.
- A performance comparison of AINV–JD with PARPACK and AINV–DACG codes shows that they can display quite the same parallel efficiency, while on the ground of mere CPU times AINV–JD appears to be particularly apt to solve our FE and MFE eigenproblems.

Acknowledgements

This work has been supported in part by INDAM-GNCS and MURST Italian project *Mathematical Models and Numerical Methods for Environmental Fluid Dynamics*. The authors thank the anonymous referees for suggesting valuable modifications to the draft, which improved the paper.

References

[1] E.C. Anderson, Y. Saad, Solving sparse triangular systems on parallel computers, Int. J. High Speed Comput. 1 (1989) 73–96.

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

[3] K.J. Bathe, E. Wilson, Solution methods for eigenvalue problems in structural dynamics, Int. J. Numer. Methods Eng. 6 (1973) 213–226.
[4] M. Benzi, C.D. Meyer, M. Tůma, A sparse approximate inverse preconditioner for the conjugate gradient method, SIAM J. Sci. Comput. 17 (1996) 1135–1149.
[5] L. Bergamaschi, G. Gambolati, G. Pini, Asymptotic convergence of conjugate gradient methods for the partial symmetric eigenproblem, Numer. Linear Algebra Appl. 4 (1997) 69–84.
[6] L. Bergamaschi, G. Gambolati, M. Putti, Iterative methods for the partial symmetric eigenproblem, in: Proceedings of the 2000 Copper Mountain Conference on Iterative Methods, April 3–7, 2000.
[7] L. Bergamaschi, G. Pini, F. Sartoretto, Approximate inverse preconditioning in the parallel solution of sparse eigenproblems, Numer. Linear Algebra Appl. 7 (2000) 99–116.
[8] L. Bergamaschi, G. Pini, F. Sartoretto, Parallel preconditioning of a sparse eigensolver, Parallel Comput. 27 (2001) 963–976.
[9] L. Bergamaschi, M. Putti, Mixed finite elements and Newton-like linearization for the solution of Richard's equation, Int. J. Numer. Methods Eng. 45 (1999) 1025–1046.
[10] L. Bergamaschi, M. Putti, Numerical comparison of iterative eigensolvers for large sparse symmetric matrices, Comp. Methods Appl. Mech. Eng. 191 (2002) 5233–5247.
[11] L. Bergamaschi, M. Putti, Efficient parallelization of preconditioned conjugate gradient schemes for matrices arising from discretizations of diffusion equations, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, March, 1999 (CD–ROM).
[12] E. Chow, A priori sparsity patterns for parallel sparse approximate inverse preconditioners, SIAM J. Sci. Comput. 21 (2000) 1804–1822.
[13] E. Chow, Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns, Int. J. High Performance Comput. Appl. 15 (2001) 56–74.
[14] T. Ericsson, A. Ruhe, The spectral transformation Lanczos method for the numerical solution of large sparse generalized symmetric eigenvalue problems, Math. Comp. 35 (1980) 1251–1268.
[15] R.W. Freund, A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems, SIAM J. Sci. Comput. 14 (1993) 470–482.
[16] G. Gambolati, A. di Monaco, G. Galeati, F. Uliana, P. Mosca, C. Mascardi, New approaches and applications in subsurface flow modeling: 3-D finite element analysis of dewatering for an electro-nuclear plant, in: E. Custodio (Ed.), Groundwater Flow and Quality Modelling, D. Reidel Publishing Company, Dordrecht, 1988, pp. 717–759.
[17] M. Genseberger, G. Sleijpen, H. Van der Vorst, Using domain decomposition in the Jacobi–Davidson method. Universiteit Utrecht, Preprint 1164, October 2000.
[18] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, SIAM J. Sci. Comput. 22 (2001) 2194–2215.
[19] M.T. Jones, P.E. Plassman, Solution of large, sparse systems of linear equations in massively parallel applications, in: Proceedings of Supercomputing'92, IEEE Computer Society Press, Silver Spring, MD, 1992, pp. 551–560.
[20] M.T. Jones, P.E. Plassman, Scalable iterative solution of sparse linear systems, in: G.J. George, J.W.H. Liu (Eds.), Graph Theory and Sparse matrix Computation, IMA Volumes on Mathematics and Its Applications, vol. 56, Springer, Berlin, 1993, pp. 229–245.
[21] A.V. Knyazev, A.L. Skorokhodov, The preconditioned gradient-type iterative methods in a subspace for partial generalized symmetric eigenvalue problem, SIAM J. Numer. Anal. 31 (1994) 1226–1239.
[22] L.Yu. Kolotilina, A.Yu. Yeremin, Factorized sparse approximate inverse preconditioning I. Theory, SIAM J. Matrix Anal. Appl. 14 (1993) 45–58.
[23] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, J. Res. Nat. Bur. Standard 45 (1950) 255–282.
[24] R.B. Lehoucq, D.C. Sorensen, Deflation techniques for an implicit restarted Arnoldi iteration, SIAM J. Matrix Anal. Appl. 17 (1996) 789–821.
[25] M. Magolu Monga Made, H.A. van der Vorst, Parallel incomplete factorizations with pseudo-overlapped subdomains, Parallel Comput. 27 (2001) 989–1008.
[26] K.J. Maschhoff, D.C. Sorensen, A portable implementation of ARPACK for distributed memory parallel architectures, in: Proceedings of the Copper Mountain Conference on Iterative Methods, vol. 1, April 9–13, 1996.
[27] J.A. Meijerink, H.A. van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix, Math. Comp. 31 (1977) 148–162.
[28] R.T. Mills, A. Stathopoulos, E. Smirni, Algorithmic modifications to the Jacobi–Davidson parallel eigensolver to dynamically balance external CPU and memory load, in: Proceedings of the International Conference on Supercomputing 2001, Sorrento, Italy, June 18–22, 2001, pp. 454–463.
[29] Y. Notay, Combination of Jacobi–Davidson and conjugate gradients for the partial symmetric eigenproblem, Numer. Linear Algebra Appl. 9 (2002) 21–44.

[30] C.C. Paige, Computational variants of the Lanczos method for the eigenproblem, J. Inst. Math. Appl. 10 (1972) 373–381.
[31] B.N. Parlett, The Symmetric Eigenvalue Problem, Prentice-Hall, Englewood Cliffs, NJ, 1980.
[32] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Company, Boston, MA, 1996.
[33] F. Sartoretto, G. Pini, G. Gambolati, Accelerated simultaneous iterations for large finite element eigenproblems, J. Comp. Phys. 81 (1989) 53–69.
[34] G.L.G. Sleijpen, H.A. van der Vorst, A Jacobi–Davidson method for linear eigenvalue problems, SIAM J. Matrix Anal. Appl. 17 (1996) 401–425.
[35] G.L.G. Sleijpen, H.A. Van der Vorst, E. Meijerink, Efficient expansion of subspaces in the Jacobi–Davidson method for standard and generalized eigenproblems, ETNA 7 (1998) 75–89.
[36] A. Stathopoulos, Y. Saad, K. Wu, Dynamic thick restarting of the Davidson, and the implicitly restarted Arnoldi methods, SIAM J. Sci. Comput. 19 (1998) 227–245.
[37] J. Van den Eshof, The convergence of Jacobi–Davidson iterations for Hermitian eigenproblems, Numer. Linear Algebra Appl. 9 (2002) 163–179.
[38] H.A. van der Vorst, Bi-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644.
[39] K. Wu, Y. Saad, A. Stathopoulos, Inexact Newton preconditioning techniques for large symmetric eigenvalue problems, Electron. Trans. Numer. Anal. 7 (1998) 202–214.