Document downloaded from:

http://hdl.handle.net/10251/161207

This paper must be cited as:

Campos, C.; Jose E. Roman (2020). A polynomial Jacobi-Davidson solver with support for non-monomial bases and deflation. BIT Numerical Mathematics. 60(2):295-318. https://doi.org/10.1007/s10543-019-00778-z

The final publication is available at

https://doi.org/10.1007/s10543-019-00778-z

Copyright Springer-Verlag

Additional Information

A polynomial Jacobi–Davidson solver with support for non-monomial bases and deflation*

Carmen Campos[†] Jose E. Roman[‡]

August 1, 2019

Abstract

Large-scale polynomial eigenvalue problems can be solved by Krylov methods operating on an equivalent linear eigenproblem (linearization) of size $d \cdot n$, where d is the polynomial degree and n is the problem size, or by projection methods that keep the computation in the n-dimensional space. Jacobi–Davidson belongs to the latter class of methods, and, since it is a preconditioned eigensolver, it may be competitive in cases where explicitly computing a matrix factorization is exceedingly expensive. However, a fully fledged implementation of polynomial Jacobi–Davidson has to consider several issues, including deflation to compute more than one eigenpair, use of non-monomial bases for the case of large degree polynomials, and handling of complex eigenvalues when computing in real arithmetic. We discuss these aspects and present computational results of a parallel implementation in the SLEPc library.

1 Introduction

The polynomial eigenvalue problem (PEP) consists in determining eigenpairs $(x, \lambda) \in \mathbb{C}^n \times \mathbb{C}$, $x \neq 0$, satisfying

$$P(\lambda)x = 0, \tag{1}$$

where $P : \mathbb{C} \to \mathbb{C}^{n\times n}$ is a matrix polynomial of degree d,

$$P(\lambda) = A_0 + \lambda A_1 + \lambda^2 A_2 + \cdots + \lambda^d A_d, \tag{2}$$

with $A_i \in \mathbb{C}^{n\times n}$, $i = 0, \ldots, d$. The polynomial may also be expressed in a basis other than the monomial one, as we will discuss later below. We assume that P is regular, that is, $\det P(\lambda)$ is not identically zero. This problem can be found in scientific computing applications more and more [31, 3]. In this paper, we focus on the case that only a few eigenvalues λ and corresponding eigenvectors x of large-scale problems with A_i sparse are required. This is often the case in problems arising from the discretization of partial differential equations.

A common approach for solving large-scale polynomial eigenvalue problems is linearization [20], where a linear eigenvalue problem $(A - \lambda B)y = 0$ is built having the same eigenvalues λ as the polynomial problem. The corresponding eigenvectors x of (1) can be extracted from the computed vectors y. The size of the linearization is $d \cdot n$, much larger than the original problem size n. Fortunately, there exist Krylov methods that are able to exploit the structure of the

*This work was supported by Agencia Estatal de Investigación (AEI) under grant TIN2016-75985-P (including European Commission ERDF funds).

†D. Sistemes Informàtics i Computació, Universitat Politècnica de València, València, Spain (mccampos@dsic.upv.es).

‡D. Sistemes Informàtics i Computació, Universitat Politècnica de València, València, Spain (jroman@dsic.upv.es).

matrices of the linearization A, B and restrict their computational and storage requirements to order n rather than $d \cdot n$ [19, 6]. One drawback of linearization is that it may have an impact on the conditioning of eigenvalues [13]. Another downside of Krylov methods for PEP is that they need to compute a matrix factorization, in particular it is necessary to factorize $P(\sigma)$ when computing eigenvalues closest to a given target value σ. For large problems, computing a factorization may be too expensive, or have bad scalability in case of parallel computing. Sometimes this factorization can be replaced by an iterative linear solver, but this is not robust enough in most applications.

An alternative to linearization is to employ a projection method that operates on the original n-dimensional space. These methods impose a Galerkin-type condition on the residual of an eigenpair of the polynomial eigenproblem, and the resulting projected problem is also a (small-scale) PEP. Examples of such methods are SOAR [1] and Jacobi–Davidson [28]. In this paper we focus on the Jacobi–Davidson method, that has the advantage of being a preconditioned eigensolver in the sense that linear systems appearing within the method need not be solved accurately to guarantee robustness. Hence, the need of computing a factorization in Krylov methods is completely avoided and a preconditioned iterative method may be used instead.

The Jacobi–Davidson method for the PEP has been used in several application areas such as acoustics [32, 16], combustion [27], plasma physics [14], or quantum dot simulation [17]. Our goal is to provide a general-purpose, robust and efficient implementation that can be used in different contexts. Our implementation is included in SLEPc, the Scalable Library for Eigenvalue Problem Computations [23, 11], a freely available library for the solution of large-scale eigenproblems on parallel computers. SLEPc already contained an implementation of Jacobi–Davidson for linear eigenvalue problems, which is described in [24], and in this work we present a new version for the PEP case. Compared to the linear case, polynomial Jacobi–Davidson requires consideration of several important issues that we briefly discuss below.

One thing to consider is the computation of more than one eigenpair. Jacobi–Davidson computes approximations of eigenvalues one at a time. In the case of linear eigenproblems, it is possible to avoid reconvergence to previously converged eigenvalues by adding a deflation technique (also called locking). Deflation relies on the fact that eigenvectors are linearly independent, and is essentially based on performing the search in the orthogonal complement of previously converged eigenvectors. However, in polynomial eigenproblems there are $d \cdot n$ eigenvectors so linear independence cannot be guaranteed and hence deflation must be accomplished in a different way. Meerbergen [22] suggests a locking procedure based on the Schur form of the linearized problem. A more general approach was proposed more recently by Effenberger [7]. We use this latter scheme in our implementation.

Another issue is the numerical treatment of high-degree polynomials. If the degree is relatively large, e.g., $d \geq 6$, the large exponents in the monomial representation of the polynomial (2) will likely lead to numerical instability. It then turns out to be highly convenient to switch to a non-monomial basis, where the matrix polynomial is expressed as

$$P(\lambda) = \Phi_0(\lambda)A_0 + \Phi_1(\lambda)A_1 + \cdots + \Phi_d(\lambda)A_d, \tag{3}$$

where $\{\Phi_j\}_{j=0}^{d}$ is a polynomial basis with real coefficients in which Φ_j has degree j, $j = 0, \ldots, d$. In our implementation, we have considered families of orthogonal polynomials satisfying a three-term recurrence relation,

$$\lambda\, \Phi_j(\lambda) = \alpha_j\, \Phi_{j+1}(\lambda) + \beta_j\, \Phi_j(\lambda) + \gamma_j\, \Phi_{j-1}(\lambda), \quad \text{for } j = 1, 2, \ldots. \tag{4}$$

with $\Phi_{-1} \equiv 0$, $\Phi_0 \equiv 1$. In particular, our implementation supports Chebyshev, Legendre, Laguerre and Hermite polynomials (as was done for Krylov solvers in [6]). High-degree polynomials appear for instance in the solution of nonlinear eigenvalue problems via polynomial interpolation [8].

A final topic is the use of real or complex arithmetic. In many numerical algorithms, it is often convenient to perform all the computation in real arithmetic if possible. As we will discuss, SLEPc provides the option of doing all calculations either in real or complex arithmetic. Jacobi–Davidson is quite challenging for the case of real arithmetic, because the Ritz approximation is often complex even if the matrices are real, and this implies that the correction equation must deal with a complex linear system in real arithmetic. We will discuss how this can be implemented, and which implications it has with respect to the previously discussed topics (deflation and non-monomial bases).

The rest of the paper is organized as follows. Section 2 gives an overview of polynomial Jacobi–Davidson, considering the case of non-monomial polynomial bases. Section 3 deals with the deflation of converged eigenpairs, while the topic of real arithmetic is covered in section 4. The specific details of the implementation in SLEPc are discussed in section 5, and some computational results are given in section 6. We wrap up with a few concluding remarks.

2 Jacobi–Davidson for polynomial eigenvalue problems

In this section, we briefly describe the Jacobi–Davidson variant used in our solver when computing one eigenpair of a polynomial eigenvalue problem. The Jacobi–Davidson method [29] is a projection method that generates a search space using a Newton-like correction equation. This method, that was developed initially to solve linear eigenproblems, has been adapted for the solution of the polynomial and nonlinear eigenproblems [28, 15, 34]. Our implementation is based on the nonlinear versions proposed in [34, 7], which are appropriate to address polynomial eigenproblems expressed in the general form (3).

Jacobi–Davidson is a projection method based on an expanding subspace V, and hence it has two stages: on one hand the extraction phase, in which the approximate eigenpairs are obtained via a projection, and on the other hand the expansion phase, in which the search subspace is extended using corrections obtained from a Newton-type scheme. The extraction is accomplished by enforcing a Galerkin (or Petrov–Galerkin) condition, which results in a small-dimensional polynomial eigenproblem,

$$W^*P(\mu)Vy = 0. \tag{5}$$

In the case of an orthogonal projection, the left (test) and right (search) subspaces are equal, $W = V$. The solution of (5) gives Ritz approximations $(u = Vy, \mu)$. If the most wanted of these approximations is not accurate enough, the subspace is extended with a new direction. The so called correction equation has the form

$$\left(I - \frac{P'(\mu)uw^*}{w^*P'(\mu)u}\right) P(\mu)(I - uu^*)t = -P(\mu)u, \tag{6}$$

from which a vector $t \perp u$ is obtained and used to extend the search subspace. Here w is a constant vector coming from a normalization condition, orthogonal to the residual $P(\mu)u$, and P' denotes the derivative of P.

Algorithm 1 summarizes the Jacobi–Davidson method for polynomial eigenvalue problems for approximating one eigenpair. In step 2, the search subspace is initialized with a normalized random vector, and so is the test subspace, which is selected as $W = P(\sigma)V$ in this case. Step 4 extracts an approximate Ritz vector from the subspace V, whose accuracy is checked in steps 5–8. If the iteration does not stop at step 7, the search and test subspaces are extended, in steps 15 and 16 using a vector t obtained by solving the correction equation with an appropriate preconditioner in steps 12–14. When the subspaces reach a prescribed maximum dimension,

ALGORITHM 1: Computation of an eigenpair by means of polynomial Jacobi–Davidson

Input: Polynomial basis Φ and matrices $\{A_i\}_{i=0}^d \subset \mathbb{C}^{n\times n}$ defining the PEP, target value $\sigma \in \mathbb{C}$, tolerance *tol*, maximum subspace dimension m.
Output: Approximate eigenpair (x, λ), satisfying $\|P(\lambda)x\| \leq tol$.

1: Compute random $v \in \mathbb{C}^n$, $r = P(\sigma)v$
2: $V = [\,\frac{v}{\|v\|}\,]$, $W = [\,\frac{r}{\|r\|}\,]$ (*Initialization*)
3: **for** $j = 1, 2, \dots$ **do**
4: $(u, \mu) = (Vy, \mu)$ with $W^*P(\mu)Vy = 0$ for μ closest to σ (*Extraction*)
5: $r = P(\mu)u$
6: **if** $\|r\| \leq tol$ **then**
7: $(x, \lambda) = (u, \mu)$, **stop** (*Finalization*)
8: **end if**
9: **if** $\dim V \geq m$ **then**
10: Compress V, W, extract (u, μ), recompute r (*Restart*)
11: **end if**
12: Build preconditioner K from $P(\mu)$ (*Preconditioner*)
13: Form shell preconditioner $\hat{K}$ according to (9)
14: $t = \text{solve}(P(\mu), \hat{K}, -r)$ (*Correction equation*)
15: $v = (I - VV^*)t$, $w = (I - WW^*)r$ (*Orthogonalization*)
16: $V \leftarrow [V, \frac{v}{\|v\|}]$, $W \leftarrow [W, \frac{w}{\|w\|}]$ (*Subspace expansion*)
17: **end for**

a restart is forced (step 10) to compress them to a smaller dimension trying to keep the most useful spectral information.

The algorithm performs an oblique projection using a test subspace W different from the search subspace V. However, we have also implemented the possibility of doing an orthogonal projection using $W = V$, which can be found more often in literature [5, 34, 21, 17, 14]. Our choice for oblique projection is the one used by Effenberger [7], which is related to the harmonic Ritz space proposed in [30].

The resolution of the correction equation (6) is done approximately by means of a preconditioned Krylov method [26] such as BiCGStab or GMRES. The equation can be expressed as

$$M_\pi t = -r, \quad M_\pi = \pi_1 P(\mu)\pi_2, \quad t \perp u, \tag{7}$$

with $r = P(\mu)u$ and

$$\pi_1 := \left(I - \frac{P'(\mu)uw^*}{w^*P'(\mu)u}\right), \quad \pi_2 := (I - uu^*), \tag{8}$$

that is, M_π is the restriction of $M := P(\mu)$ to two subspaces, each of them of codimension one. If K is a preconditioner for M, then its restriction $K_\pi = \pi_1 K \pi_2$ is an appropriate preconditioner for M_π [34]. When applying an iterative method for solving (7), operations of the type $y = K_\pi^{-1} M_\pi v$ are performed repeatedly. Following the reasoning described in [34], it can be shown that these operations are equivalent to $y = \hat{K}^{-1}Mv$, being

$$\hat{K}^{-1} := \left(I - \frac{K^{-1}\dot{r}u^*}{u^*K^{-1}\dot{r}}\right) K^{-1}, \tag{9}$$

where $\dot{r} = P'(\mu)u$. As a consequence, we see that using an iterative method to solve the system with coefficient matrix M_π and preconditioner K_π is equivalent to applying such method to solve the system with matrix M using a preconditioner whose action is given by $\hat{K}^{-1}$. The latter is not built explicitly, but instead the elements necessary for its application are computed. This requires, on one hand, the construction of a preconditioner K for $M = P(\mu)$ (step 12 of

Algorithm 1), and, on the other hand, the elements that intervene in the projection to be applied after K^{-1} in (9), $z = K^{-1}\dot{r}$ and $\gamma = u^*z$. The operation indicated in step 14 of Algorithm 1 corresponds to the application of the Krylov method for the solution of the correction equation, using matrix $P(\mu)$ and the preconditioner defined by $\hat{K}$.

It is well known that in the context of nested iterations where the outer loop is of Newton type, there is no point in oversolving the inner iteration during the initial steps of the outer loop. For this reason, we use a variable tolerance for the Krylov method in step 14 of Algorithm 1. In particular, the convergence criterion is

$$\|r^{(k)}\|_2 \leq \max\{2^{-j}, \texttt{tol}\}\|r^{(0)}\|_2, \tag{10}$$

where $r^{(k)}$ denotes the linear system residual at step k of the Krylov iteration, j is the iteration number of the outer loop, and `tol` is the tolerance requested to the eigensolver. This strategy was already suggested in [9].

The restart of step 10 in Algorithm 1 aims at reducing the dimension of subspaces V and W from m to $\hat{p}$, to keep memory and computational requirements bounded. We implement the restart as $V \leftarrow VQ$, $W \leftarrow WS$, where Q, S are $m \times \hat{p}$ matrices with orthonormal columns spanning the primitive right and left Ritz vectors y, z corresponding to the p eigenvalues closest to the target σ, with $\hat{p} \leq p$. The p parameter can be set by the user, see section 5.4.

The process described in Algorithm 1 generates an eigenpair of the polynomial eigenvalue problem. When more than one eigenpair is required, an outer loop is added to such process, that iterates until the desired number of eigenpairs is obtained. However, it is necessary to add some type of mechanism to avoid recomputing previously computed eigenpairs. This is discussed in the following section.

3 Deflation of converged eigenpairs

There are several possible strategies to avoid reconvergence to previously computed eigenpairs. A simple scheme is to modify the extraction (step 4 of Algorithm 1) to avoid picking eigenvalues that are close (according to a tolerance) to one of the eigenvalues computed so far. Obviously, this scheme is viable only for problems with well-separated eigenvalues. A better approach is to deflate converged eigenpairs. In linear eigenproblems, deflation is usually effected by shifting or by orthogonalization. The former has been extended to polynomial problems as non-equivalence deflation [10], that maps already computed eigenvalues to a different location. In linear eigenproblems, deflation by orthogonalization is usually preferred. However, in polynomial eigenproblems maintaining the search subspace orthogonal to the previous eigenvectors may miss other eigenvalues because eigenvectors need not be linearly independent. One possible workaround is to deflate based on the Schur form of the linearization [22, 25]. The strategy that we have used is based on the deflation approach proposed by Effenberger for the general nonlinear eigenvalue problem [7].

3.1 Minimal invariant pairs

The deflation technique proposed in [7] is based on expanding invariant pairs. Invariant pairs are generalizations for the nonlinear case of invariant subspaces for linear eigenproblems. The concept of invariant pair has been defined in [18] for general nonlinear eigenproblems, and in [4] for polynomials expressed in the monomial basis. Here we give a definition particularized for non-monomial bases.

Definition 1. *A pair* $(X,H) \in \mathbb{C}^{n\times k} \times \mathbb{C}^{k\times k}$ *is said to be an invariant pair for a matrix polynomial* P *defined as in* (3) *if it verifies*

$$\mathbb{P}(X,H) := A_0 X\,\Phi_0(H) + A_1 X\,\Phi_1(H) + \cdots + A_d X\,\Phi_d(H) = 0, \tag{11}$$

where $\Phi_i(H)$ *stands for the matrix function defined in terms of the polynomial* Φ_i *[12].*

Generalizing the linear case, Kressner [18, lemma 4] shows that eigenvalues of a matrix H from an invariant pair (X,H) are also eigenvalues of the associated nonlinear eigenproblem, provided that the invariant pair is minimal. For our purpose, we give the following definition of minimal invariant pair, which is shown to be equivalent to the standard definition [7, lemma 2.4].

Definition 2. *An invariant pair* (X,H) *for a matrix polynomial* (3) *is minimal if there exists* $\ell \geq 1$ *such that*

$$V_\ell(X,H) := \begin{bmatrix} X\,\Phi_0(H) \\ X\,\Phi_1(H) \\ \vdots \\ X\,\Phi_{\ell-1}(H) \end{bmatrix} \tag{12}$$

has full column rank. The minimality index of (X,H) *is defined as the minimum value of* ℓ *satisfying this condition.*

In the case of a matrix polynomial of degree d, it is sufficient to consider invariant pairs with minimality index d at most [4, theorem 3].

3.2 Deflation in the Jacobi–Davidson solver

Effenberger [7] studies the way in which, once a minimal invariant pair $(X,H) \in \mathbb{C}^{n\times k} \times \mathbb{C}^{k\times k}$ of a nonlinear eigenproblem has been computed, an extended pair

$$(\tilde{X},\tilde{H}) = \left(\begin{bmatrix} X & x \end{bmatrix}, \begin{bmatrix} H & t \\ & \lambda \end{bmatrix} \right) \tag{13}$$

is obtained that in turn is a minimal invariant pair of the same problem. By imposing the conditions of invariant pair and minimality on the extended pair (13), a new extended (of size $n+k$) nonlinear eigenvalue problem

$$\begin{bmatrix} P(\lambda) & U(\lambda) \\ A(\lambda) & B(\lambda) \end{bmatrix} \begin{bmatrix} x \\ t \end{bmatrix} = 0 \tag{14}$$

is obtained. Its solution provides the required data for the wanted extension (13).

In this section, we give explicit expressions for the extended nonlinear eigenproblem (14) particularized for the case when a polynomial eigenvalue problem in the general form (3) is considered. In this case, the resulting problem will also be a PEP that we will solve by the Jacobi–Davidson method described in section 2. We are interested in expressing the small-dimensional PEP resulting from applying the Galerkin condition to the extended problem in terms of the same basis as the polynomial P.

In order to derive the explicit formulas that have been used in our implementation, we first impose the invariant pair condition (Definition 1),

$$0 = \sum_{i=0}^{d} A_i \begin{bmatrix} X & x \end{bmatrix} \Phi_i\left(\begin{bmatrix} H & t \\ & \lambda \end{bmatrix} \right) = \sum_{i=0}^{d} A_i \begin{bmatrix} X & x \end{bmatrix} \begin{bmatrix} \Phi_i(H) & \hat{\Phi}_i(H,\lambda)t \\ & \Phi_i(\lambda) \end{bmatrix}, \tag{15}$$

where $\hat{\Phi}_i(H,\lambda)t$ is obtained in a recursive way as follows. Using the recurrence (4) applied to $\tilde{H}$ we obtain $\Phi_{-1}(\tilde{H})=0$, $\Phi_0(\tilde{H})=I$ and, for $j\geq 0$,

$$\Phi_{j+1}(\tilde{H})=\frac{1}{\alpha_j}\left(\tilde{H}\,\Phi_j(\tilde{H})-\beta_j\,\Phi_j(\tilde{H})-\gamma_j\,\Phi_{j-1}(\tilde{H})\right), \tag{16}$$

from where we derive the wanted recurrence,

$$\begin{aligned}\hat{\Phi}_{-1}(H,\lambda)t&=0,\quad \hat{\Phi}_0(H,\lambda)t=0,\\ \hat{\Phi}_{j+1}(H,\lambda)t&=\begin{bmatrix}I_k & 0\end{bmatrix}\Phi_{j+1}(\tilde{H})e_{k+1}\\ &=\frac{1}{\alpha_j}\left(\begin{bmatrix}H & t\end{bmatrix}\begin{bmatrix}\hat{\Phi}_j(H,\lambda)t\\ \Phi_j(\lambda)\end{bmatrix}-\beta_j\hat{\Phi}_j(H,\lambda)t-\gamma_j\hat{\Phi}_{j-1}(H,\lambda)t\right)\\ &=\frac{1}{\alpha_j}\left(\Phi_j(\lambda)I_k+(H-\beta_jI_k)\hat{\Phi}_j(H,\lambda)-\gamma_j\hat{\Phi}_{j-1}(H,\lambda)\right)t,\quad j\geq 0.\end{aligned} \tag{17}$$

Defining the matrix function

$$U(\lambda):=\sum_{i=0}^{d}A_iX\hat{\Phi}_i(H,\lambda), \tag{18}$$

the condition (15) gives way to the equality $0=\begin{bmatrix}\mathbb{P}(X,H) & P(\lambda)x+U(\lambda)t\end{bmatrix}$, from which it follows (using the fact that (X,H) is an invariant pair) that the equation

$$P(\lambda)x+U(\lambda)t=0, \tag{19}$$

determines an invariant pair condition for (13).

In order to enforce the minimality requirement, we consider Definition 2, that is expressed in the same polynomial basis as the PEP. Denoting $Z:=V_\ell(\tilde{X},\tilde{H})$, each block Z^i, $i=0,1,\ldots,\ell$–1, has the form

$$Z^i=\begin{bmatrix}X & x\end{bmatrix}\Phi_i\left(\begin{bmatrix}H & t\\ & \lambda\end{bmatrix}\right)=\begin{bmatrix}X\,\Phi_i(H) & X\hat{\Phi}_i(H,\lambda)t+x\,\Phi_i(\lambda)\end{bmatrix}. \tag{20}$$

As in [7], to make sure that $V_\ell(\tilde{X},\tilde{H})$ has full column rank, we force orthogonality of the last column with respect to the previous ones, that is, to $V_\ell(X,H)$, whose columns are linearly independent since (X,H) is minimal. This results in the equation

$$0=\sum_{i=0}^{\ell-1}(X\,\Phi_i(H))^*(X\hat{\Phi}_i(H,\lambda)t+x\,\Phi_i(\lambda))=A(\lambda)x+B(\lambda)t, \tag{21}$$

where the matrix polynomials A and B are given by

$$A(\lambda):=\sum_{i=0}^{\ell-1}\Phi_i(\lambda)\,\Phi_i(H)^*X^*,\quad B(\lambda):=\sum_{i=0}^{\ell-1}\Phi_i(H)^*X^*X\hat{\Phi}_i(H,\lambda). \tag{22}$$

The expressions (22) and (18) determine the extended polynomial eigenvalue problem (14) that an extended minimal invariant pair of the form (13) must verify. Each of the four blocks of the matrix in (14) is a matrix polynomial in the polynomial basis $\{\Phi_j(\lambda)\}_{j\geq 0}$. As a consequence, we have that the new problem (14) is a PEP defined by a matrix polynomial $\tilde{P}(\lambda)$ of type (3), analog to the initial one, $P(\lambda)$. When applying the extraction phase (step 4 of Algorithm 1) to the extended problem, we want to generate a projected PEP, $W^*\tilde{P}(\lambda)V=0$, with the polynomial

expressed in the basis Φ. For this, it is necessary to determine the coefficients of the polynomial $\tilde{P}(\lambda)$ in such basis. The blocks $P(\lambda)$ and $A(\lambda)$ are already expressed in the desired form, by definition. However, matrices $U(\lambda)$ and $B(\lambda)$ have been defined in (18) and (22), respectively, in terms of the matrix functions $\hat{\Phi}_j(H,\lambda)$. The following result provides a recurrence to express these matrix polynomials in terms of the basis Φ_j.

Proposition 1. *Given $\{M_j\}_{j=0}^{d} \subset \mathbb{C}^{m\times k}$, the matrices $\{N_j\}_{j=0}^{d}$ defined by*

$$\begin{aligned} N_d &= 0, \quad N_{d-1} = \frac{1}{\alpha_{d-1}} M_d, \\ N_{j-1} &= \frac{1}{\alpha_{j-1}}\big(M_j + N_j(H - \beta_j I_k) - \gamma_{j+1} N_{j+1}\big), \quad j < d-1, \end{aligned} \tag{23}$$

verify that $\sum_{j=0}^{d} M_j \hat{\Phi}_j(H,\lambda) = \sum_{j=0}^{d} N_j\, \Phi_j(\lambda)$, for $\hat{\Phi}_j$ defined in (17).

Proof. We proceed by induction on d. For $d = 0$ we immediately see that $N_0 = 0$. For $d = 1$, using the relations (17) we have that $N_1\,\Phi_1(\lambda) + N_0\,\Phi_0(\lambda) = M_1\hat{\Phi}_1(H,\lambda) = (M_1/\alpha_0)\,\Phi_0(\lambda)$, from where it follows that $N_1 = 0$ and $N_0 = M_1/\alpha_0$. Now, supposing that the result is true up to $d-1$, we must check that the result is verified also up to order d.

Defining the matrices $\tilde{M}_{d-1} := M_{d-1} + \frac{1}{\alpha_{d-1}} M_d(H - \beta_{d-1} I_k)$, $\tilde{M}_{d-2} := M_{d-2} - \frac{\gamma_{d-1}}{\alpha_{d-1}} M_d$ and $\tilde{M}_j := M_j$, for $j < d-2$, by the induction hypothesis we have

$$\begin{aligned} \sum_{j=0}^{d} M_j \hat{\Phi}_j(H,\lambda) &= M_d \hat{\Phi}_d(H,\lambda) + \sum_{j=0}^{d-1} M_j \hat{\Phi}_j(H,\lambda) = \frac{M_d}{\alpha_{d-1}}\,\Phi_{d-1}(\lambda) + \sum_{j=0}^{d-1} \tilde{M}_j \hat{\Phi}_j(H,\lambda) \\ &= \frac{M_d}{\alpha_{d-1}}\,\Phi_{d-1}(\lambda) + \sum_{j=0}^{d-1} \tilde{N}_j\,\Phi_j(\lambda) = N_{d-1}\,\Phi_{d-1}(\lambda) + \sum_{j=0}^{d-2} \tilde{N}_j\,\Phi_j(\lambda), \end{aligned} \tag{24}$$

for matrices $\{\tilde{N}_j\}_{j=0}^{d-1}$ verifying (23) with respect to $\{\tilde{M}_j\}_{j=0}^{d-1}$. On the other hand,

$$\begin{aligned} \tilde{N}_{d-2} &= \frac{\tilde{M}_{d-1}}{\alpha_{d-2}} = \frac{1}{\alpha_{d-2}}\left(M_{d-1} + N_{d-1}(H - \beta_{d-1} I_k)\right) = N_{d-2}, \\ \tilde{N}_{d-3} &= \frac{1}{\alpha_{d-3}}\left(\tilde{M}_{d-2} + \tilde{N}_{d-2}(H - \beta_{d-2} I_k) - \gamma_{d-1}\tilde{N}_{d-1}\right) \\ &= \frac{1}{\alpha_{d-3}}\left(M_{d-2} - \frac{\gamma_{d-1}}{\alpha_{d-1}} M_d + N_{d-2}(H - \beta_{d-2} I_k)\right) = N_{d-3}, \\ \tilde{N}_j &= \frac{1}{\alpha_j}\left(\tilde{M}_{j+1} + \tilde{N}_{j+1}(H - \beta_{j+1} I_k) - \gamma_{j+2}\tilde{N}_{j+2}\right) = N_j, \quad j < d-3. \end{aligned} \tag{25}$$

Substituting (25) in (24) we have the desired result. □

For solving the correction equation associated with Jacobi–Davidson (Algorithm 1) by using an iterative Krylov method, it is necessary to perform matrix-vector multiplications with the matrix $\tilde{P}(\lambda)$ evaluated at some Ritz value μ, and apply a preconditioner for the same matrix. Effenberger [7] suggests using the preconditioner

$$\tilde{K}^{-1} = \begin{bmatrix} K^{-1} & -X(\mu I - H)^{\dagger}(B(\mu) - A(\mu)X(\mu I - H)^{\dagger})^{-1} \\ 0 & (B(\mu) - A(\mu)X(\mu I - H)^{\dagger})^{-1} \end{bmatrix} \tag{26}$$

for the matrix $\tilde{P}(\mu)$, supposing that K^{-1} is a preconditioner for the matrix $P(\mu)$.

In our implementation the extended operators (14) and (26) are not explicitly created for each value μ, but implicit operations with them are performed. In section 5.2 we give details related to the implementation of these operations using PETSc and SLEPc functionality.

A final comment is that when a Ritz value converges to working accuracy, the search subspace already contains good approximations of other eigenvectors. Hence, when locking the converged eigenvalue, the subspace must somehow be recycled to continue the search of other eigenvalues in subsequent iterations. This is accomplished by extending the vectors of the V basis by one additional element as proposed in [7]. This extension of vectors is done essentially as follows: suppose (Y, T) is an invariant pair of the projected eigenproblem, with $T = \begin{bmatrix} \lambda & t_{12} \\ & T_{22} \end{bmatrix}$ in Schur form, then the new search space for the extended problem is set to the Q-matrix obtained from an economy QR decomposition of $\begin{bmatrix} V \\ t_{12} \end{bmatrix}$. In case of an oblique projection, we take $W = P(\sigma)V$ for the test subspace.

4 Real arithmetic

The described Jacobi–Davidson method may generate a complex Ritz pair when solving a real eigenproblem, even for problems in which all the eigenvalues are real. In this case, the associated correction equation involves solving a complex linear system. The SLEPc design does not allow mixed real and complex arithmetic computations. Also, from a computational point of view, it is convenient to perform the computation in real arithmetic whenever possible, for both memory and computational efficiency. A typical case when this is important is computing real eigenvalues of a polynomial of real matrices—in this case the approximate eigenvalue may be complex in the initial iterations, and a trivial implementation in SLEPc would have to abort. Due to these considerations, we have implemented the Jacobi–Davidson polynomial eigensolver in SLEPc in a way that allows solving real eigenproblems by means of real arithmetic exclusively. In this section, we describe the modifications included to achieve this, in particular to extend an invariant pair with a converged complex Ritz pair, or to solve a complex linear system coming from the correction equation associated with a complex Ritz approximation.

First we show how a real invariant pair is extended when a complex Ritz pair (x, λ) converges. For real eigenvalue problems the convergence of complex Ritz pairs comes in conjugate pairs. Hence, in this case, we extend the invariant pair with two complex Ritz pairs.

Proposition 2. *Let* (X, H) *be a minimal invariant pair for a matrix polynomial* (3)*, and* $\{(y_i, \lambda_i)\}_{i=1}^{2}$ *two eigenpairs for the extended PEP associated with* (X, H) *with* $\lambda_1 \neq \lambda_2$*. Then,*

$$\left(\begin{bmatrix} X & x_1 & x_2 \end{bmatrix}, \begin{bmatrix} H & t_1 & t_2 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{bmatrix} \right), \quad \text{with } y_i = \begin{bmatrix} x_i \\ t_i \end{bmatrix}, \quad i = 1, 2, \tag{27}$$

is also a minimal invariant pair for (3)*.*

Proof. This is a particular case of [7, Lemma 4.3]. □

We apply the above result when extending a minimal invariant pair (X, H) with two complex conjugate eigenpairs, (y, λ) and $(\bar{y}, \bar{\lambda})$. Then, applying a similarity transformation with $Q = \frac{1}{2}\begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix}$ results in a real pair

$$\begin{gathered} \begin{bmatrix} X & \operatorname{Re} x & \operatorname{Im} x \end{bmatrix} = \begin{bmatrix} X & x & \bar{x} \end{bmatrix} \begin{bmatrix} I & \\ & Q \end{bmatrix}, \\ \begin{bmatrix} H & \operatorname{Re} t & \operatorname{Im} t \\ 0 & \operatorname{Re}\lambda & \operatorname{Im}\lambda \\ 0 & -\operatorname{Im}\lambda & \operatorname{Re}\lambda \end{bmatrix} = \begin{bmatrix} I & \\ & Q^{-1} \end{bmatrix} \begin{bmatrix} H & t & \bar{t} \\ 0 & \lambda & 0 \\ 0 & 0 & \bar{\lambda} \end{bmatrix} \begin{bmatrix} I & \\ & Q \end{bmatrix}, \end{gathered} \tag{28}$$

which is also invariant and minimal [7, Lemma 2.5] and has the eigenvalues $\lambda(H) \cup \{\lambda, \bar{\lambda}\}$.

Now we move to the situation where a non-converged complex Ritz value is generated in the extraction step of Jacobi–Davidson, and hence the corresponding correction equation becomes complex. Let us first consider the simpler case where the deflation of section 3 has not been activated yet. In order to restrict the computation to real arithmetic, we must write the correction equation (7) in a 2×2 block form,

$$\begin{bmatrix} \operatorname{Re} M_\pi & -\operatorname{Im} M_\pi \\ \operatorname{Im} M_\pi & \operatorname{Re} M_\pi \end{bmatrix} \begin{bmatrix} \operatorname{Re} t \\ \operatorname{Im} t \end{bmatrix} = - \begin{bmatrix} \operatorname{Re} r \\ \operatorname{Im} r \end{bmatrix}. \tag{29}$$

The details have been worked out in [33] for the linear eigenvalue problem. As explained in section 2, the solution of the correction equation by an iterative method involves operations of the form $y = \hat{K}_2^{-1} M_2 v$, where the matrix M_2 is the 2×2 form of $P(\mu)$ and $\hat{K}_2^{-1}$ is the 2×2 analog of the preconditioner (9), with the difference that the real part of μ is used to build the preconditioner instead of the complex μ. The expression for the preconditioner is

$$\hat{K}_2^{-1} = \Pi \begin{bmatrix} K_r^{-1} & 0 \\ 0 & K_r^{-1} \end{bmatrix}, \tag{30}$$

where Π is a projector and K_r^{-1} is a preconditioner constructed from $P(\operatorname{Re} \mu)$. The projector Π has the form

$$\Pi = \begin{bmatrix} I - z_r u_r^* - z_i u_i^* & -z_r u_i^* + z_i u_r^* \\ z_r u_i^* - z_i u_r^* & I - z_r u_r^* - z_i u_i^* \end{bmatrix}, \tag{31}$$

where $u = u_r + i u_i$ and $z = z_r + i z_i = \frac{K_r^{-1} \dot{r}}{u^* K_r^{-1} \dot{r}}$.

The above is also valid for the general case of an extended operator $\tilde{P}(\mu)$ involving deflation of previously converged eigenvalues. In addition to solving the correction equation in real arithmetic, it is also necessary to keep separate real and imaginary parts of all the quantities computed to operate with $\tilde{P}(\mu)$, such as for instance the recurrence (17). The details of how real arithmetic is done in practice in our implementation will be explained in section 5.3.

5 Implementation details in SLEPc

We now provide some details related to implementation in SLEPc, the Scalable Library for Eigenvalue Problem Computations [23, 11]. SLEPc is mainly concerned with linear eigenvalue problems, but in the last years we have included a lot of functionality for the polynomial and general nonlinear eigenproblems. The polynomial Jacobi–Davidson solver is the latest addition, that we describe in this paper.

In subsection 5.1 we describe how code is organized, and give minimal descriptions of concepts that will be required in the rest of the section. Then we focus on details regarding operations with extended operators and computation in real arithmetic.

5.1 Overview of SLEPc

SLEPc consists of several solver classes (one per different problem class) together with several auxiliary object classes. SLEPc depends on the PETSc library [2], and in particular it uses its data structures for linear algebra objects (vectors and matrices) as well as linear system solvers that are required in some eigenvalue computations. Both PETSc and SLEPc are parallel libraries based on the message passing paradigm (MPI), which in brief means that objects have their data structures distributed across several processes and most operations acting on an object are carried out in parallel by all participating processes.

PETSc's vector objects (`Vec`) provide common operations in vector space algebra such as addition and inner product, while matrix objects (`Mat`) are mainly intended to represent sparse matrices, with a certain internal representation, or also "matrix-free" matrices where the linear algebra operator is defined via a user-provided matrix-vector product subroutine (shell matrices in PETSc's terminology).

PETSc's linear solvers are gathered in the `KSP` class, which consists of a collection of iterative solvers such as GMRES or BiCGStab together with preconditioners (`PC`). Direct linear solvers are also available via "complete factorization preconditioners" such as LU. For the case of parallel LU factorization, it is necessary to have recourse to external libraries such as MUMPS.

SLEPc's module for linear eigenproblems is called `EPS`, and, as we have mentioned before, it contains an implementation of Jacobi–Davidson [24]. The solver class for polynomial eigenproblems is `PEP`, and here is where the new polynomial Jacobi–Davidson is located. `PEP` also provides several linearization-based Krylov solvers, most notably the TOAR solver, which is the default. In the computational experiments of section 6, we will compare TOAR with the newly developed solver. The TOAR solver requires the accurate solution of linear systems, usually via PETSc's LU factorization, whereas Jacobi–Davidson relies on a preconditioned iterative solver (possibly with low accuracy).

Both `EPS` and `PEP` use other auxiliary SLEPc classes that are relevant for our discussion. One of them is `BV`, which provides functionality related to bases of vectors. Essentially, `BV` allows to operate efficiently with a collection of vectors as a whole, rather than working with individual vectors. An example operation included in `BV` is the (parallel) orthogonalization of vectors. The other auxiliary class is `DS` (dense solver), that operates on small-size dense matrices that are stored redundantly (not distributed) in a group of processes. This is used in the projected problem arising in various projection algorithms. In most cases, the projected problem is a linear eigenproblem so LAPACK subroutines are invoked to solve it. However, as discussed above, Jacobi–Davidson produces a projected PEP. To support this, we have implemented a `DS` solver for small-size dense PEPs with dense coefficient matrices, in which we explicitly build the matrices of the linearization (using dense storage), call the LAPACK subroutine on them, and finally retrieve the eigenvectors of the PEP. Since we want to support non-monomial bases, we use the more general linearization explained in [6].

5.2 Operations involving the extended operators

We now proceed to discuss implementation aspects relative to the management of extended operators (14) and (26), and the way in which we have tried to minimize the overhead associated with the increase of the dimension of the extended problem, especially in terms of parallel communication.

The extended problem (14) is not built explicitly, but instead all the operations involving the extended matrix or its preconditioner (26) are carried out in an implicit way. The elements that intervene in the definition of the extended problem are: the (distributed) coefficient matrices of the initial PEP; the parallel vectors X that are stored in an object of class `BV`; and the matrix H, that is replicated in all processes. The bases of the search and test spaces are also stored in objects of class `BV`, which provides the necessary functionality for orthogonalization. From the beginning, the vectors of these subspaces are considered to have length $n+k$, being k the number of eigenvalues to compute. The last k entries are initialized to zero and progressively become nonzero as soon as eigenpairs get converged and the extended problem grows in size.

All parallel vectors are represented as follows. Given $y = \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] \in \mathbb{C}^{n+k}$, y_1 (of length n) is distributed conformally to the coefficient matrices of the original PEP, and y_2 is stored redundantly in all processes. This redundancy aims at avoiding the communication that would be required if y_2 was stored in just one process. From the point of view of the implementation,

the solvers operate with PETSc vector objects whose global length is $n + k \cdot p$, where p is the number of processes. The structure of such vectors will be taken into account in all operations described below, so that the algorithm always computes the same quantities irrespective of the number of processes.

The main parallel operations that appear in the execution of the Jacobi–Davidson method of Algorithm 1 are:

Multiplication. Consider vector $z = \left[\begin{smallmatrix} z_1 \\ z_2 \end{smallmatrix}\right] \in \mathbb{C}^{n+k}$ (remember that z_2 is stored redundantly as discussed above). The matrix-vector product $y = \tilde{P}(\mu)z$ is carried out by means of the calculations,

$$y_1 = P(\mu)z_1 + \sum_{i=0}^{d} A_i X \hat{\Phi}_i(H,\mu) z_2, \tag{32}$$

$$y_2 = \sum_{i=0}^{\ell-1} \Phi_i(\mu)\, \Phi_i(H)^* X^* z_1 + \sum_{i=0}^{\ell-1} \Phi_i(H)^* X^* X \hat{\Phi}_i(H,\mu) z_2, \tag{33}$$

in which, in addition to the matrix-vector multiplication with the original problem, $P(\mu)z_1$, other computations are done that require communication among the processes. In order to minimize parallel communication, we allocate additional distributed data structures (of class `BV`) in which the products $A_i X$, $i = 0, \ldots, d$, are explicitly stored. These variables are updated whenever a new eigenpair is obtained. Similarly, the products X^*X (of dimension $k \times k$) are explicitly stored redundantly in each process. In this way, the second summands appearing in expressions (32) and (33) do not require any communication since z_2 (of length k) is available in all processes. In turn, in the first summand of (33) we must compute the inner product of k pairs of vectors, which is done with a single communication operation thanks to the functionality offered by the `BV` class. The result y_2 is stored redundantly in all processes.

In our implementation, the value of ℓ in (33) does not necessarily correspond to the minimality index of the invariant pair. By default, we take $\ell = \min\{d, k\}$, but it is also possible to limit its value (see section 5.4).

Preconditioner. As mentioned before, the extended preconditioner $\tilde{K}^{-1}$ of (26) is not built explicitly. Instead, at the implementation level, a subroutine is defined that computes its application to a vector, considering that the necessary elements have been precomputed and stored in a data structure.

Regarding the parallel implementation, in order to minimize the extra communication required to apply the extended preconditioner (compared to the one of dimension n), whenever the problem is extended with a new eigenpair we precompute the small-size matrices

$$S := (\mu I - H)^{\dagger}, \quad J := (B(\mu) - A(\mu)XS)^{-1}, \tag{34}$$

and store them redundantly in all processes. In this way, the application of the preconditioner (26) takes the form

$$\tilde{K}^{-1} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} K^{-1}z_1 - XSJz_2 \\ Jz_2 \end{bmatrix}, \tag{35}$$

and no additional communication (compared to the operation $K^{-1}z_1$) is required.

In the case that $\ell = 1$, we have that $A(\mu) = X^*$, $B(\mu) = 0$, so $J = -(X^*XS)^{-1}$ and the preconditioner can be simplified to

$$\tilde{K}^{-1} = \begin{bmatrix} K^{-1} & X(X^*X)^{-1} \\ 0 & (H - \mu I)(X^*X)^{-1} \end{bmatrix}. \tag{36}$$

Orthogonalization. The orthogonalization of the search and test subspaces of extended dimension is carried out by means of the operations available in the `BV` class, and does not imply additional communication with respect to the one of dimension n. The only additional comment here is that, given a vector $y = \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] \in \mathbb{C}^{n+k}$, in our implementation the second part of the vector (which is replicated in all processes) stores $y_2/\sqrt{p}$ instead of y_2, so that all inner products and norms result in the correct value irrespective of the number of processes p. This scaling is applied on the result vectors of the operations described above.

Projection. The projected problem that is solved in the extraction phase has an expression of the form

$$W^* \tilde{P}(\mu) V = \sum_{i=0}^{d} Q_i \, \Phi_i(\mu). \tag{37}$$

The coefficient matrices Q_i associated with the basis $\{\Phi_i\}$ are stored and extended whenever the subspaces V and W are extended. For this, distributed data structures U_i, $i = 0, \ldots, d$, are created to store sets of vectors such that $\tilde{P}(\mu)V$ can be expressed by

$$\tilde{P}(\mu) V = \sum_{i=0}^{d} U_i \, \Phi_i(\mu), \tag{38}$$

for any $\mu \in \mathbb{C}$. In this way, when a column is appended to V and W (in position j), the matrices Q_i in (37) can be extended easily, by just adding a row $e_j^T Q_i = w_j^* U_i$ and a column $Q_i e_j = W^* U_i e_j$. To update the set of vectors $\{U_i\}_{i=0}^{d}$, extending them when the V basis is extended with a column v, we use the expression of the multiplication operation

$$\tilde{P}(\mu) \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} P(\mu) v_1 + U(\mu) v_2 \\ A(\mu) v_1 + B(\mu) v_2 \end{bmatrix} = \sum_{i=0}^{d} \begin{bmatrix} A_i v_1 + N_i v_2 \\ \Phi_i(H)^* X^* v_1 + \hat{N}_i v_2 \end{bmatrix} \Phi_i(\mu), \tag{39}$$

where the matrices N_i and $\hat{N}_i$ are obtained by means of the recurrence (23) using matrices $M_i = A_i X$ and $\hat{M}_i = \Phi_i(H)^* X^* X$, respectively. Each matrix U_i is extended with the corresponding coefficient associated with $\Phi_i(\mu)$ in (39).

5.3 Details for real arithmetic

In the case of using real arithmetic, if the Ritz value μ is complex the operations must be carried out storing the real and imaginary part of each variable separately. For instance, in the first part of the matrix vector product with the extended operator, corresponding to (32), the real part of y_1 is obtained as

$$\begin{aligned} \operatorname{Re} y_1 = & \sum_{i=0}^{d} A_i \left(\operatorname{Re} \Phi_i(\mu) \operatorname{Re} z_1 - \operatorname{Im} \Phi_i(\mu) \operatorname{Im} z_1 \right) \\ & + \sum_{i=0}^{d} A_i X \left(\operatorname{Re} \hat{\Phi}_i(H, \mu) \operatorname{Re} z_2 - \operatorname{Im} \hat{\Phi}_i(H, \mu) \operatorname{Im} z_2 \right), \end{aligned} \tag{40}$$

and we have similar expressions for $\operatorname{Im} y_1$, $\operatorname{Re} y_2$ and $\operatorname{Im} y_2$. The subroutine to compute $\operatorname{Re} \Phi_i(\mu)$ and $\operatorname{Im} \Phi_i(\mu)$ works in a similar way, separating real and imaginary parts in recurrence (4), and similarly for $\hat{\Phi}_i(H, \mu)$ in recurrence (17).

In real arithmetic, the correction equation needs to be expressed in the 2×2 block form (29) whenever the Ritz value μ is complex. Our implementation of the Jacobi–Davidson solver

is able to dynamically switch from a linear system solve of order n to order $2n$, and vice versa, when needed. In order to avoid reallocating all the memory when this change happens, we use a trick consisting in using a standard `KSP` solver defined on vectors of the special type `VecComp`. These vectors virtually have length $2n$, but are actually formed by two sub-vectors of length n, the second one being used only when the approximate eigenvalue is complex. This trick was already used in the Jacobi–Davidson solver for linear eigenproblems [24], and simplifies the implementation of the product of the double-sized coefficient matrix (which is not built explicitly but handled implicitly as a shell matrix) times a vector, that can be arranged as a 2-by-2 block product. Similarly, the application of the preconditioner is done in a 2-by-2 fashion using the expression of (30).

5.4 User interface

We conclude this section with a brief description of the user interface for the new polynomial Jacobi–Davidson solver. Apart from the `PEP` general options (number of eigenvalues to compute, maximum basis size, etc.), the solver has several options and parameters that can be set by the user, either by a function call or by a runtime argument.

- `PEPJDSetRestart`. Sets the restart parameter, that is, the proportion of basis vectors that must be kept after restart. The default is 0.5.
- `PEPJDSetMinimalityIndex`. Sets the maximum allowed value of ℓ used in (33).
- `PEPJDSetFix`. In step 14 of Algorithm 1 the correction equation is built from the constant value σ instead of the Ritz value μ whenever the residual norm is larger than a user-defined parameter called *fix* [9].
- `PEPJDSetReusePreconditioner`. Sets a flag indicating whether the preconditioner must be reused or not. This is off by default. If the flag is set, the preconditioner is built only at the beginning, using the target value σ.
- `PEPJDSetProjection`. Switches between oblique (the default) or orthogonal projection.

6 Computational results

In this section we show computational results to assess the accuracy of the new solver as well as its performance (including parallel efficiency). The computer used for the runs is called Tirant 3, and consists of 336 computing nodes, each with two Intel Xeon SandyBridge E5-2670 processors (16 cores each) running at 2.6 GHz with 32 GB of memory, linked with an Infiniband network. All runs placed at most 8 MPI process per node.

The details for the test problems we have used is summarized in Table 1. The table shows the degree of the polynomial, the dimension of the matrices that define the problem, the number of requested eigenvalues, and the target value around which eigenvalues have been computed. The first considered problems (qd_cylinder and qd_pyramid) arise from electronic structure calculations of quantum dots by means of the discretization of the Schrödinger equation [17]. The rest of the problems belong to the collection of nonlinear eigenvalue problems NLEVP [3]. The last test problem in the table (loaded_string) corresponds to a rational eigenproblem that has been solved by means of polynomial interpolation using Chebyshev polynomials [8], that is why the resulting polynomial eigenproblem is expressed in terms of such polynomial basis. The other considered problems are all defined using the monomial basis.

Table 2 shows the results of several executions using test problems from Table 1. The table compares Jacobi–Davidson (run in both real and complex arithmetic) and the Krylov solver

Table 1: Description of the test problems used for the performance analysis, indicating the degree (d) of the matrix polynomial, the polynomial basis, the dimension (n) of the coefficient matrices, the number of requested eigenvalues (nev) and the target (σ) around which eigenvalues have been computed.

name	d	basis	n	nev	σ
qd_cylinder	3	monomial	690,718	4	0.1
qd_pyramid-186k	5	monomial	186,543	6	0.4
qd_pyramid-1.5m	5	monomial	1.5 mill	4	0.4
sleeper	2	monomial	1 mill	20	-0.9
spring	2	monomial	1 mill	5	-10
acoustic_wave_2d	2	monomial	999,000	2	0
loaded_string	10	Chebyshev	1 mill	6	0

TOAR (run with either a direct linear solver, LU, or a preconditioned iterative solver). For the iterative linear solves, the `KSP` is configured to run BiCGStab with a tolerance of 10^{-5} (and 150 iterations at most) in the case of Jacobi–Davidson, and 10^{-8} in the case of TOAR. In the table, BJacobi and ASM stand for block Jacobi and (restricted) Additive Schwarz Method, respectively, which are preconditioners implemented in PETSc. The other preconditioners are provided by third-party packages: LU is implemented in MUMPS, ARMS in PARMS, and AMG in Hypre. Both Jacobi–Davidson and TOAR had a maximum subspace dimension 30 and a tolerance of 10^{-8}. The accuracy of a computed solution (x, λ) is assessed by means of its relative backward error

$$\eta_P(x,\lambda) = \frac{\|P(\lambda)x\|}{\left(\sum_{i=0}^{d} |\Phi_i(\lambda)| \|A_i\|\right)\|x\|}, \tag{41}$$

where we use ∞-norms for practical computation of matrix norms. We can see from the maximum values of η_P shown in Table 1 that the new Jacobi–Davidson solver works correctly in terms of accuracy.

From the execution times shown in Table 1, we can draw several conclusions:

- One could expect that the real version of Jacobi–Davidson has a different convergence history compared with the complex version. However, in these test problems both versions converge more or less with the same number of iterations, probably because all computed eigenvalues are real except in the acoustic_wave problem that has complex eigenvalues. In terms of computation time, the complex version takes up to twice as much time, due to complex arithmetic.

- The TOAR solver is generally the fastest one, either when a direct linear solver (LU) is used or in other cases with an iterative one. Using a direct solver is feasible in these tests due to their moderate problem size, except in the qd_pyramid problem of size 1,5 million. For this latter problem we see that Jacobi–Davidson is faster than TOAR.

- The difference between the overall solve time and the `KSP` solve time is significantly larger in Jacobi–Davidson, because it is doing more operations compared to TOAR. The splitting of these operations is shown in Table 3 for two problems. The linear solve (`KSP`) takes more than 75% or the time, and the other operations are: updating the preconditioner, which includes computing $P(\theta)$, the orthogonalization of the basis vectors, the computation of

Table 2: Results for several executions with 16 MPI processes of the polynomial eigensolvers Jacobi–Davidson (JD) and TOAR, showing the employed preconditioner (PC), the number of iterations used by the polynomial eigensolver ($I_{\mathtt{PEP}}$) and the linear solver ($I_{\mathtt{KSP}}$), the execution time in seconds of both the polynomial eigensolver ($t_{\mathtt{PEP}}$) and the linear solver ($t_{\mathtt{KSP}}$) and the maximum backward error η_P.

problem	method	PC	$I_{\mathtt{KSP}}$	$t_{\mathtt{KSP}}$	$I_{\mathtt{PEP}}$	$t_{\mathtt{PEP}}$	η_P
qd_cylinder	JD real	ASM	4367	21	42	25	$1 \cdot 10^{-13}$
	JD complex	ASM	4412	42	43	47	$5 \cdot 10^{-13}$
	TOAR	ASM	17375	60	100	61	$2 \cdot 10^{-10}$
	TOAR	LU	100	14	100	21	$2 \cdot 10^{-10}$
qd_pyramid-186k	JD real	ASM	1466	3.5	37	5.6	$6 \cdot 10^{-11}$
	JD complex	ASM	1447	7.1	36	10	$5 \cdot 10^{-11}$
	TOAR	ASM	2503	3.2	58	3.4	$8 \cdot 10^{-11}$
	TOAR	LU	57	2.9	57	14	$1 \cdot 10^{-12}$
qd_pyramid-1.5M	JD real	ASM	1494	23	20	30	$4 \cdot 10^{-11}$
	JD complex	ASM	1591	50	22	60	$7 \cdot 10^{-11}$
	TOAR	ASM	3417	38	45	40	$5 \cdot 10^{-11}$
	TOAR	LU	45	23	45	760	$2 \cdot 10^{-14}$
sleeper	JD real	BJacobi	5916	53	95	68	$1 \cdot 10^{-10}$
	JD complex	BJacobi	5256	98	89	130	$2 \cdot 10^{-10}$
	TOAR	BJacobi	-	-	-	-	-
	TOAR	LU	83	17	83	21	$2 \cdot 10^{-13}$
spring	JD real	BJacobi	699	3.4	20	5.0	$1 \cdot 10^{-10}$
	JD complex	BJacobi	533	5.6	19	7.8	$3 \cdot 10^{-11}$
	TOAR	BJacobi	1131	3.6	30	3.9	$4 \cdot 10^{-12}$
	TOAR	LU	30	5.9	30	67	$1 \cdot 10^{-13}$
acoustic_wave	JD real	ARMS	3868	331	30	393	$2 \cdot 10^{-9}$
	JD complex	ARMS	3520	300	21	361	$1 \cdot 10^{-9}$
	TOAR	ARMS	7602	356	30	358	$5 \cdot 10^{-10}$
	TOAR	LU	30	6.9	30	15	$5 \cdot 10^{-10}$
loaded_string	JD real	AMG	117	2.9	20	8.8	$2 \cdot 10^{-16}$
	TOAR	AMG	60	1.3	30	2.4	$1 \cdot 10^{-11}$
	TOAR	LU	30	59	30	68	$4 \cdot 10^{-13}$

Table 3: Split execution times for Jacobi–Davidson (in real arithmetic) when solving qd_cylinder and qd_pyramid-1.5M with 16 MPI processes.

operation	qd_cylinder	qd_pyramid-1.5M
Linear solve	21.2	22.9
Preconditioner update	1.75	4.23
Orthogonalization in JD	0.36	0.34
Projection	0.45	0.52
Projected PEP	0.14	0.29
PEP solve	24.6	29.5

the projection (37), and the solution of the projected PEP. It should be noted that the latter operation is not parallel, all processes compute the solution redundantly.

For the parallel performance experiments we use the quantum dot problems (with monomial basis) and the loaded_string problem (with Chebyshev basis). Figure 1 shows the execution time with varying number of MPI processes for qd_cylinder and qd_pyramid-1.5m. In the plot for qd_cylinder, we observe that Jacobi–Davidson is faster than TOAR. We also see that scalability of Jacobi–Davidson is very similar to that of TOAR, with no further improvement after 128 processes (this can also be observed if we consider only the linear solves performed within Jacobi-Davidson, KSP). For the larger problem qd_pyramid-1.5m, performance does not degrade and we observe a very similar behaviour of Jacobi–Davidson and TOAR (with the same preconditioner), while TOAR with LU lags behind due to a exceedingly high cost of the parallel factorization.

Figure 2 shows results for a problem with non-monomial basis, loaded_string. In this case, TOAR with iterative linear solver is faster than Jacobi–Davidson, and we also observe a degradation of parallel performance in both solvers (especially in TOAR). In this case, TOAR with direct linear solves displays a terrible scalability. This Figure also includes a plot corresponding to Jacobi-Davidson with minimality index prescribed to 1, since we know that all wanted eigenvectors are linearly independent in this problem. This allows saving some operations when working with extended operators, which translates into slightly faster times.

7 Conclusions

In this paper we have presented a parallel solver for the polynomial eigenvalue problem based on the Jacobi–Davidson method. A remarkable feature of this solver is its free availability, since it is integrated in the SLEPc package. Other features that are not usually found in other Jacobi–Davidson solvers are its support for non-monomial polynomial bases, the possibility to solve problems with real coefficient matrices using only real arithmetic, and deflation of previously computed eigenvalues.

Our numerical experiments do not show a clear benefit of Jacobi–Davidson with respect to the Krylov solver (TOAR). When many eigenvalues need to be computed, Krylov methods are likely to be faster, because they approximate several eigenvalues at a time while Jacobi–Davidson computes one eigenvalue after the other in sequence. However, Krylov methods have the drawback of often needing a direct linear solver, which usually implies an exceedingly high computational cost and a bad parallel scalability, as seen in some of our tests. Jacobi–Davidson relies on a preconditioned iterative solver, and then the gist becomes how to choose a good preconditioner for the particular problem. In our experiments, we have used standard black-

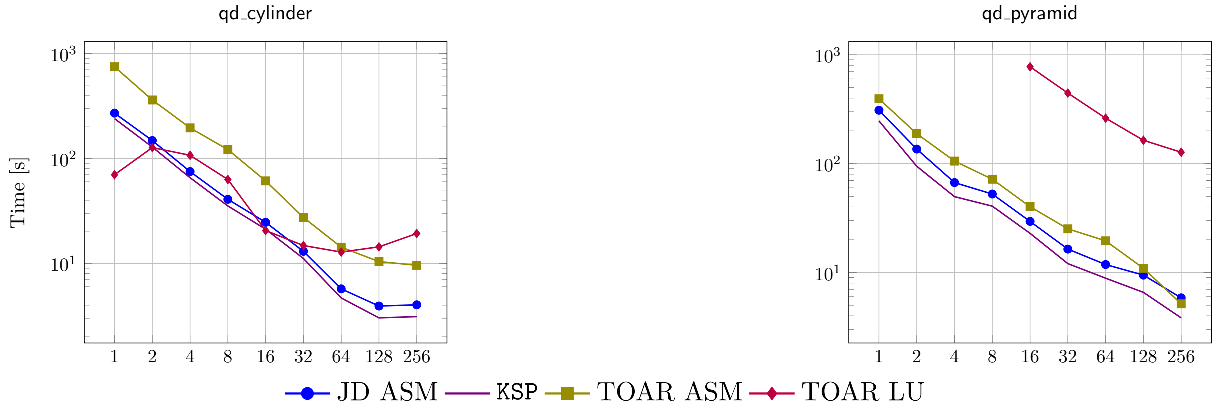

Figure 1: Execution time (in seconds) with the tests qd_cylinder (left) and qd_pyramid (right) for the Jacobi–Davidson solver (with real arithmetic and ASM preconditioner), for the (accumulated) linear solves in the Jacobi–Davidson run (KSP) and for the TOAR solver (both with ASM preconditioner and with direct solver), with up to 256 MPI processes and requesting 4 eigenvalues.

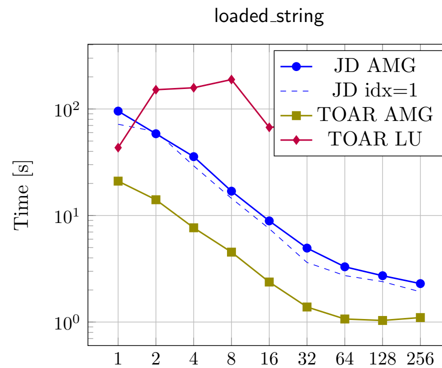

Figure 2: Execution time (in seconds) with the loaded_string test for the Jacobi–Davidson solver (with real arithmetic and AMG preconditioner), for the same solver prescribing the minimality index to 1, and for the TOAR solver (both with AMG preconditioner and with direct solver), with up to 256 MPI processes and requesting 6 eigenvalues.

box preconditioners that may not work especially well, but we would expect a much better performance with problem-specific preconditioners.

In terms of parallel efficiency, our solver has shown a reasonably good behaviour, with a similar scalability to the Krylov solver for the same problem and the same preconditioner.

Regarding plans for future work, it would be interesting to analyze the feasibility of adapting the harmonic Ritz strategies proposed in [15] to the general non-monomial framework proposed here.

Acknowledgements We thank Eloy Romero for useful comments on an initial version of the manuscript. The computational experiments of section 6 were carried out on the supercomputer Tirant 3 belonging to Universitat de València. The authors of [17] are acknowledged for kindly providing the code of their polynomial Jacobi–Davidson solver, which served as inspiration for building our own solver, as well as the matrices coming from the quantum dot simulation used in section 6.

References

[1] Z. Bai and Y. Su. SOAR: a second-order Arnoldi method for the solution of the quadratic eigenvalue problem. *SIAM J. Matrix Anal. Appl.*, 26(3):640–659, 2005.

[2] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. Curfman McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.10, Argonne National Laboratory, 2018.

[3] T. Betcke, N. J. Higham, V. Mehrmann, C. Schröder, and F. Tisseur. NLEVP: a collection of nonlinear eigenvalue problems. *ACM Trans. Math. Software*, 39(2):7:1–7:28, 2013.

[4] T. Betcke and D. Kressner. Perturbation, extraction and refinement of invariant pairs for matrix polynomials. *Linear Algebra Appl.*, 435(3):514–536, 2011.

[5] T. Betcke and H. Voss. A Jacobi–Davidson-type projection method for nonlinear eigenvalue problems. *Future Gener. Comp. Sy.*, 20(3):363–372, 2004.

[6] C. Campos and J. E. Roman. Parallel Krylov solvers for the polynomial eigenvalue problem in SLEPc. *SIAM J. Sci. Comput.*, 38(5):S385–S411, 2016.

[7] C. Effenberger. Robust successive computation of eigenpairs for nonlinear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 34(3):1231–1256, 2013.

[8] C. Effenberger and D. Kressner. Chebyshev interpolation for nonlinear eigenvalue problems. *BIT*, 52(4):933–951, 2012.

[9] Diederik R. Fokkema, Gerard L. G. Sleijpen, and Henk A. van der Vorst. Jacobi–Davidson style QR and QZ algorithms for the reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1):94–125, 1998.

[10] J.-S. Guo, W.-W. Lin, and C.-S. Wang. Numerical solutions for large sparse quadratic eigenvalue problems. *Linear Algebra Appl.*, 225:57–89, 1995.

[11] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005.

[12] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010.

[13] N. J. Higham, D. S. Mackey, and F. Tisseur. The conditioning of linearizations of matrix polynomials. *SIAM J. Matrix Anal. Appl.*, 28(4):1005–1028, 2006.

[14] M. Hochbruck and D. Lochel. A multilevel Jacobi–Davidson method for polynomial PDE eigenvalue problems arising in plasma physics. *SIAM J. Sci. Comput.*, 32(6):3151–3169, 2010.

[15] Michiel E. Hochstenbach and Gerard L. G. Sleijpen. Harmonic and refined Rayleigh-Ritz for the polynomial eigenvalue problem. *Numer. Linear Algebra Appl.*, 15(1):35–54, 2008.

[16] T.-M. Huang, F.-N. Hwang, S.-H. Lai, W. Wang, and Z.-H. Wei. A parallel polynomial Jacobi–Davidson approach for dissipative acoustic eigenvalue problems. *Computers & Fluids*, 45(1):207–214, 2011.

[17] F.-N. Hwang, Z.-H. Wei, T.-M. Huang, and W. Wang. A parallel additive Schwarz preconditioned Jacobi–Davidson algorithm for polynomial eigenvalue problems in quantum dot simulation. *J. Comput. Phys.*, 229(8):2932–2947, 2010.

[18] D. Kressner. A block Newton method for nonlinear eigenvalue problems. *Numer. Math.*, 114:355–372, 2009.

[19] D. Kressner and J. E. Roman. Memory-efficient Arnoldi algorithms for linearizations of matrix polynomials in Chebyshev basis. *Numer. Linear Algebra Appl.*, 21(4):569–588, 2014.

[20] P. Lancaster. Linearization of regular matrix polynomials. *Electron. J. Linear Algebra*, 17:21–27, 2008.

[21] Y. Matsuo, H. Guo, and P. Arbenz. Experiments on a parallel nonlinear Jacobi–Davidson algorithm. *Procedia Comp. Sci.*, 29:565–575, 2014.

[22] K. Meerbergen. Locking and restarting quadratic eigenvalue solvers. *SIAM J. Sci. Comput.*, 22(5):1814–1839, 2001.

[23] J. E. Roman, C. Campos, E. Romero, and A. Tomas. SLEPc users manual. Technical Report DSIC-II/24/02–Revision 3.10, D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2018.

[24] E. Romero and J. E. Roman. A parallel implementation of Davidson methods for large-scale eigenvalue problems in SLEPc. *ACM Trans. Math. Software*, 40(2):13:1–13:29, 2014.

[25] J. Rommes and N. Martins. Computing transfer function dominant poles of large-scale second-order dynamical systems. *SIAM J. Sci. Comput.*, 30(4):2137–2157, 2008.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM Publications, 2nd edition, 2003.

[27] C. Sensiau, F. Nicoud, M. van Gijzen, and J. W. van Leeuwen. A comparison of solvers for quadratic eigenvalue problems from combustion. *Int. J. Numer. Methods Fluids*, 56(8):1481–1488, 2008.

[28] Gerard L. G. Sleijpen, Albert G. L. Booten, Diederik R. Fokkema, and Henk A. van der Vorst. Jacobi-Davidson type methods for generalized eigenproblems and polynomial eigenproblems. *BIT*, 36(3):595–633, 1996.

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

[30] Gerard L. G. Sleijpen, Henk A. van der Vorst, and Ellen Meijerink. Efficient expansion of subspaces in the Jacobi–Davidson method for standard and generalized eigenproblems. *Electron. Trans. Numer. Anal.*, 7:75–89, 1998.

[31] F. Tisseur and K. Meerbergen. The quadratic eigenvalue problem. *SIAM Rev.*, 43(2):235–286, 2001.

[32] M. B. van Gijzen and F. Raeven. The parallel computation of the smallest eigenpair of an acoustic problem with damping. *Int. J. Numer. Methods Eng.*, 45(6):765–777, 1999.

[33] T. van Noorden and J. Rommes. Computing a partial generalized real Schur form using the Jacobi–Davidson method. *Numer. Linear Algebra Appl.*, 14(3):197–215, 2007.

[34] H. Voss. A Jacobi–Davidson method for nonlinear and nonsymmetric eigenproblems. *Comput. & Structures*, 85(17-18):1284–1292, 2007.