PARALLEL KRYLOV SOLVERS FOR THE POLYNOMIAL EIGENVALUE PROBLEM IN SLEPc*

CARMEN CAMPOS† AND JOSE E. ROMAN†

Abstract. Polynomial eigenvalue problems are often found in scientific computing applications. When the coefficient matrices of the polynomial are large and sparse, usually only a few eigenpairs are required and projection methods are the best choice. We focus on Krylov methods that operate on the companion linearization of the polynomial, but exploit the block structure with the aim of being memory-efficient in the representation of the Krylov subspace basis. The problem may appear in the form of a low-degree polynomial (quartic or quintic, say) expressed in the monomial basis, or a high-degree polynomial (coming from interpolation of a nonlinear eigenproblem) expressed in a non-monomial basis. We have implemented a parallel solver in SLEPc covering both cases, that is able to compute exterior as well as interior eigenvalues via spectral transformation. We discuss important issues such as scaling and restart, and illustrate the robustness and performance of the solver with some numerical experiments.

Key words. matrix polynomial, eigenvalues, companion linearization, Krylov subspace, non-monomial bases, spectral transformation, parallel computing, SLEPc

AMS subject classifications. 65F15, 65F50, 15A18, 41A05

1. Introduction.

Consider the matrix polynomial

$$P(\lambda) = A_0 + \lambda A_1 + \lambda^2 A_2 + \cdots + \lambda^d A_d, \tag{1.1}$$

where A_i are $n \times n$ matrices (real or complex), and assume it is regular, that is, $\det P(\lambda)$ is not identically zero. This paper is concerned with the polynomial eigenvalue problem, where the goal is to compute eigenpairs (x, λ) satisfying

$$P(\lambda)x = 0, \qquad x \neq 0, \tag{1.2}$$

where $\lambda \in \mathbb{C}$ is the eigenvalue and $x \in \mathbb{C}^n$ is the (right) eigenvector. There is an increasing demand for efficiently solving this kind of problems in scientific computing applications. Some examples can be found in the NLEVP collection [8]. We are interested in the case of large-scale computations where the coefficient matrices A_i are large and sparse, arising typically from the discretization of partial differential equations. Second-order models are the most commonly found, leading to quadratic eigenvalue problems [29], but higher order ones can also appear in practical applications, resulting in polynomials of modest degree, $d \leq 6$, say. Also, matrix polynomials of arbitrary degree are obtained when solving nonlinear eigenvalue problems via polynomial interpolation, see, e.g., [11]. This latter case requires expressing the polynomial in a non-monomial basis, as opposed to the representation in (1.1).

Projection methods for large-scale polynomial eigenvalue problems may opt to project the problem directly by imposing a Galerkin-type condition on the residual associated with the polynomial eigenproblem, see, e.g., [18, 19, 5]. An alternative approach is to apply a standard projection method such as Arnoldi to the linearization of the matrix polynomial. The main drawback of this latter approach is that the linear

*This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2013-41049-P. Carmen Campos was supported by the Spanish Ministry of Education, Culture and Sport through an FPU grant with reference AP2012-0608.

†D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 Valencia, Spain (mccampos@dsic.upv.es, jroman@dsic.upv.es).

eigenproblem to be solved is much larger, increased by a factor of d, although it is possible to exploit the block structure of the linearization resulting in memory-efficient variants such as Q-Arnoldi [24] that do not need to explicitly store basis vectors of length dn.

In this paper, we take the linearization route and consider Krylov-type iterations, in particular memory-efficient variants of Krylov-Schur [26]. We present implementations in the context of SLEPc, the Scalable Library for Eigenvalue Problem Computations [16]. The structure of the polynomial eigensolver in SLEPc is very similar to the linear eigensolver, with a lot of flexibility for the user to set various parameters and options, even at run time. For instance, the user can choose the method to use, set the convergence tolerance, the dimension of the Krylov subspace, or the restart parameter. For computing interior eigenvalues, our implementation provides the shift-and-invert spectral transformation, which has been tailored specifically to the structure of the linearized problem, either in the monomial or non-monomial basis [21].

Compared to the linear case, implementing a robust and efficient polynomial eigensolver is much more difficult, since issues such as conditioning may have more importance. We provide details concerning various relevant aspects such as how to lock converged Ritz vectors, or how to obtain eigenpairs of (1.1) from the eigenpairs of the linearization. This latter issue is discussed by Betcke and Kressner in [9], and we make the necessary adaptations for our specific algorithms. Furthermore, scaling of the matrix polynomial coefficients turns out to be crucial for improving the conditioning of the linearized problem, and several strategies have been proposed to this end, e.g., [7]. Our solvers combine all these ingredients in a flexible way, enabling the solution of a wide variety of polynomial eigenproblems. Moreover, all the computations are carried out in parallel and hence very large scale problems can be addressed.

Our software implementation is unique. No other publicly available software provides Krylov solvers for the polynomial eigenproblem. Only some Matlab implementations for the complete solution of quadratic eigenproblems can be found [14].

The rest of the paper is organized as follows. Section 2 provides background information on basic tools such as linearization, memory-efficient Krylov solvers, spectral transformation, and polynomial bases. Also, a short description of SLEPc is given. Section 3 proceeds to detail the Krylov methods that we have implemented, namely plain Arnoldi, Q-Arnoldi and TOAR (Two-level Orthogonal Arnoldi). The description includes topics such as restart and locking, convergence criteria, and scaling. In §4 we discuss the user interface for SLEPc's polynomial eigensolvers and provide some usage examples. Section 5 deals with numerical results for various test cases, including the evaluation of parallel performance. We wrap up with some conclusions in §6.

2. Preliminaries. In the methods that approximate the solution of a polynomial eigenproblem of dimension n and degree d via linearization, the involved vectors have length dn. We will consider that vectors $v \in \mathbb{C}^{dn}$ and tall-skinny matrices $V \in \mathbb{C}^{dn\times k}$ are divided in d blocks of n rows,

$$v = \begin{bmatrix} v^0 \\ \vdots \\ v^{d-1} \end{bmatrix}, \qquad V = \begin{bmatrix} V^0 \\ \vdots \\ V^{d-1} \end{bmatrix}, \tag{2.1}$$

where $v^i \in \mathbb{C}^n$ and $V^i \in \mathbb{C}^{n\times k}$ for $i = 0, \ldots, d-1$. Throughout the text, we will use superindices to denote each of the blocks of the split form (2.1).

The identity matrix of order m will be denoted by I_m. The ith canonical vector, that is, the ith column of the identity matrix, will be denoted by e_i, and its length will be assumed to match the dimension of the expression where it appears.

The reverse of a matrix polynomial $P(\lambda)$ of degree d is defined as the matrix polynomial given by $\operatorname{rev} P(\lambda) := \lambda^d P(\frac{1}{\lambda})$.

2.1. Linearization. A usual way to solve the polynomial eigenvalue problem (1.1) is to build and solve an equivalent generalized eigenproblem, $(A - \lambda B)z = 0$, having the same spectrum as the original problem. In general, a pencil $L(\lambda) = A - \lambda B$ is said to be a linearization of $P(\lambda)$ if there exist unimodular[1] matrix polynomials, $U(\lambda)$ and $V(\lambda)$, such that

$$U(\lambda)L(\lambda)V(\lambda) = \begin{bmatrix} P(\lambda) & 0 \\ 0 & I_{(d-1)n} \end{bmatrix}. \tag{2.2}$$

Unimodular transformations preserve the finite spectral structure (finite elementary divisors). Strong linearizations, in which $L(\lambda)$ and $\operatorname{rev} L(\lambda)$ are linearizations of $P(\lambda)$ and $\operatorname{rev} P(\lambda)$, respectively, preserve the spectral structure also for the ∞ eigenvalue.

For a matrix polynomial $P(\lambda)$ defining a polynomial eigenproblem, there exist many possible linearizations sharing the same spectrum. Those linearizations may have different structural properties and conditioning. From a computational point of view, in order to solve a polynomial eigenproblem, it would be preferable to choose a linearization having the best possible conditioning, and with structural properties that preserve the spectral properties of the original problem in the computed result. For example, this is the case when using a symmetric linearization when the matrices of the original polynomial are symmetric. For a detailed information about linearizations and conditioning see [13, 23, 17].

Once the polynomial eigenproblem has been reduced to a linear one via linearization, it can be solved by any method for generalized eigenproblems. The drawback of using this approach is that the new problem to be solved has a dimension of d times the original dimension, being d the degree of P. Hence, it is important to somehow take advantage of the block structure of the linearization used, in order to reduce the cost (in memory or computation) of solving the linear problem.

In this work, to linearize the polynomial eigenproblem (1.1), we use the first companion form given by

$$L(\lambda) = A - \lambda B, \quad \text{where} \tag{2.3a}$$

$$A = \begin{bmatrix} 0 & I & 0 & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & I \\ -A_0 & -A_1 & -A_2 & \cdots & -A_{d-1} \end{bmatrix}, \quad B = \begin{bmatrix} I & & & & \\ & I & & & \\ & & \ddots & & \\ & & & I & \\ & & & & A_d \end{bmatrix}. \tag{2.3b}$$

The first companion form is a strong linearization of (1.1), implying that P and L share the same spectral structure for finite and infinite eigenvalues. Also, the eigenvectors can be easily recovered from those of (2.3), taking into account that each right eigenvector, z, of (2.3) has the form $z = \begin{bmatrix} 1 & \lambda & \dots & \lambda^{d-1} \end{bmatrix}^T \otimes x$, being x a right eigenvector of (1.1).

[1] A matrix polynomial $U(\lambda)$ is unimodular if $\det(U(\lambda)) \equiv c$ for some constant c.

Another potential advantage of the first companion form is its block structure containing many zero blocks, which can be exploited to save memory when computing a few eigenpairs, as in the methods of Q-Arnoldi or TOAR, see §2.2. Other linearizations also share this property.

2.2. Krylov methods for linearized polynomial eigenproblems. After linearizing the matrix polynomial, $P(\lambda)$, the generalized eigenproblem produced is reduced to standard form,

$$Sz = \lambda z, \tag{2.4}$$

where $S = B^{-1}A$, for A and B as in (2.3), and A_d is assumed to be non-singular. Then, from an initial vector $u \neq 0$, the well-known Arnoldi method [4] can be used to compute an orthonormal basis $\{v_1, \ldots, v_{k+1}\}$ of the Krylov subspace $\mathcal{K}_{k+1} := \text{span}\{u, Su, \ldots, S^k u\}$, and a (projected) Hessenberg matrix, H_k, satisfying Arnoldi's relation,

$$SV_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^*, \tag{2.5}$$

where $V_k = \begin{bmatrix} v_1 & \cdots & v_k \end{bmatrix}$.

From (2.5), eigenpairs of H_k, (y, μ), are used to compute Ritz values, μ, and Ritz vectors, $\hat{z} = V_k y$, which yield approximate eigenpairs of (2.4). The norm of the residual, $r(\mu, \hat{z}) := S\hat{z} - \mu\hat{z}$, provides a bound of the (absolute) backward error for these approximate eigenpairs, and the expression

$$\rho(\hat{z}, \mu) := \|S\hat{z} - \mu\hat{z}\| = h_{k+1,k} |e_k^* y|, \tag{2.6}$$

can be used to determine which of them are sufficiently converged.

For a matrix polynomial of degree d, the linear eigenproblem solved is d times larger than the original problem (of dimension n). Hence, from a computational point of view, it is important to avoid the explicit creation of matrices (2.3) (of dimension nd). For expanding the Krylov subspace when performing Arnoldi's iteration, $w = Sv = B^{-1}Av$, it is possible to carry out the multiplication with matrix S, operating directly with the polynomial coefficients, A_i. For this, considering that each vector of $\mathbb{C}^{nd}$ is split in the form (2.1), the special structure of A and B allows the computation of each block of w with

$$\begin{cases} w^i = v^{i+1}, \quad i = 0, \ldots, d-2, \\ w^{d-1} = -A_d^{-1}(A_0 v^0 + \cdots + A_{d-1} v^{d-1}). \end{cases} \tag{2.7}$$

This strategy avoids storing the complete matrices (2.3) in full form, but Arnoldi vectors (of dimension nd) are still computed and stored explicitly.

The Q-Arnoldi method [24] is a memory-efficient variant of Arnoldi, that takes advantage of the structure of the linearization (2.3) and works storing only the first n entries of each computed Arnoldi vector. Considering vectors split in the form (2.1), equating the first $d-1$ block rows of (2.5) yields relations between blocks V_k^i,

$$V_k^{i+1} = V_k^i H_k + h_{k+1,k} v_{k+1}^i e_k^*, \quad i = 0, \ldots, d-2, \tag{2.8}$$

that can be used to carry out Arnoldi's iteration, storing only V_k^0 and the full continuation vector v_{k+1}.

In order to illustrate the method, we now show the operations involved in one iteration of the Q-Arnoldi method for the case of degree 2. Starting from an Arnoldi relation (2.5) of order k in which only the first block of V_k has been explicitly stored, a relation of order $k+1$ can be obtained with the following steps:

1. The Krylov subspace is expanded with $w = Sv_{k+1}$, operating with the matrices of the polynomial problem as in (2.7). The stored block is updated as $V_{k+1}^0 \leftarrow [\, V_k^0 \; v_{k+1}^0 \,]$.
2. Vector w is orthogonalized against the columns of V_{k+1}, using (2.8),

$$\tilde{w} := w - \begin{bmatrix} V_{k+1}^0 \\ [V_k^1 \quad v_{k+1}^1] \end{bmatrix} V_{k+1}^* w = \tag{2.9}$$
$$= \begin{bmatrix} w^0 \\ w^1 \end{bmatrix} - \begin{bmatrix} V_{k+1}^0 \\ [V_k^0 H_k + h_{k+1,k} v_{k+1}^0 e_k^* \quad v_{k+1}^1] \end{bmatrix} h_{k+1},$$

   where the h_{k+1} coefficients corresponding to the $(k+1)$th column of H_{k+1} are

$$h_{k+1} := \begin{bmatrix} V_{k+1}^0 \\ [V_k^1 \quad v_{k+1}^1] \end{bmatrix}^* \begin{bmatrix} w^0 \\ w^1 \end{bmatrix} = \tag{2.10}$$
$$= (V_{k+1}^0)^* w^0 + \begin{bmatrix} H_k^* (V_k^0)^* w^1 + h_{k+1,k} (v_{k+1}^0)^* w^1 e_k \\ (v_{k+1}^1)^* w^1 \end{bmatrix}.$$

3. The normalization of the vector $\tilde{w}$ results in the Arnoldi vector generated in this iteration, $v_{k+2} := \tilde{w}/h_{k+2,k+1}$, being $h_{k+2,k+1} := \|\tilde{w}\|$. Vector v_{k+1}^1 is no longer used so it can be discarded.

In analogy to Q-Arnoldi, the TOAR method [27, 22] keeps the idea of recovering each block, V_k^i, when needed, from a set of vectors in $\mathbb{C}^n$. In the case of Q-Arnoldi, these vectors are given by $\{V_k^0 e_j\}_{j=1}^k \cup \{v_{k+1}^0, \ldots, v_{k+1}^{d-1}\}$, whereas TOAR takes these vectors to be an orthonormal basis of the subspace

$$\text{span}(\cup_{i=0}^{d-1} \{V_k^i e_j\}_{j=1}^k \cup \{v_{k+1}^i\}_{i=0}^{d-1}) = \text{span}(\{V_k^0 e_j\}_{j=1}^k \cup \{v_{k+1}^i\}_{i=0}^{d-1}), \tag{2.11}$$

where equality comes from relations (2.8) and hence $k+d$ vectors are needed at most.

Hence, the TOAR method generates a matrix with orthonormal columns, $U_{k+d} \in \mathbb{C}^{n\times(k+d)}$, matrices $\{G_k^i\}_{i=0}^{d-1} \subset \mathbb{C}^{(k+d)\times k}$ and vectors $\{g_{k+1}^i\}_{i=0}^{d-1} \subset \mathbb{C}^{k+d}$ from which to recover Arnoldi vectors (columns of V_k),

$$V_k^i = U_{k+d} G_k^i, \quad i = 0, \ldots, d-1, \tag{2.12a}$$

$$v_{k+1}^i = U_{k+d} g_{k+1}^i, \quad i = 0, \ldots, d-1. \tag{2.12b}$$

Continuing with the particular case of degree 2, now suppose that an Arnoldi relation of order k (2.5) has been computed, and blocks V_k^0 and V_k^1 as well as the continuation vector v_{k+1} are expressed with the TOAR format as in (2.12) (taking $d = 2$). Then, the TOAR steps that will lead to a relation of order $k+1$ will be:

1. For the expansion of the Krylov subspace, the TOAR method adds a new column to matrix U_{k+2} and generates coefficients $g^i \in \mathbb{C}^{k+3}$ in such a way that the two fragments of $w = Sv_{k+1}$ can be expressed as

$$w^i = U_{k+3} g^i, \quad i = 0, 1, \tag{2.13}$$

   where $U_{k+3} := \begin{bmatrix} U_{k+2} & u_{k+3} \end{bmatrix}$ is the extended matrix. For this, vector $w^1 = -A_2^{-1}(A_0 v_{k+1}^0 + A_1 v_{k+1}^1)$ is obtained from (2.7) (reconstructing v_{k+1}^0 and v_{k+1}^1 first) and then orthogonalized against the columns of U_{k+2}. After normalization, it results in u_{k+3}, being $g^1 \in \mathbb{C}^{k+3}$ the orthogonalization coefficients plus the normalization factor, so that (2.13) is satisfied for $i = 1$. Also, from (2.7) we have that $w^0 = v_{k+1}^1 = U_{k+2} g_{k+1}^1$, so defining $g^0 := \begin{bmatrix} g_{k+1}^1 \\ 0 \end{bmatrix}$ the relation (2.13) is satisfied for $i = 0$.

2. To orthogonalize the new vector w against the columns of V_{k+1}, we must compute the orthogonalization coefficients

$$h_{k+1} := \begin{bmatrix} V_{k+1}^0 \\ V_{k+1}^1 \end{bmatrix}^* \begin{bmatrix} w^0 \\ w^1 \end{bmatrix} = \tag{2.14}$$
$$= \begin{bmatrix} G_{k+1}^0 \\ 0 \end{bmatrix}^* U_{k+3}^* U_{k+3} g^0 + \begin{bmatrix} G_{k+1}^1 \\ 0 \end{bmatrix}^* U_{k+3}^* U_{k+3} g^1 =$$
$$= \begin{bmatrix} \tilde{G}_{k+1}^0 \\ \tilde{G}_{k+1}^1 \end{bmatrix}^* \begin{bmatrix} g^0 \\ g^1 \end{bmatrix}, \quad \text{with } \tilde{G}_{k+1}^i := \begin{bmatrix} G_{k+1}^i \\ 0 \end{bmatrix},$$

and then the orthogonalized vector is

$$\tilde{w} := w - V_{k+1} V_{k+1}^* w = \tag{2.15}$$
$$= \begin{bmatrix} U_{k+3} g^0 \\ U_{k+3} g^1 \end{bmatrix} - \begin{bmatrix} U_{k+3} \tilde{G}_{k+1}^0 \\ U_{k+3} \tilde{G}_{k+1}^1 \end{bmatrix} h_{k+1}$$
$$= \begin{bmatrix} U_{k+3} & \\ & U_{k+3} \end{bmatrix} \left(\begin{bmatrix} g^0 \\ g^1 \end{bmatrix} - \begin{bmatrix} \tilde{G}_{k+1}^0 \\ \tilde{G}_{k+1}^1 \end{bmatrix} h_{k+1} \right).$$

In (2.14) and (2.15) we observe that, since the blocks V_k^i are expressed in the form (2.12), the orthogonalization can be formulated in terms of the columns of the G_k^i matrices. That is, each TOAR iteration requires the orthogonalization of a vector of length n (when computing u_{k+3}) and another one of length $2(k+3)$ (when orthogonalizing g against the columns of $\tilde{G}_{k+1}$), instead of one orthogonalization of length dn that would be required in the plain Arnoldi method.

3. The normalization step can also save operations with respect to plain Arnoldi. Define $\tilde{g} := g - \tilde{G}_{k+1} h_{k+1}$, then it can be shown that $\|\tilde{w}\| = \|\tilde{g}\|$, and since $\tilde{w}$ is also expressed in the (2.13) form it is sufficient to normalize $\tilde{g}$ to obtain $g_{k+2} := \tilde{g}/h_{k+2,k+1}$, with $h_{k+2,k+1} := \|\tilde{g}\|$, and the new Arnoldi vector expressed in the form $v_{k+2}^i = U_{k+3} g_{k+2}^i$, for $i = 0, 1$.

Both Q-Arnoldi and TOAR were developed for quadratic eigenproblems and can be generalized to solve polynomial eigenproblems of arbitrary degree [21]. In this work, we present implementations of both Q-Arnoldi and TOAR, but only our TOAR solver has been extended to degrees larger than 2, because this method has a simpler expression for blocks V_k^i when the polynomial eigenproblem is expressed in a non-monomial basis. Also, TOAR has better stability properties because it represents the matrix of Arnoldi vectors, V_k, as a product of 2 matrices with orthonormal columns, and it recovers each block of V_k from a matrix whose columns form an orthonormal basis. In §3.2 we provide details of our implementation of the TOAR method.

2.3. Spectral transformation. As the number of steps of the Arnoldi method increases, the computed Ritz values converge to exterior eigenvalues. When interior eigenvalues are wanted, a spectral transformation can be used, that maps desired eigenvalues to the exterior ones. For instance, the shift-and-invert transformation for linear eigenproblems [4] performs a trivial mapping of eigenvalues while eigenvectors are preserved. We next show how to apply this to the polynomial eigenproblem.

In order to use the shift-and-invert transformation on a polynomial problem, the change of variable can be done either on the parameter λ of the linearized problem or the original polynomial eigenproblem. In the first case, the mapping $\lambda = \frac{1}{\theta} + \sigma$

transforms $L(\lambda)x = 0$ as defined in (2.3) to

$$\tilde{L}(\theta)x = (B - \theta(A - \sigma B))x = 0. \tag{2.16}$$

Dominant eigenvalues θ of the pencil $\tilde{L}(\theta)$ correspond to eigenvalues of $L(\lambda)$ closest to σ. In case of performing the change of variable on the polynomial problem $P(\lambda)x = 0$ defined in (1.1), we first shift the spectrum as $\lambda = \theta + \sigma$, obtaining a new polynomial

$$\begin{aligned}\tilde{P}(\theta) := P(\theta+\sigma) &= A_0 + (\theta+\sigma)A_1 + \cdots + (\theta+\sigma)^d A_d = \\ &= A_0 + (\theta+\sigma)A_1 + \cdots + \left(\sum_{j=0}^{d}\binom{d}{j}\theta^j\sigma^{d-j}\right)A_d,\end{aligned}$$

from which, after reordering and grouping the terms for different powers of θ, we obtain the form $\tilde{P}(\theta) = T_0 + \theta T_1 + \cdots + \theta^d T_d$ with

$$T_k = \sum_{j=k}^{d}\binom{j}{k}\sigma^{j-k}A_j = \sum_{j=0}^{d-k}\binom{j+k}{k}\sigma^j A_{j+k}. \tag{2.17}$$

The last step to obtain the shift-and-invert transformation is to operate with rev $\tilde{P}(\theta)$.

We have implemented both options (transforming the polynomial or the linearization), and the user can choose between them at run time as will be described in §4.

2.4. Polynomial bases. Even though the matrix polynomial defining a polynomial eigenproblem is most often expressed in the monomial basis, in some problems it is more convenient to define it using other polynomial bases, for instance when solving nonlinear eigenproblems via interpolation with a large degree polynomial [21].

With the aim of giving support to a large variety of applications, in our work we consider that matrix polynomials can be expressed in terms of a more general set of polynomials, $\{\phi_j\}_{j=0}^{d}$,

$$P(\lambda) = A_0\,\phi_0(\lambda) + A_1\,\phi_1(\lambda) + \cdots + A_d\,\phi_d(\lambda). \tag{2.18}$$

We consider polynomial bases defined in [3] satisfying a three-term recurrence

$$\lambda\,\phi_j(\lambda) = \alpha_j\,\phi_{j+1}(\lambda) + \beta_j\,\phi_j(\lambda) + \gamma_j\,\phi_{j-1}(\lambda), \quad \text{for } j = 1, 2, \ldots \tag{2.19}$$

where $\phi_{-1} \equiv 0$, $\phi_0 \equiv 1$, and for $j = 0, 1, \ldots$, α_j, β_j and γ_j are real, $\alpha_j > 0$ and $\alpha_j = \frac{c_j}{c_{j+1}}$, being c_j the leading coefficient of $\phi_j(\lambda)$. This type of polynomial sets include any sequence of orthogonal polynomials with increasing degree.

Some sets of polynomial bases satisfying (2.19) are:

- *Monomial.* Particular case taking $\alpha_j = 1$ and $\beta_j = \gamma_j = 0$, $\forall j \geq 0$.
- *Chebyshev first kind.* $T_0(\lambda) = 1$, $T_1(\lambda) = \lambda$, $T_{j+1}(\lambda) = 2\lambda T_j(\lambda) - T_{j-1}(\lambda)$, with coefficients $\alpha_0 = 1$, $\alpha_j = \frac{1}{2}$, $\beta_j = 0$, $\gamma_j = \frac{1}{2}$.
- *Chebyshev second kind.* $U_0(\lambda) = 1$, $U_1(\lambda) = 2\lambda$, $U_{j+1}(\lambda) = 2\lambda U_j(\lambda) - U_{j-1}(\lambda)$, with coefficients $\alpha_j = \frac{1}{2}$, $\beta_j = 0$, $\gamma_j = \frac{1}{2}$.
- *Legendre.* $P_0(\lambda) = 1$, $P_1(\lambda) = \lambda$, $(j+1)P_{j+1}(\lambda) = (2j+1)\lambda P_j(\lambda) + jP_{j-1}(\lambda)$, with coefficients $\alpha_j = j+1$, $\beta_j = -2j$, $\gamma_j = j$.
- *Laguerre.* $L_0(\lambda) = 1$, $L_1(\lambda) = 1-\lambda$, $(j+1)L_{j+1}(\lambda) = (2j+1-\lambda)L_j(\lambda) - jL_{j-1}(\lambda)$, with $\alpha_0 = -1$, $\alpha_j = -(j+1)$, $\beta_0 = 0$, $\beta_j = (2j+1)$, $\gamma_j = -j$.
- *Hermite.* $H_0(\lambda) = 1$, $H_1(\lambda) = 2\lambda$, $H_j + 1(\lambda) = 2\lambda H_j(\lambda) - 2jH_{j-1}(\lambda)$, with coefficients $\alpha_j = \frac{1}{2}$, $\beta_j = 0$, $\gamma_j = j$.

To linearize a polynomial eigenproblem defined in terms of a matrix polynomial as in (2.18), we use a linearization, $L(\lambda)$, similar to the one proposed in [3],

$$L(\lambda) = A - \lambda B, \tag{2.20}$$

$$A = \begin{bmatrix} \beta_0 I & \alpha_0 I & & & & & \\ \gamma_1 I & \beta_1 I & \alpha_1 I & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \gamma_{d-2} I & \beta_{d-2} I & \alpha_{d-2} I \\ \tilde{A}_0 & \tilde{A}_1 & \tilde{A}_2 & \cdots & \tilde{A}_{d-3} & \tilde{A}_{d-2} & \tilde{A}_{d-1} \end{bmatrix}, \; B = \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & c_d A_d \end{bmatrix},$$

with $\tilde{A}_j = -c_{d-1} A_j$ $(j = 0, \ldots, d-3)$, $\tilde{A}_{d-2} = -c_{d-1} A_{d-2} + c_d \gamma_{d-1} A_d$ and $\tilde{A}_{d-1} = -c_{d-1} A_{d-1} + c_d \beta_{d-1} A_d$. Note that, in the case of using the monomial basis, this linearization coincides with the first companion form (2.3). On the other hand, also when using the monomial basis, the linearization defined in [3] coincides with the second companion form related to (1.1). The reason of using linearization (2.20) lies in our interest in computing right eigenvectors of (2.18), which are easily obtained from those of (2.20). Indeed, it can be proved, in a similar way to [3], that the pencil (2.20) is a strong linearization of (2.18), so both eigenproblems have the same spectrum, and on the other hand that right eigenvectors, z, of (2.20) have the form

$$z = \begin{bmatrix} x \\ \phi_1(\lambda) x \\ \vdots \\ \phi_{d-1}(\lambda) x \end{bmatrix}, \tag{2.21}$$

where x is a right eigenvector of (2.18). Thus, by using this expression, eigenvectors of $P(\lambda)$ can be recovered easily from those of $L(\lambda)$.

Similarly to the expressions given in [2], it is possible to write the LU factorization of (2.20)

$$(A - \lambda B)\Pi = L_\lambda U_\lambda, \tag{2.22}$$

where $\Pi = \left[\begin{smallmatrix} 0 & I_n \\ I_{(d-1)n} & 0 \end{smallmatrix}\right]$ is a block permutation matrix, and

$$L_\lambda = \begin{bmatrix} \alpha_0 I & & & & & & \\ (\beta_1 - \lambda) I & \alpha_1 I & & & & & \\ \gamma_2 I & (\beta_2 - \lambda) I & \ddots & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & (\beta_{d-2} - \lambda) I & \alpha_{d-2} I & & \\ -c_{d-1} A_1 & -c_{d-1} A_2 & \ldots & \hat{A}_{d-2} & \hat{A}_{d-1} & -c_{d-1} P(\lambda) \end{bmatrix}, \tag{2.23}$$

$$U_\lambda = \begin{bmatrix} I & & & -\frac{\phi_1(\lambda)}{\phi_0(\lambda)} I \\ & I & & -\frac{\phi_2(\lambda)}{\phi_0(\lambda)} I \\ & & \ddots & \vdots \\ & & I & -\frac{\phi_{d-1}(\lambda)}{\phi_0(\lambda)} I \\ & & & I \end{bmatrix}, \tag{2.24}$$

with $\hat{A}_{d-2} = -c_{d-1}A_{d-2} + c_d\gamma_{d-1}A_d$ and $\hat{A}_{d-1} = -c_{d-1}A_{d-1} + c_d(\beta_{d-1} - \lambda)A_d$.

This factorization will be used later in §3.2, when adapting TOAR to general matrix polynomials in the form (2.18) for the shift-and-invert spectral transformation.

2.5. PETSc and SLEPc. We now briefly describe the structure and main features of SLEPc, that will be helpful for understanding some of the design decisions and implementation details of our polynomial eigensolvers.

SLEPc, the Scalable Library for Eigenvalue Problem Computations [16, 25], is a software library for the parallel solution of large-scale eigenvalue problems. It started as a collection of solvers for the linear eigenvalue problem, both standard and generalized, covering both the Hermitian and non-Hermitian cases in either real or complex arithmetic. It was later extended to also provide solvers for related problems. In this paper we discuss the solvers for the polynomial eigenproblem.

SLEPc is an extension of PETSc (Portable, Extensible Toolkit for Scientific Computation [6]), a parallel framework for the numerical solution of partial differential equations, which is based on defining abstract data objects such as vectors and matrices, and building solver objects on top of them. PETSc and SLEPc are designed to be scalable to a large number of processors, to be portable to virtually any parallel computer, and to be flexible in terms of run-time control of the solution process (one can for instance specify the solver at run time, or change relevant parameters such as the tolerance or the size of the subspace basis).

PETSc follows an object-oriented design, with all the code organized in a few data and solver classes. The application programmer interacts with objects of these classes with a simple interface, instead of diving into the details of underlying data structures. The basic data objects are index sets, vectors and matrices, on top of which stand different solver classes for linear and non-linear system of equations, and for integration of differential equations.

For the solution of linear systems, PETSc provides a number of iterative solvers such as GMRES, together with a variety of preconditioners including Jacobi (diagonal) preconditioning, and block Jacobi/additive Schwarz (with a choice of incomplete factorizations for the blocks). Complete factorizations (LU and Cholesky) are also included in the category of preconditioners. Furthermore, it is possible to use preconditioners available in third-party packages that are seamlessly integrated into PETSc. For instance, we are able to perform parallel direct linear solves with MUMPS [1].

SLEPc provides five solver classes. The `EPS` class covers the linear eigenvalue problem, while `PEP` contains the new solvers for polynomial eigenproblems, to be detailed in the rest of the paper. The other solver classes (for the SVD, matrix functions, and general nonlinear eigenproblems) will not be discussed here. Apart from the main solver classes, there are 5 auxiliary classes, the most relevant for our discussion being `ST` for specifying spectral transformations.

The `EPS` package provides a collection of linear eigensolvers, most of which are based on subspace projections. In particular, it includes a parallel implementation of Krylov-Schur [26]. In addition, other methods such as generalized Davidson, Jacobi-Davidson, conjugate gradients, and contour integral, are also included.

Every `EPS` object has an `ST` object internally, that is used to perform the spectral transformation discussed in §2.3. Both classes are designed in such a way that eigensolvers do not worry about which spectral transformation is being used. This confers the flexibility to combine any solver with any spectral transformation, even at run time. For instance, for a generalized problem $Ax = \lambda Bx$, the command line

```
$ ./example -eps_smallest_real -st_type shift
```

will search for the smallest eigenvalues of $B^{-1}Ax = \lambda x$, while

```
$ ./example -eps_target_magnitude -eps_target 0 -st_type sinvert
```

will obtain the dominant eigenvalues θ of $(A-\sigma B)^{-1}Bx = \theta x$ (for $\sigma = 0$ in this case), then backtransform them as $\lambda = \theta^{-1} + \sigma$.

In many cases, as illustrated above, `ST` needs to handle matrix inverses. This is done implicitly via linear solves with the aid of PETSc's `KSP` solvers together with the companion class `PC` for preconditioners. These objects included inside `ST` are also customizable, allowing, e.g., the specification of an inexact shift-and-invert strategy:

```
$ ./example -eps_target 0 -st_type sinvert -eps_tol 1e-7
            -st_ksp_type gmres -st_ksp_rtol 1e-9 -st_pc_type ilu
```

We finish this section with a short description of how SLEPc eigensolvers are parallelized. PETSc and SLEPc are oriented to large-scale computations on distributed memory parallel computers with a message-passing paradigm (MPI). Sparse matrices in PETSc are stored by blocks of rows, and vectors also follow the same data distribution, so every processor owns a contiguous chunk of the vector elements. So for instance the Arnoldi relation (2.5) is stored in this way, with the exception of the small, dense matrix H_k that is stored "sequentially" (meaning that all processes own a copy of the full matrix). The main operations involved in the computation of the Arnoldi relation are:

- Sparse matrix-vector product, that is implemented as usual in mesh-based computations, with nearest neighbour communication among processes.
- Vector operations requiring global communication, such as dot products and norms. In SLEPc, orthogonalization has been optimized to avoid global communication as much as possible [15].
- Other vector operations such as addition, that are trivially parallelizable.
- Computations on the small projected matrix, H_k. These are performed redundantly by all processes, since data is stored in all of them. Unless the size of the projected problem is large, this will not hinder parallel performance.

3. Polynomial eigensolver in SLEPc. In this section, we describe the most relevant aspects concerning the polynomial eigenproblem solvers developed in SLEPc. We will denote this solver class as `PEP`, as already mentioned in §2.5. Several methods currently available in `PEP` address the polynomial problem via linearization and use the variants of the Arnoldi iteration described in §2.2 to solve the linear eigenproblem. The main difference between those variants lie in the way Arnoldi vectors are stored and recovered, that is done by each variant as follows:

Plain Arnoldi stores full Arnoldi vectors of dimension d times the dimension of the polynomial eigenproblem, n. For quadratic eigenproblems there is the possibility of explicitly building the matrices for the linearized eigenproblem, and then solve the linear problem by any of the linear solvers available in SLEPc, such as Jacobi-Davidson (see §2.5). In parallel, we build the explicit matrices by interlacing the locally stored submatrices of the parallel matrices that compose it, with the aim of avoiding poor load balancing in the one-dimensional partitioning of the large matrix. For instance, in the case of a quadratic polynomial $P(\lambda) = K + \lambda C + \lambda^2 M$ with two processes, the first

matrix of the linearization, $A = \left[\begin{smallmatrix} 0 & I \\ -K & -C \end{smallmatrix}\right]$, is stored as

$$A = \left[\begin{array}{cc|cc} 0 & I & 0 & 0 \\ -K_{11} & -C_{11} & -K_{12} & -C_{12} \\ \hline 0 & 0 & 0 & I \\ -K_{21} & -C_{21} & -K_{22} & -C_{22} \end{array}\right], \tag{3.1}$$

where $[K_{11}, K_{12}]$ corresponds to the part of $K = \left[\begin{smallmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{smallmatrix}\right]$ locally stored at the first process, and similarly for C.

This option of building explicit matrices for the linearization is not available for degrees larger than 2 because of the increase in computational cost and memory requirements. Instead, the computations with such matrices are done efficiently by implicitly working with the matrices of the polynomial eigenproblem, as discussed in §3.1.

Q-Arnoldi stores only the first block (length n) of each Arnoldi vector generated, and works rebuilding the remaining blocks when necessary. Our implementation corresponds to the method described in [24], so it is available for quadratic eigenproblems defined in terms of the monomial basis. We have not extended this method for degree larger than 2 or for non-monomial bases.

TOAR reconstructs Arnoldi vectors when required, from an orthonormal basis in $\mathbb{C}^n$ that grows as the Krylov subspace is expanded. The implementation we have made is based on the description given in [22] for degree 2, and the generalization to arbitrary degree for polynomial eigenproblems defined in terms of Chebyshev basis described in [21].

These methods can be used together with the `shift` or `sinvert` spectral transformations, as described in §2.5. For a monomial basis, the transformation on the polynomial eigenproblem is supported for all three solvers. In this case, the initial problem is transformed according to (2.17), and the resulting problem is then linearized. An alternative is to do the transformation on the linearization, which is supported for any polynomial basis but only in the plain Arnoldi and TOAR methods.

3.1. Plain Arnoldi. The Q-Arnoldi and TOAR methods that try to reduce the memory requirements introduced by the linearization may have a disadvantage in terms of numerical error compared to plain Arnoldi. Errors present in the explicitly stored part of the basis are propagated and possibly amplified when reconstructing full Arnoldi vectors during the algorithm. For instance, in (2.8) a large value of $\|H_k\|$ results in numerical instability [24]. The plain Arnoldi method does not suffer from this problem since it computes and stores full Arnoldi vectors.

The plain Arnoldi solver in SLEPc behaves like the Krylov-Schur linear solver except that the `PEP` implementation avoids the explicit formation of the linearization matrices (optional for degree 2) and operates with them in an implicit form, using the coefficient matrices of the initial problem, A_i. This computation is intimately related to the linearization used as well as the spectral transformation possibly applied. For instance, equation (2.7) shows how to compute the product of $B^{-1}A$ times a vector, for the problem (2.3) when no spectral transformation is being used. We next discuss how to operate implicitly with the matrices of the more general linearization (2.20), associated with a polynomial problem (2.18), when either the shift or shift-and-invert spectral transformation is being done on the linearization.

In the case of the shift transformation, it is immediate to derive expressions to compute the matrix-vector product $w = B^{-1}(A - \sigma B)v$, for A and B defined as in

(2.20) and $v \in \mathbb{C}^{dn}$, referencing the matrices $\{A_j\}_{j=0}^{d-1}$ only,

$$\begin{cases} w^0 = (\beta_0 - \sigma)v^0 + \alpha_0 v^1, \\ w^j = \gamma_j v^{j-1} + (\beta_j - \sigma)v^j + \alpha_j v^{j+1}, \quad j = 1, \ldots, d-2, \\ w^{d-1} = -\alpha_{d-1} A_d^{-1}\Big(\sum_{j=0}^{d-1} A_j v^j\Big) + \gamma_{d-1} v^{d-2} + (\beta_{d-1} - \sigma)v^{d-1}. \end{cases} \tag{3.2}$$

To deduce similar expressions when using the shift-and-invert transformation, we consider the block LU decomposition for $(A - \sigma B)$ given in (2.22). To compute

$$w = (A - \sigma B)^{-1} B v = \Pi U_\sigma^{-1} L_\sigma^{-1} B v, \tag{3.3}$$

first we consider $t = L_\sigma^{-1} B v$, which is computed using the recurrence resulting from block forward elimination of $L_\sigma t = Bv$,

$$\begin{cases} t^0 = \alpha_0^{-1} v^0, \\ t^1 = \alpha_1^{-1}(v^1 + (\sigma - \beta_1)t^0), \\ t^j = \alpha_j^{-1}(v^j + (\sigma - \beta_j)t^{j-1} - \gamma_j t^{j-2}), \quad j = 2, \ldots, d-2, \\ t^{d-1} = -P(\sigma)^{-1}(A_1 t^0 + \cdots + A_{d-2} t^{d-3} + A_{d-1} t^{d-2} + A_d \hat{t}^{d-1}), \end{cases} \tag{3.4}$$

where $\hat{t}^{d-1} = \alpha_{d-1}^{-1}(-\gamma_{d-1} t^{d-3} + (\sigma - \beta_{d-1})t^{d-2} + v^{d-1})$. Then, considering the block backward substitution of $U_\sigma \Pi^{-1} w = t$, we have

$$\begin{cases} w^0 = t^{d-1} \\ w^j = \dfrac{\phi_j(\sigma)}{\phi_0(\sigma)} t^{d-1} + t^{j-1}, \quad j = 1, \ldots, d-1, \end{cases} \tag{3.5}$$

where vectors $\{t^j\}_{j=0}^{d-1}$ are computed using the recurrence (3.4).

3.2. TOAR. When computing an Arnoldi relation for the matrix $S := B^{-1}A$ from a linearization (2.20), we obtain equations similar to (2.8) that relate different blocks, $\{V_k^i\}_{i=0}^{d-1}$, of the matrix representing Arnoldi vectors. These equations are

$$\begin{cases} V_k^1 = \alpha_0^{-1}(-\beta_0 V_k^0 + V_k^0 H_k + h_{k+1,k} v_{k+1}^0 e_k^*) \\ V_k^i = \alpha_{i-1}^{-1}(-\beta_{i-1} V_k^{i-1} - \gamma_{i-1} V_k^{i-2} + \\ \qquad + V_k^{i-1} H_k + h_{k+1,k} v_{k+1}^{i-1} e_k^*), \qquad i = 2, \ldots, d-1. \end{cases} \tag{3.6}$$

Relations (3.6) imply that the equality (2.11) holds, and a TOAR approach can be used in this case to represent Arnoldi vectors in the form (2.12). This is also the case when a spectral transformation such as shift or shift-and-invert is used. In this latter case, for example, we can see this by equating each block row of the equation

$$BV_k = (A - \sigma B)V_k H_k + h_{k+1,k}(A - \sigma B) v_{k+1} e_k^*, \tag{3.7}$$

that results from using Arnoldi on $S = (A - \sigma B)^{-1} B$.

With the TOAR method, Arnoldi vectors are represented as

$$v_k = \begin{bmatrix} U_{k+d} & & \\ & \ddots & \\ & & U_{k+d} \end{bmatrix} \begin{bmatrix} g_k^0 \\ \vdots \\ g_k^{d-1} \end{bmatrix}, \tag{3.8}$$

U_{k+d} having orthonormal columns, and the generated Arnoldi relation takes the form

$$S\hat{U}_{k+d}G_k = \hat{U}_{k+d}G_kH_k + h_{k+1,k}\hat{U}_{k+d}g_{k+1}e_k^*, \tag{3.9}$$

where $\hat{U}_{k+d} := I_d \otimes U_{k+d}$.

Algorithm 1 TOAR for linearized polynomial eigenproblems

Input: Matrices $\{A_i\}_{i=0}^d \subset \mathbb{C}^{n\times n}$ defining the matrix polynomial, initial vectors $\{w^i\}_{i=0}^{d-1} \subset \mathbb{C}^n$, number of iterations $k \in \mathbb{N}$

Output: $H_k \in \mathbb{C}^{k\times k}$, $h_{k+1,k} \in \mathbb{R}$, $U_{k+d} \in \mathbb{C}^{n\times(k+d)}$, $\{G_k^i\}_{i=0}^d \subset \mathbb{C}^{(k+d)\times k}$, $\{g_{k+1}^i\}_{i=0}^{d-1} \subset \mathbb{C}^{k+d}$ satisfying (3.9) for G_k and g_{k+1} split in the form (2.1)

1: $G_0 \leftarrow [\ ]$, $H_0 \leftarrow [\ ]$

2: /* Orthogonalize initial vectors $\{w^i\}_{i=0}^{d-1}$ */
$[Q,R] = \mathrm{qr}(\begin{bmatrix} w^0 & \cdots & w^{d-1}\end{bmatrix})$ /* reduced QR factorization */
$U_d \leftarrow Q$

3: /* Normalize first Arnoldi vector */
$g_1 \leftarrow \mathrm{vec}\, R/\|\,\mathrm{vec}\, R\,\|$

4: **for** $j = 1,\ldots,k$ **do**

5: /* Expand Krylov subspace */
$[u_{j+d}, g] = \mathrm{expand}(\{A_i\}_{i=0}^d, U_{j-1+d}, g_j)$
$U_{j+d} \leftarrow \begin{bmatrix} U_{j-1+d} & u_{j+d}\end{bmatrix}$

6: /* Gram-Schmidt orthogonalization */
$G_j^i \leftarrow \begin{bmatrix} G_{j-1}^i & g_j^i \\ 0 & 0\end{bmatrix}$, $i = 0,\ldots,d-1$
$h_j = G_j^* g$, $\hat{g} \leftarrow g - G_j h_j$

7: /* Normalize new Arnoldi vector */
$h_{j+1,j} = \|\hat{g}\|$, $g_{j+1} \leftarrow \hat{g}/h_{j+1,j}$

8: /* Update Hessenberg matrix */
$H_j \leftarrow \left[\begin{array}{c|c} H_{j-1} \\ 0 & h_j\end{array}\right]$

9: **end for**

The main steps of the TOAR implementation in SLEPc are summarized in Algorithm 1. In general, they are a direct extension, for degree larger than 2, of those described in §2.2 for the quadratic TOAR.

For the sake of simplicity, in step 2 we suppose that a basis of d vectors (columns of U_d) is built, and then it is extended with one vector per iteration (step 5). The actual SLEPc implementation, however, takes into account that the matrix Q resulting from the initial QR factorization could have less than d columns, and that the vector generated in the jth iteration (step 5) could belong to the range of U_{j-1+d}, resulting in the number of columns of this matrix not being increased.

To extend the Krylov subspace, by computing $w = Sv_j$, the TOAR method obtains u_{j+d} and g such that $w = (I_d \otimes [\,U_{j-1+d}\ u_{j+d}\,])g$. In step 5, this computation is indicated with the function $[u_{j+d}, g] = \mathrm{expand}(\{A_i\}_{i=0}^d, U_{j-1+d}, g_j)$, where the last Arnoldi vector computed is represented by the input parameters, $v_j = (I_d \otimes U_{j-1+d})g_j$. Once u_{j+d} and g have been computed, the orthogonalization and normalization of w is made implicitly by orthogonalizing and normalizing g in the same way as discussed in §2.2 (steps 6 and 7 of Algorithm 1).

As in plain Arnoldi, the computation of the expansion vector, w, depends on the linearization and spectral transformation used on the linearized problem. Let us see now how to carry out this operation (step 5), when either the shift or shift-and-invert spectral transformation is used.

When the shift transformation is being used, from (3.2) we have that w^i belongs to the range of U_{j-1+d}, for $i = 0, \ldots, d-2$, and we define

$$\begin{cases} g^0 := (\beta_0 - \sigma)g_j^0 + \alpha_0 g_j^1, \\ g^i := \gamma_i g_j^{i-1} + (\beta_i - \sigma)g_j^i + \alpha_i g_j^{i+1}, \quad i = 1, \ldots, d-2. \end{cases}$$

On the other hand, vector w^{d-1} is computed as in (3.2), replacing each v_j^i by $U_{j-1+d}g_j^i$, $i = 0, \ldots, d-1$. This vector is used to extend the orthonormal set U_{j-1+d}, so we define

$$u_{j+d} := \frac{\hat{u}}{\|\hat{u}\|_2}, \quad \text{and} \quad g^{d-1} := \begin{bmatrix} \hat{g} \\ \|\hat{u}\|_2 \end{bmatrix}, \tag{3.10}$$

being $\hat{g} := U_{j-1+d}^* w^{d-1}$ and $\hat{u} := w^{d-1} - U_{j-1+d}\hat{g}$.

For the shift-and-invert spectral transformation, using recurrences (3.4) and (3.5), we first define coefficients

$$\begin{cases} \tilde{g}^0 := \alpha_0^{-1} g_j^0, \\ \tilde{g}^1 := \alpha_1^{-1}(g_j^1 + (\sigma - \beta_1)\tilde{g}^0), \\ \tilde{g}^i := \alpha_i^{-1}(g_j^i + (\sigma - \beta_i)\tilde{g}^{i-1} - \gamma_i \tilde{g}^{i-2}), \quad i = 2, \ldots, d-2, \end{cases} \tag{3.11}$$

that are used to compute the vector t^{d-1} as in (3.4), replacing each t^i by $U_{j-1+d}\tilde{g}^i$ for $i = 0, \ldots, d-2$. Secondly, this vector is used to extend U_{j-1+d}, defining $g^0 := \begin{bmatrix} \hat{g} \\ \|\hat{u}\|_2 \end{bmatrix}$ and $u_{j+d} := \frac{\hat{u}}{\|\hat{u}\|_2}$, being $\hat{g} := U_{j-1+d}^* t^{d-1}$ and $\hat{u} := t^{d-1} - U_{j-1+d}\hat{g}$. Finally, coefficients g^i are defined in the form $g^i := \frac{\phi_i(\sigma)}{\phi_0(\sigma)} g^0 + \tilde{g}^{i-1}$, $i = 1, \ldots, d-1$.

3.3. Restarting the Krylov method. The default SLEPc linear eigensolver and the methods described in this work are based on the Krylov-Schur method [26], which is a restarted variant of Arnoldi. This variant uses Krylov relations,

$$SV_k = V_k C + v_{k+1} b^*, \tag{3.12}$$

that are more general than Arnoldi relations (2.5), since matrix C does not necessarily have Hessenberg form and vector b can be different from e_k. Columns of V_k are Krylov vectors, not Arnoldi vectors, and they also generate the Krylov subspace $\mathcal{K}_k$. One advantage of using Krylov relations is that they can be truncated to other relations with smaller dimension, maintaining the most desirable directions in the corresponding Krylov subspace while unwanted directions are filtered out.

The restart procedure uses the fact that, provided C has the form $C = \left[\begin{smallmatrix} C_1 & \star \\ 0 & C_2 \end{smallmatrix}\right]$, with $C_1 \in \mathbb{C}^{p\times p}$ and $C_2 \in \mathbb{C}^{q\times q}$, relation (3.12) can be truncated to get another Krylov relation of dimension p, that can later be extended again to order k. The goal is to maintain a bounded subspace dimension while keeping the most valuable directions. This restart can be accomplished as follows:

1. Compute the (real) Schur decomposition of C, $CQ_k = Q_k T$, sorted so that the p most wanted Ritz values remain in the leading diagonal block of T.
2. Write $Q_k = [Q_p, Q']$ and truncate the Krylov relation to order p,

   $$S\tilde{V}_p = \tilde{V}_p T_1 + v_{k+1}\tilde{b}^*, \tag{3.13}$$

   where $T_1 = Q_p^* C Q_p$, $\tilde{V}_p = V_k Q_p$ and $\tilde{b} = Q_p^* b$.
3. Restart the Arnoldi iteration, expanding the Krylov subspace by computing $w = Sv_{k+1}$ and orthogonalizing it against the columns of the transformed $\tilde{V}_p$.

For Q-Arnoldi, the `PEP` solver also follows a Krylov-Schur restart scheme. This is possible because after truncating the Krylov relation (3.12), producing (3.13), a relation analog to (2.8) still holds for the updated matrix of Krylov vectors, $\tilde{V}_p$.

In the case of the TOAR variant, updating a Krylov relation of the type (3.9) by multiplying on the right by a unitary matrix Q_k, produces a new one in which the updated matrix of Krylov vectors takes the form $V_p = (I_d \otimes U_{k+d})G_kQ_p$. This means that only matrix G_k is updated and truncated, and the TOAR basis matrix U_{k+d} remains unchanged, so the number of columns in U_{k+d} and rows in G_p are still $k+d$, hence, something else is needed for reducing the TOAR basis size as well. For this, we follow the procedure described in [21] that uses the fact that matrix $\Omega_{p+1} := \left[G_p^0, g_{k+1}^0, \ldots, G_p^{d-1}, g_{k+1}^{d-1}\right]$ has the same rank as $\left[V_p^0, v_{k+1}^0, \ldots, V_p^{d-1}, v_{k+1}^{d-1}\right]$, that is, $p+d$, to ensure that the compact SVD of Ω_{p+1},

$$\Omega_{p+1} = \check{U}\check{\Sigma}\check{V}^*, \tag{3.14}$$

has a matrix of singular values, $\check{\Sigma}$, of dimension less than or equal to $p+d$. Although in our implementation we consider that this dimension can be less than $p+d$, for simplicity, now we suppose that it is $p+d$, and that $\check{U}$ and $\check{V}$ have dimensions $(k+d)\times(p+d)$ and $d(p+1)\times(p+d)$, respectively.

After truncating, Krylov vectors satisfy $\begin{bmatrix}V_p^i & v_{k+1}^i\end{bmatrix} = U_{k+d}\begin{bmatrix}G_p^i & g_{k+1}^i\end{bmatrix}$, $i = 0,\ldots,d-1$, but each G_p^i has $(k+d)$ rows and U_{k+d} $(k+d)$ columns. Then, using decomposition (3.14), splitting $\check{V}^* = \left[\check{V}^0, \ldots, \check{V}^{d-1}\right]$ (with $\check{V}^i \in \mathbb{C}^{(p+d)\times(p+1)}$), and updating matrices $U_{p+d} \leftarrow U_{k+d}\check{U}$ and $G_{p+1}^i \leftarrow \check{\Sigma}\check{V}^i$, $i = 0,\ldots,d-1$, we obtain that Krylov vectors can be expressed in the form (2.12), with matrices G_{p+1}^i and U_{p+d} having $p+d$ rows and columns, respectively.

3.4. Locking converged eigenpairs. The Krylov-Schur restart [26] implemented in SLEPc's linear eigensolvers considers the possibility of deflating converged invariant subspaces, thus facilitating the convergence of eigenvectors outside of them. In addition, this deflation procedure is carried out in such a way that Krylov vectors associated with converged subspaces are locked, and are no longer modified, which results in reduced computational cost.

After truncating an Arnoldi relation following the Krylov-Schur scheme, we obtain a Krylov relation (3.13) where T_1 is an upper (block) triangular matrix. The columns of $\tilde{V}_q$, the leading q columns of $\tilde{V}_p$ in (3.13), span an invariant subspace of S when the first q entries of vector $\tilde{b}$ are zero. On the other hand, when $\tilde{V}_q$ is an approximate invariant subspace of S, the first q entries of $\tilde{b}$ are nonzero, but very small, and could be forced to be zero. In this case, when relation (3.13) is extended to another one of order k, the projected matrix H_k is a reduced upper Hessenberg matrix of the form

$$H_k = \left[\begin{array}{cc|c} T_{11} & T_{12} & M_1 \\ 0 & T_{22} & M_2 \\ \hline 0 & b_2^* & \\ 0 & 0 & H \end{array}\right], \tag{3.15}$$

where b_2 represents the $(p–q)$ last positions of $\tilde{b}$, the matrix T_1 is written as $\left[\begin{smallmatrix} T_{11} & T_{12} \\ 0 & T_{22}\end{smallmatrix}\right]$, T_{11} having dimension $q\times q$, and H is an upper Hessenberg matrix. Hence, to reduce H_k to (block) triangular form, we can use a unitary matrix of the form $\tilde{Q} = \left[\begin{smallmatrix} I_q & 0 \\ 0 & Q\end{smallmatrix}\right]$, Q also being a unitary matrix. In this way, the dimension of the projected problem to be solved is reduced by q, but also the number of Arnoldi vectors to be updated is reduced, since the first q vectors remain unchanged when using $\tilde{Q}$ to update V_k.

In the case of the TOAR method, when updating Krylov vectors represented in the TOAR way, $V_k = (I_d \otimes U_{k+d})G_k$, making zeros in the first positions of b has the same effect, as explained above, of reducing the dimension of the projected problem and also leaving the leading columns of matrix G_k untouched, but in general, when restarting at the next iteration (see §3.3), the full set of TOAR vectors U_{k+d} will be updated. Thus, in order to achieve effective computational savings in the locking variant of TOAR, the restarting procedure described in §3.3 has been modified so that the corresponding vectors in the TOAR basis are also locked when deflating. Next we give an explanation of this new restarting procedure included in the TOAR solver.

PROPOSITION 3.1. *Consider a Krylov relation of order k, generated using the TOAR method on the linearization* (2.20)*,*

$$SV_k = V_k T_k + v_{k+1} b^*, \tag{3.16}$$

with $V_k = (I_d \otimes U_{k+d})G_k$ and T_k (block) upper triangular, with structure

$$T_k = \begin{array}{c} \\ {\scriptstyle q} \\ {\scriptstyle p-q} \\ {\scriptstyle k-p} \end{array} \begin{array}{c} \begin{array}{ccc} {\scriptstyle q} & {\scriptstyle p-q} & {\scriptstyle k-p} \end{array} \\ \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ & T_{22} & T_{23} \\ & & T_{33} \end{bmatrix} \end{array}. \tag{3.17}$$

Also, suppose that the first q entries of b are zero. Then there exist matrices $\tilde{U}_{p+d} \in \mathbb{C}^{n\times(p+d)}$ and $\tilde{G}_{p+1} \in \mathbb{C}^{d(p+d)\times(p+1)}$ with $\tilde{G}^i_{p+1} = \left[\begin{smallmatrix} \tilde{G}^i_{11} & \tilde{G}^i_{12} \\ 0 & \tilde{G}^i_{22} \end{smallmatrix}\right]$ and $\tilde{G}^i_{11} \in \mathbb{C}^{q\times q}$, for $i = 0, \ldots, d-1$, such that $\begin{bmatrix} V_p & v_{k+1} \end{bmatrix} = (I_d \otimes \tilde{U}_{p+d})\tilde{G}_{p+1}$.

Proof. Since the first q entries of b are zero, defining $T_q = T_{11}$ and $G_q \in \mathbb{C}^{d(k+d)\times q}$ as the first q columns of G_k, relation

$$S(I_d \otimes U_{k+d})G_q = (I_d \otimes U_{k+d})G_q T_q, \tag{3.18}$$

holds. Taking into account the block structure of S, equation (3.18) gives relations between the block rows of V_q, $V_q^i = U_{k+d}G_q^i$, revealing that the rank of $[V_q^0, \ldots, V_q^{d-1}]$, and hence of $\Omega_1 := \begin{bmatrix} G_q^0, \ldots, G_q^{d-1} \end{bmatrix}$, is q. Thus, working as in the restart procedure described in §3.3, using the compact SVD of $\Omega_1 = \check{U}_1\check{\Sigma}_1\check{V}_1^*$, and splitting $\check{V}_1^*$ as $\check{V}_1^* = \begin{bmatrix} \check{V}_1^0, \ldots, \check{V}_1^{d-1} \end{bmatrix}$, we have

$$V_q^i = U_{k+d}G_q^i = U_{k+d}\check{U}_1\check{\Sigma}_1\check{V}_1^i, \quad i = 0, \ldots, d-1. \tag{3.19}$$

On the other hand, when truncating (3.16) to another relation of order p, resulting in

$$S(I_d \otimes U_{k+d})G_p = (I_d \otimes U_{k+d})G_p T_p + U_{k+d}g_{k+1}b_p^*, \tag{3.20}$$

where b_p represents the first p elements of b, we have that $\begin{bmatrix} G_p^0, g_{k+1}^0, \ldots, G_p^{d-1}, g_{k+1}^{d-1} \end{bmatrix}$ has rank $p+d$. Also, if we split $G_p = [G_q, G_2]$ and let $\Omega_2 := \begin{bmatrix} G_2^0, g_{k+1}^0, \ldots, G_2^{d-1}, g_{k+1}^{d-1} \end{bmatrix}$ and $\tilde{\Omega}_2 := (I_{k+d} - \check{U}_1\check{U}_1^*)\Omega_2$, it follows that $\tilde{\Omega}_2$ will be of rank (at most) $(p+d-q)$. Thus, considering the compact SVD of $\tilde{\Omega}_2 = \check{U}_2\check{\Sigma}_2\check{V}_2^*$, we have $\Omega_2 = \check{U}_2\check{\Sigma}_2\check{V}_2^* + \check{U}_1\check{U}_1^*\Omega_2$, and splitting $\check{V}_2^*$ as $\check{V}_2^* = \begin{bmatrix} \check{V}_2^0, \ldots, \check{V}_2^{d-1} \end{bmatrix}$ (with $\check{V}_2^i \in \mathbb{C}^{(p+d-q)\times(p+1)}$), we conclude

$$\begin{aligned} V_p^i &= U_{k+d}\begin{bmatrix} G_q^i & G_2^i \end{bmatrix} = U_{k+d}\begin{bmatrix} \check{U}_1\check{\Sigma}_1\check{V}_1^i & \check{U}_2\check{\Sigma}_2\check{V}_2^i + \check{U}_1\check{U}_1^*\begin{bmatrix} G_2^i & g_{k+1}^i \end{bmatrix} \end{bmatrix} = \\ &= U_{k+d}\begin{bmatrix} \check{U}_1 & \check{U}_2 \end{bmatrix}\check{G}^i \end{aligned} \tag{3.21}$$

where $\check{G}^i := \begin{bmatrix} \check{\Sigma}_1\check{V}_1^i & \check{U}_1^*\begin{bmatrix} G_2^i & g_{k+1}^i \end{bmatrix} \\ 0 & \check{\Sigma}_2\check{V}_2^i \end{bmatrix}$. Finally, the result follows from (3.21), defining $\tilde{U}_{p+d} := U_{k+d}\begin{bmatrix} \check{U}_1 & \check{U}_2 \end{bmatrix}$ and $\tilde{G}^i_{p+1} := \check{G}^i$, for $i = 0, \ldots, d-1$. ∎

Notes apropos of Proposition 3.1.

(i) For simplicity, we have supposed that a TOAR basis of dimension $p+d$ is required to express the basis $[V_p, v_{k+1}]$, although, as pointed out in §3.3, the actual implementation takes into account that it could be smaller.

(ii) In finite precision arithmetic, it could happen that the computed basis $\begin{bmatrix}\check{U}_1 & \check{U}_2\end{bmatrix}$ is not orthogonal, so the restart implementation includes another step to compute the QR factorization $\begin{bmatrix}\check{U}_1 & \check{U}_2\end{bmatrix} = \check{Q}\check{R}$ that is then used to define $\tilde{U}_{p+d} := U_{k+d}\check{Q}$ and $\tilde{G}^i_{p+1} := \check{R}\check{G}^i$, for $i = 0, \ldots, d-1$.

Now we suppose that with the conditions of Proposition 3.1, the Krylov relation (3.16) is truncated and then extended anew to another one of order k, giving a similar relation with a projected matrix as in (3.15), and Krylov vectors $\begin{bmatrix}V_k & v_{k+1}\end{bmatrix} = (I_d \otimes U_{k+d})\begin{bmatrix}G_k & g_{k+1}\end{bmatrix}$ in which matrices G^i_k have the form $\left[\begin{smallmatrix} G^i_{11} & G^i_{12} \\ 0 & G^i_{22}\end{smallmatrix}\right]$, with $G^i_{11} \in \mathbb{C}^{q\times q}$, for $i = 0, \ldots, d-1$. In this situation, when updating the basis V_k by a truncated unitary matrix in the form $\tilde{Q}_t = \left[\begin{smallmatrix} I_q & 0 \\ 0 & Q\end{smallmatrix}\right]\left[\begin{smallmatrix} I_p \\ 0\end{smallmatrix}\right] = \left[\begin{smallmatrix} I_q & 0 \\ 0 & Q_t\end{smallmatrix}\right]$, being Q_t the first $p-q$ columns of Q, then the following relation holds for $i = 0, \ldots, d-1$,

$$\begin{bmatrix}V^i_k\tilde{Q}_t & v_{k+1}\end{bmatrix} = U_{k+d}\begin{bmatrix}G^i_k\tilde{Q}_t & g^i_{k+1}\end{bmatrix} = U_{k+d}\begin{bmatrix} G^i_{11} & G^i_{12}Q_t & g^i_1 \\ 0 & G^i_{22}Q_t & g^i_2\end{bmatrix}, \tag{3.22}$$

where we denote $g^i_{k+1} = \left[\begin{smallmatrix} g^i_1 \\ g^i_2\end{smallmatrix}\right]$, with $g^i_1 \in \mathbb{C}^{q\times 1}$, for $i = 0, \ldots, d-1$. Thus, defining $\Omega := \begin{bmatrix}G^0_{22}Q_t & g^0_2 & \cdots & G^{d-1}_{22}Q_t & g^{d-1}_2\end{bmatrix}$, which has rank $(p-q+d)$ at most, and considering now the SVD of $\Omega = \check{U}\check{\Sigma}\check{V}^*$, we have

$$\begin{bmatrix}V^i_k\tilde{Q}_t & v_{k+1}\end{bmatrix} = \tilde{U}_{p+d}\begin{bmatrix} G^i_{11} & \begin{bmatrix}G^i_{12}Q_t & g^i_1\end{bmatrix} \\ 0 & \check{\Sigma}\check{V}^i\end{bmatrix}, \tag{3.23}$$

where $\check{V}^* = \begin{bmatrix}\check{V}^0, \ldots, \check{V}^{d-1}\end{bmatrix}$, and $\tilde{U}_{p+d} := U_{k+d}\begin{bmatrix} I_q & 0 \\ 0 & \check{U}\end{bmatrix} \in \mathbb{C}^{n\times(p+d)}$.

In this way, when implicit Krylov vectors V_q are locked, also the first q TOAR vectors, U_q, will remain locked in subsequent restarts, and will never be modified, thus reducing the number of vectors to be updated as well as the dimension of the SVD required in the restart procedure.

3.5. Accuracy assessment. For measuring the quality of the computed eigenpairs we use the relative backward error, which is defined for an approximate right eigenpair (x, λ) of P as in (2.18) by

$$\eta_P(x,\lambda) = \min\{\epsilon : (P(\lambda) + \Delta P(\lambda))\, x = 0,\ \|\Delta A_j\|_2 \le \epsilon\|A_j\|_2,\ j = 0, \ldots, d\}. \tag{3.24}$$

Tisseur [28] gives an explicit expression of the backward error for an approximate right eigenpair (x, λ) of a polynomial eigenproblem defined in terms of the monomial basis (1.1). In case of using other polynomial bases we use the expression given by:

PROPOSITION 3.2. *The backward error of an approximate right eigenpair (x, λ) of P defined in* (2.18) *is given by the formula*

$$\eta_P(x,\lambda) = \frac{\|P(\lambda)x\|_2}{\left(\sum_{i=0}^d |\,\phi_i(\lambda)|\|A_i\|_2\right)\|x\|_2}. \tag{3.25}$$

Proof. For ϵ and $\{A_j\}_{j=0}^d$ such that $\|\Delta A_j\|_2 \leq \epsilon\|A_j\|_2$ and $(P(\lambda)+\Delta P(\lambda))\,x=0$, we have that $\|P(\lambda)x\|_2 \leq \|\Delta P(\lambda)\|_2\|x\|_2 \leq \left(\sum_{i=0}^d |\,\phi_i(\lambda)|\|A_i\|_2\right)\|x\|_2\epsilon$, and hence the right-hand side of (3.25) is less than or equal to the left-hand side.

Given $\Delta A_j := -\frac{1}{\sum_{i=0}^d |\,\phi_i(\lambda)|\|A_i\|_2}\operatorname{sign}\phi_j(\lambda)\|A_j\|_2 P(\lambda)x\frac{x^*}{\|x\|_2}$, where $\operatorname{sign}\mu := \bar{\mu}/|\mu|$ if $\mu\neq 0$ and zero otherwise, the reverse inequality follows since these perturbations ΔA_j satisfy $\|\Delta A_j\|_2 < \eta_P(x,\lambda)\|A_j\|_2$ and $(P(\lambda)+\Delta P(\lambda))\,x=0$. ∎

To decide about convergence in the PEP solver we use a bound of the backward error of the computed approximate eigenpairs for the associated linearized eigenproblem. In this case, for problem (2.20) the backward error takes the form

$$\eta_L(z,\lambda) = \frac{\|L(\lambda)z\|_2}{(\|A\|_2+\|B\|_2|\lambda|)\,\|z\|_2}. \tag{3.26}$$

When (z,λ) represents a Ritz pair computed from a Krylov relation, the expression (2.6) gives a cheap way for bounding the value of $\eta_L(z,\lambda)$,

$$\eta_L(z,\lambda) \leq \frac{\|B\|_2\|(B^{-1}A-I)z\|_2}{(\|A\|_2+\|B\|_2|\lambda|)\,\|z\|_2} \leq \frac{\|B\|_2|h_{k+1,k}e_k^*y|}{\|A\|_2+\|B\|_2|\lambda|}. \tag{3.27}$$

Note that in (3.27) Ritz vectors are assumed to have unit 2-norm. In the case of using the shift-and-invert spectral transformation ($\lambda=\frac{1}{\theta}+\sigma$) on the linearized problem, we have the backward error bound

$$\eta_L(z,\lambda) \leq \frac{\|A-\sigma B\|_2|h_{k+1,k}e_k^*y|}{|\theta|\,(\|A\|_2+\|B\|_2|\lambda|)}. \tag{3.28}$$

We use expressions (3.27) and (3.28), with ∞-norms instead of 2-norms, in the convergence criterion to determine which of the computed Ritz pairs are good approximate eigenpairs of the linearized eigenproblem, and should be accepted as converged.

In addition, other stopping criteria are also available. These consider the residual norm associated with a Ritz pair in the Arnoldi relation, using $\rho(z,\lambda)$ of (2.6). This value, either in absolute terms or relative to $|\theta|$, is used to decide about the convergence of the approximate eigenpair of the linearized eigenproblem. The default in PEP solvers is the one relative to $|\theta|$, a cheap alternative that approximates expressions (3.27) and (3.28) without needing to compute any matrix norms.

3.6. Scaling. We now briefly describe scaling techniques that have been incorporated in our PEP solver. These techniques, proposed by Betcke [7] for the monomial case (1.1), include a diagonal scaling of the matrices defining the polynomial eigenproblem, which affects the sensitivity of the eigenvalues, and a parameter scaling that affects the conditioning of the linearized eigenproblem.

Diagonal scaling. The original polynomial $P(\lambda)$ is transformed by multiplying it on both sides by two nonsingular diagonal matrices, D_1 and D_2,

$$\tilde{P}(\lambda) := D_1P(\lambda)D_2. \tag{3.29}$$

The resulting polynomial eigenproblem has the same eigenvalues as P, whereas eigenvectors x and $\tilde{x}$ of P and $\tilde{P}$, respectively, are related as $\tilde{x}=D_2^{-1}x$. The diagonal scaling procedure described in [7] aims at minimizing the backward error associated with a particular eigenvalue λ, so it will depend on the set of eigenvalues to be computed. Betcke describes an algorithm for efficiently

computing matrices D_1 and D_2, being one of the inputs an estimation of the magnitude of the desired eigenvalues. Due to the extra computational cost of obtaining D_1 and D_2, we have left this feature as an option for the case of polynomial eigenproblems with foreseen conditioning difficulties.

Parameter scaling. This strategy performs a transformation on the eigenvalue parameter of the polynomial eigenproblem, $\lambda = \rho\theta$, taking $\rho := \sqrt[d]{\|A_0\|_2/\|A_d\|_2}$. This transformation yields an equivalent eigenproblem with matrix polynomial $\tilde{P}(\theta) := P(\rho\theta) = \sum_{j=0}^{d} \theta^j \tilde{A}_j$, with $\tilde{A}_j := \rho^j A_j$, which has the same eigenvectors as the original polynomial eigenproblem, and related eigenvalues $\theta = \frac{\lambda}{\rho}$. This parameter scaling, described in [7], is intended for polynomial eigenproblems solved via linearization and aims at minimizing the condition number of the computed eigenvalues in the linearization (2.3), not on the polynomial eigenproblem (1.1). It is a generalization for polynomial eigenproblems of the scaling proposed by Fan, Lin, and Van Dooren for quadratic eigenproblems in [12]. This latter scheme also includes a global scaling factor,

$$\delta_2 := \frac{2}{\|A_0\|_2 + \|A_1\|_2\sqrt{\|A_0\|_2/\|A_2\|_2}}, \tag{3.30}$$

to scale all three matrices and produce a transformed quadratic eigenproblem whose matrix norms are close to one. The SLEPc implementation of this scaling uses the ∞-norm instead of the 2-norm,

$$\rho := \sqrt[d]{\frac{\|A_0\|_\infty}{\|A_d\|_\infty}}, \tag{3.31}$$

and includes, also for degree larger than 2, a global scaling factor given by

$$\delta := \frac{d}{\|A_0\|_\infty + \|\rho A_1\|_\infty + \cdots + \|\rho^{d-1} A_{d-1}\|_\infty}, \tag{3.32}$$

that is, the reciprocal of the average of the norms of the scaled coefficient matrices (except A_d), which coincides for $d = 2$ with the expression (3.30) when the 2-norm is used.

The scaling procedures described above are carried out implicitly without modifying the matrices of the original polynomial. The scaling is applied properly on the vectors that participate in each multiplication made with the matrices to be scaled.

Even though our `PEP` solver does not compute the scaling parameters in the case of polynomial eigenproblems defined in a non-monomial basis (2.18), it provides the possibility of performing the two mentioned scaling procedures if the various scaling parameters are supplied by the user. For the diagonal scaling, two vectors defining the right and left diagonal matrices are required, and these matrices pre- and post-multiply every matrix in (2.18) in the same way as in the monomial case.

For the case of performing a parameter scaling, $\lambda = \rho\theta$, the solver is applied to a transformed problem $\tilde{P}(\theta) = \sum_{j=0}^{d} \tilde{\phi}_j(\theta)\tilde{A}_j$ yielding eigenpairs (x, θ) of $\tilde{P}$ from which we obtain pairs $(x, \rho\theta)$ that are the solution of the original problem.

PROPOSITION 3.3. *The polynomial basis $\{\tilde{\phi}_j\}_{j=0}^{d}$ satisfying a three-term recurrence* (2.19) *defined by the sets $\{\tilde{\alpha}\}_{j=0}^{d}$, $\{\tilde{\beta}\}_{j=0}^{d}$ and $\{\tilde{\gamma}\}_{j=0}^{d}$, with $\tilde{\alpha}_j := \alpha_j$, $\tilde{\beta}_j := \frac{\beta_j}{\rho}$, and $\tilde{\gamma}_j := \frac{\gamma_j}{\rho^2}$ satisfy $\phi_j(\lambda) = \tilde{\phi}_j(\theta)\rho^j \quad \forall j \in \mathbb{N}$.*

Proof. To prove the proposition by induction, we first see that it holds for $j = 0, 1$, $\phi_0(\lambda) = 1 = \tilde{\phi}_0(\theta)$ and

$$\phi_1(\lambda) = \alpha_0^{-1}(\lambda - \beta_0) = \alpha_0^{-1}(\rho\theta - \beta_0) = \alpha_0^{-1}(\theta - \frac{\beta_0}{\rho})\rho = \tilde{\phi}_1(\theta)\rho.$$

Now we assume that $\phi_j(\lambda) = \tilde{\phi}_j(\theta)\rho^j \quad \forall j \leq k$, then

$$\begin{aligned}\phi_{k+1}(\lambda) &= \alpha_k^{-1}((\lambda - \beta_k)\,\phi_k(\lambda) - \gamma_k\,\phi_{k-1}(\lambda)) = \\ &= \alpha_k^{-1}\big((\theta - \frac{\beta_k}{\rho})\tilde{\phi}_k(\theta)\rho^k - \frac{\gamma_k}{\rho}\tilde{\phi}_{k-1}\rho^{k-1}\big)\rho = \\ &= \alpha_k^{-1}\big((\theta - \frac{\beta_k}{\rho})\tilde{\phi}_k(\theta) - \frac{\gamma_k}{\rho^2}\tilde{\phi}_{k-1}(\theta)\big)\rho^{k+1} = \tilde{\phi}_{k+1}(\theta)\rho^{k+1}.\end{aligned}$$

□

Using Proposition 3.3, and defining $\tilde{A}_j = \rho^j A_j$, we have

$$P(\lambda) = P(\rho\theta) = \sum_{j=0}^{d} \phi_j(\rho\theta)A_j = \sum_{j=0}^{d} \tilde{\phi}_j(\theta)\rho^j A_j = \tilde{P}(\theta). \tag{3.33}$$

To generate the transformed problem, the matrices A_j are scaled in the same way as in the monomial case. On the other hand, $\tilde{\Phi}_j = \Phi_j$ when using the monomial basis. Thus, provided the scaling factor is supplied, this scaling procedure generalizes for (2.18) the one described for (1.1).

3.7. Extraction. We now discuss how to obtain eigenvectors of the polynomial eigenproblem (2.18) from those of the linearization (2.20). Our polynomial solvers incorporate several strategies for this, all of them being adaptations of those proposed in [9] for polynomial eigenproblems defined in the monomial basis (1.1) in terms of invariant pairs (a generalization of eigenpairs). Here we describe the methods in terms of eigenpairs, although we have implemented also the invariant pair extension, see [10].

Due to the block structure of the eigenvector of the linearization, z (2.21), every block of the computed eigenvector z^i is a candidate to be picked up as the corresponding eigenvector of the polynomial problem, provided $\phi_i(\lambda)$ is not zero. Next we describe the different extraction strategies available in the `PEP` solver.

None No specific strategy is performed. The first block of z is taken.

Residual This strategy picks the block z^i that minimizes the residual norm $\rho(z^i, \lambda)$.

Norm This strategy (the default) picks the ith block for which $|\,\phi_i(\lambda)|$ is maximum.

Structured This strategy computes a linear combination of all the blocks of z, and looks for $x \in \mathbb{C}^n$ that minimizes $\|V_d(x, \lambda) - z\|$, where $V_d(x, \lambda) := \Phi(\lambda) \otimes x$ and $\Phi(\lambda) := \big[1, \phi_1(\lambda), \ldots, \phi_{d-1}(\lambda)\big]^T$. It can be shown that the wanted vector is $x = \sum_{i=0}^{d-1} z^i\overline{\phi_i(\lambda)}/\|\Phi(\lambda)\|_2^2$. In the case of TOAR, this expression simplifies to $x = U\sum_{i=0}^{d-1} g^i\overline{\phi_i(\lambda)}/\|\Phi(\lambda)\|_2^2$, allowing a considerable cost reduction.

3.8. Newton refinement. We remark that we have also implemented iterative refinement for invariant pairs, as described in [20], particularized for polynomial eigenproblems as in [9], but extended to the general non-monomial case. Details of how this is implemented in SLEPc can be found in [10].

```
#define NMAT 5
PEP         pep;           /*  eigensolver context  */
Mat         A[NMAT];       /*  coefficient matrices */
Vec         xr, xi;        /*  eigenvector, x       */
PetscScalar kr, ki;        /*  eigenvalue, k        */
PetscInt    j, nconv;
PetscReal   error;

PEPCreate( PETSC_COMM_WORLD, &pep );
PEPSetOperators( pep, NMAT, A );
PEPSetFromOptions( pep );
PEPSolve( pep );
PEPGetConverged( pep, &nconv );
for (j=0; j<nconv; j++) {
  PEPGetEigenpair( pep, j, &kr, &ki, xr, xi );
  PEPComputeError( pep, j, PEP_ERROR_BACKWARD, &error );
}
PEPDestroy( &pep );
```

FIG. 1. *Example code for basic solution with PEP.*

4. User interface. As mentioned in §2.5, the `PEP` module in SLEPc contains the polynomial eigensolvers presented in this paper. We now briefly discuss the user interface and give some examples of usage.

Figure 1 provides basic code for solving a polynomial eigenproblem. The example shows how the solver context `pep` is created (and destroyed at the end), passing an MPI communicator on which parallel operations will be performed. The problem to be solved is specified by providing the coefficient matrices with `PEPSetOperators` (the code for building the matrices is omitted), and optionally indicating the type of polynomial basis with `PEPSetBasis` (defaults to the monomial basis). Calling `PEPSetFromOptions` allows the user to set up various options through the command line, as illustrated below. The call to `PEPSolve` invokes the actual solver. Then, the solution is retrieved with `PEPGetConverged` and `PEPGetEigenpair`. Note that both the eigenvalue and eigenvector are defined with two variables, so that complex eigenpairs can be returned when operating in real arithmetic; otherwise, `PetscScalar` has been defined as a complex floating point type, so the result is placed in the first variable and the second one is unused.

The eigensolver to be used can be fixed in the code by calling `PEPSetType` with the solver name, e.g., `PEPTOAR`, or alternatively at run time as for instance:

```
$ ./example -pep_type toar -pep_nev 6 -pep_ncv 24 -pep_tol 1e-9
```

The other options indicate that 6 eigenpairs must be computed with 24 column vectors available for the Krylov basis and a tolerance of 10^{-9} for the convergence criterion. All these settings can also be given in code with the corresponding interface function, see the details in the users guide [25] or the online documentation.

The user can specify if the wanted eigenvalues are those of largest (or smallest) magnitude, real part, or imaginary part. Alternatively, sought-after eigenvalues can be defined by the distance to a given target τ or by means of an inclusion (or exclusion) region in the complex plane. In any case, interior eigenvalues are most successfully obtained if the solver is used in combination with the shift-and-invert spectral transformation of §2.3. In `PEP`, spectral transformations are managed also via an `ST` object, similarly to the description in §2.5 for linear eigensolvers. There are two ways of accomplishing the spectral transformation: (i) by explicitly computing the transformed matrices (2.17), or (ii) by letting the linearization-aware eigensolver

handle it implicitly via the block-LU factorization (2.22). The former approach must be explicitly activated with `STSetTransform`. A command line example would be:

```
$ ./ex16 -pep_type toar -pep_target 0 -st_type sinvert
```

The example computes eigenpairs closest to the origin with TOAR and shift-and-invert. The `-st_transform` could be added optionally to switch to `ST` being in charge of the transformation. The same example with Q-Arnoldi would be

```
$ ./ex16 -pep_type qarnoldi -pep_target 0 -st_type sinvert -st_transform
```

where in this case `-st_transform` is required. Similarly, the `PEPLINEAR` solver would run the plain Arnoldi method on the explicit linearization and solve linear systems via the recurrences described in §3.1. The following example uses MUMPS for solving the linear systems associated with the last block, with coefficient matrix $P(\sigma)$:

```
$ ./sleeper -pep_type linear -pep_target 0 -st_type sinvert
            -st_ksp_type preonly -st_pc_type lu
            -st_pc_factor_mat_solver_package mumps -mat_mumps_icntl_14 100
```

There are other configurable settings related to topics discussed in this paper:

- `PEPSetScale` activates parameter or diagonal scaling. In the former case, the user can provide the scaling factor or, alternatively, let SLEPc compute it as explained in §3.6. In the diagonal scaling, it is necessary to provide a value λ representing an approximation to the wanted eigenvalues (in modulus).
- The restarting strategy can be tuned with `PEPTOARSetRestart`, specifying the percentage of basis vectors to be kept at restart, as well as the locking strategy (locking or non-locking) with `PEPTOARSetLocking`.
- `PEPSetExtract` selects one of the extraction strategies described in §3.7.
- `PEPSetRefine` can be used to activate the iterative refinement and specify some settings (details can be found in [10]).

5. Numerical results. The computer system used for the computational experiments in this section is Tirant, an IBM cluster consisting of 512 JS21 blade computing nodes, each of them with two 64-bit PowerPC 970MP dual core processors running at 2.2 GHz with 4 GB of memory, interconnected with a low latency Myrinet network. All executions placed a single MPI process per node. The software consists of SLEPc 3.6 and PETSc 3.6, together with MUMPS 5.0 that is used where indicated, all of them compiled with gcc-4.6.1 and MPICH2.

We have used several test problems to assess the robustness and performance of our polynomial eigensolvers. Results are summarized in Table 1. The first problems arise in the computation of the electronic structure of quantum dots via discretization of the Schrödinger equation [19]. The rest belong to the NLEVP collection [8], all of them polynomial eigenproblems except the last one (loaded_string) which is a rational eigenproblem that we have used to illustrate how general nonlinear eigenproblems can be solved via polynomial interpolation. All computations have been carried out in real arithmetic, except for pdde_stability whose coefficient matrices are complex. Executions use default parameters, that is, locking variants, no scaling, and norm extraction, except planar_waveguide which has been run with parameter scaling and structured extraction. We also remark that pdde_stability required a user-defined sorting criterion in order to compute eigenvalues satisfying $\|\lambda\|_2 = 1$.

The two representative test cases from the quantum dot simulation are: qd_cylinder (cubic polynomial from a cylinder quantum dot discretized with finite differences on a uniform mesh) and qd_pyramid (quintic polynomial from a pyramid quantum dot

TABLE 1
Computational results for the TOAR solver with matrix polynomials of various degrees and matrix dimensions. The executions requested nev *eigenvalues selected from different parts of the spectrum (*which*), using* ncv *columns for the Krylov basis. Results include maximum backward error η_P, number of restarts (*its*) and running time (in seconds) with 16 MPI processes. The tolerance used for the stopping criterion was 10^{-8}.*

name	deg	dim	nev	which	ncv	η_P	its	time
qd_cylinder	3	690,718	10	close to 0.1	40	2×10^{-10}	9	479
qd_pyramid	5	∼12 mill	5	close to 0.4	40	2×10^{-11}	1	1991
sleeper	2	1 mill	40	close to -0.9	80	5×10^{-12}	2	82
pdde_stability	2	250,000	4	$\|\lambda\|_2 = 1$	100	4×10^{-11}	154	582
planar_waveguide	4	50,001	2	smallest real	60	6×10^{-7}	53	56
acoustic_wave_2d	2	999,000	10	close to 0	25	1×10^{-9}	4	65
butterfly	4	90,000	1	close to 0.1	100	3×10^{-9}	183	684
loaded_string	10	1 mill	7	close to 0	40	6×10^{-12}	1	44

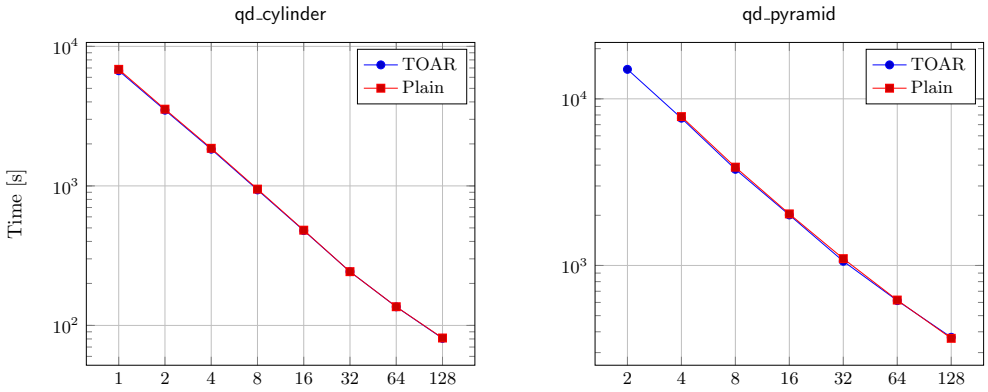

FIG. 2. *Execution times (in seconds) with up to 128 MPI processes for the TOAR and plain Arnoldi (implicit variant) methods with the two test problems arising in the quantum dot simulation: cylinder (left) and pyramid (right). The parameters of the execution are shown in Table 1.*

discretized with finite volumes). These problems are solved with inexact shift-and-invert, in particular with Bi-CGStab with block Jacobi preconditioning (using ILU for the local subdomains). Figure 2 shows parallel execution times of both TOAR and plain Arnoldi for an increasing number of MPI processes. Due to memory requirements, the quintic polynomial problem needs at least 2 processes in TOAR and 4 processes in plain Arnoldi (that uses more memory as discussed later in this section). In both cases, parallel performance is quite good, close to linear scaling. When analyzing in more detail the split times of different steps of the computation, we see that the percentage of time corresponding to linear system solves in qd_pyramid is between 93% and 96% for TOAR and between 90% and 93% for plain Arnoldi (in qd_cylinder, percentages are even higher). Therefore, in these cases scalability of the eigencomputation is about the same as that of linear solves. It also explains why the plots for TOAR and plain Arnoldi match almost exactly, since convergence of both methods is about the same and the relative cost of orthogonalization is practically negligible in this case.

Figure 3 illustrates parallel performance of two problems with different behaviour.

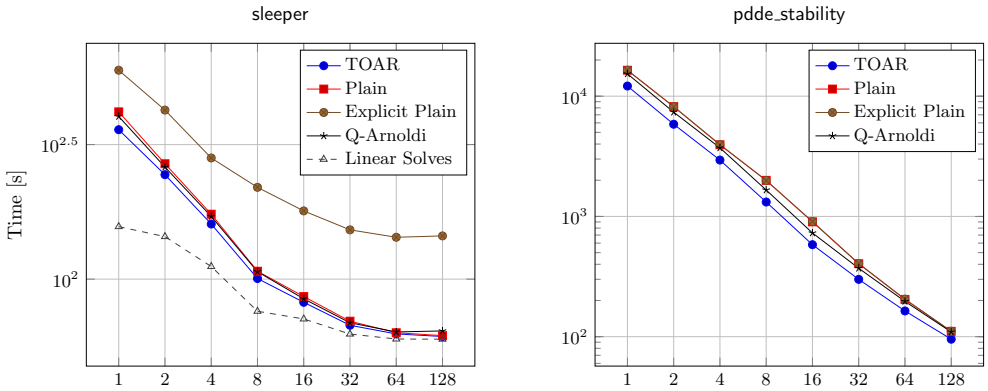

FIG. 3. *Execution times (in seconds) with up to 128 MPI processes for TOAR, Q-Arnoldi and plain Arnoldi (both implicit and explicit variants) with two NLEVP problems:* sleeper *(left) and* pdde_stability *(right). The parameters of the execution are shown in Table 1.*

TABLE 2
Memory consumption (in megabytes) per MPI process (when run with 16 processes) accounting all objects of the program, including Krylov basis vectors, coefficient matrices and factorizations.

	qd_cylinder	qd_pyramid	sleeper	pdde_stability	loaded_string
TOAR	50	1275	100	39	168
Plain Arnoldi	97	2666	184	92	372

The sleeper problem (left plot) uses a parallel direct solver (MUMPS) for the linear systems. The percentage of time for linear solves in TOAR, including symbolic and numeric factorization, is about 44 % when using just one process, but it grows up to as much as 98 % with 128 processes (to stress this point we also plot the time for linear solves in the TOAR run). We remark that probably these times corresponding to MUMPS may be improved by appropriately tuning the parameters, but we have not investigated this. Since linear solves do not scale well in this case, this is hindering the scalability of the whole computation. Now the amount of work associated with orthogonalization is more significant, and hence we can appreciate that TOAR is a bit faster than plain Arnoldi, while Q-Arnoldi is somewhere in between. We have also included in this plot the case in which plain Arnoldi explicitly builds the matrices of the linearization, which is slower since linear solves are then tied to the much larger matrix $A - \sigma B$ of (2.16). In contrast, in the pdde_stability problem (right plot) the coefficient matrix of the linear solves is diagonal and hence this operation is trivial compared to the high cost associated with the orthogonalization of a large Krylov basis (ncv=100). Again, TOAR is faster than Q-Arnoldi and plain Arnoldi. As expected, in this case the explicit matrix variant of plain Arnoldi overlaps exactly with the non-explicit one because matrix B of (2.16) is also diagonal.

Regarding memory consumption, Table 2 compares memory requirements per process for TOAR and plain Arnoldi in several tests problems. In all cases, there is a significant memory savings in TOAR, as expected since the Krylov basis is stored as a set of vectors of length n (plus other smaller matrices) as opposed to length dn in plain Arnoldi. The benefit is more evident with larger polynomial degree, but it also depends on how much memory is used by coefficient matrices and other objects.

Finally, we discuss a test problem that uses a polynomial representation based on

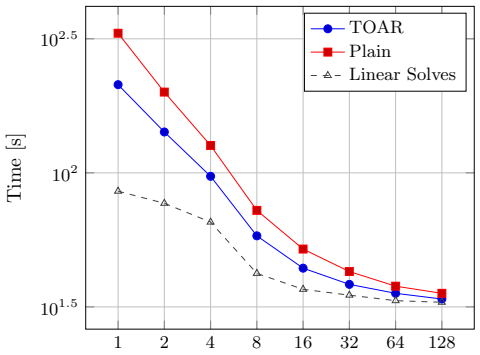

FIG. 4. *Execution times (in seconds) with up to 64 MPI processes for the TOAR and plain Arnoldi (implicit variant) methods with the* loaded_string *problem from the NLEVP collection, solved via interpolation with Chebyshev polynomials of degree 10. The parameters of the execution are shown in Table 1. The dashed line corresponds to the fraction of time in the TOAR run devoted to linear system solves (including factorization).*

non-monomial bases. The loaded_string problem from the NLEVP benchmark collection is a rational eigenvalue problem that can be addressed by means of polynomial interpolation. In this scenario, it is often recommended to use a polynomial basis such as Chebyshev, which enables working with a relatively large polynomial degree without having to worry about possible underflow, provided that the computation is restricted to a certain interval, see [11, 20]. In the case of loaded_string, the interval of interest is $[4, 400]$, where 6 eigenvalues can be found. We have solved this problem with a Chebyshev interpolation polynomial of degree 10, as indicated in Table 1. We remark that the coefficient matrices of the interpolation polynomial are computed automatically by SLEPc via the `NEP` object, but we omit these details here and discuss only the solution of the `PEP`[2]. Figure 4 shows parallel computing times for both TOAR and plain Arnoldi. Since in this case the orthogonalization of basis vectors in plain Arnoldi requires about 10 times more operations than in TOAR, the performance gain in the latter is evident (also in terms of memory savings, see Table 2). In Figure 4 we also include the time invested in the solution of linear systems with MUMPS (including factorization), that is between 40% and 97% of total computation time in TOAR, and between 26% and 93% in plain Arnoldi.

6. Conclusions. SLEPc provides a collection of parallel solvers for the polynomial eigenvalue problem. In this paper we have focused on solvers based on the Arnoldi iteration operating on the linearized eigenproblem, either built explicitly or handled implicitly in various ways. Our flagship eigensolver TOAR is equipped with many features, including support for spectral transformation, in different polynomial bases, restarting with and without locking, scaling and extraction. We have also described implementations of plain Arnoldi and Q-Arnoldi, emphasizing the benefits of the TOAR solver in terms of computational and storage efficiency.

The TOAR method was proposed by other authors, initially for quadratic eigenproblems and then extended to higher degree polynomials. In terms of the method, we have just made minor additions to support more general polynomial bases (not only Chebyshev as in [21]), including shift-and-invert, scaling and extraction for this

[2]While eigenpairs computed by `PEP` have $\eta_P(x,\lambda) < 2 \times 10^{-12}$, the relative error in `NEP`, $\|T(\lambda)x\|/|\lambda|$, is 3×10^{-5}, so more accuracy would require a subsequent iterative refinement phase.

case. Another contribution in terms of algorithms is how to perform effective locking during the restart of TOAR.

From the user's perspective, the solvers are very easy to use, and default parameters are generally good enough for solving many different kinds of problems. For difficult problems, using the command-line interface is very convenient for parameter tuning. In any case, it is always recommended that the user is familiar with the different steps of the computations that are taking place under the hood, and this paper aims at providing the necessary details.

In terms of parallel performance, we tried to demonstrate with the experiments in §5 that the scalability of our solvers is very good provided that the linear solver and preconditioner scales well. PETSc provides many different scalable preconditioners, including algebraic multigrid and domain decomposition. The good parallel performance, together with the numerical robustness of our eigensolvers, opens the door to cope with very large-scale, computationally challenging problems that could not be tackled before.

When designing our solvers, we have chosen the first companion form for the linearization, due to its interesting properties. However, other alternatives are also possible, and as a future work we plan to extend the set of linearizations available in SLEPc in order to broaden the possibilities of preserving spectral properties and having a better conditioning for different polynomial eigenproblems. Also, we remark that SLEPc's `PEP` class is not restricted to Krylov solvers. In particular, the development of a Jacobi-Davidson solver for polynomial eigenproblems is under way.

Acknowledgements. The authors are grateful to Daniel Kressner for insightful comments and suggestions. The computational experiments of §5 were carried out on the supercomputer Tirant at Universitat de València. The matrices from the quantum dot simulation used in §5 were kindly provided by the authors of [19].

REFERENCES

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[2] A. Amiraslani, D. A. Aruliah, and R. M. Corless, *Block LU factors of generalized companion matrix pencils*, Theor. Comput. Sci., 381 (2007), pp. 134–147.

[3] A. Amiraslani, R. M. Corless, and P. Lancaster, *Linearization of matrix polynomials expressed in polynomial bases*, IMA J. Numer. Anal., 29 (2009), pp. 141–157.

[4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[5] 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 (2005), pp. 640–659.

[6] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, K. Rupp, B. Smith, S. Zampini, and H. Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[7] T. Betcke, *Optimal scaling of generalized and polynomial eigenvalue problems*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1320–1338.

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

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

[10] C. Campos and J. E. Roman, *Parallel iterative refinement in polynomial eigenvalue problems*. In preparation, 2015.

[11] C. Effenberger and D. Kressner, *Chebyshev interpolation for nonlinear eigenvalue problems*, BIT, 52 (2012), pp. 933–951.
[12] H. Fan, W. Lin, and P. Van Dooren, *Normwise scaling of second order polynomial matrices*, SIAM J. Matrix Anal. Appl., 26 (2004), pp. 252–256.
[13] I. Gohberg, P. Lancaster, and L. Rodman, *Matrix polynomials*, Academic Press, New York, 1982.
[14] S. Hammarling, C. J. Munro, and F. Tisseur, *An algorithm for the complete solution of quadratic eigenvalue problems*, ACM Trans. Math. Softw., 39 (2013), pp. 18:1–18:19.
[15] V. Hernandez, J. E. Roman, and A. Tomas, *Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement*, Parallel Comput., 33 (2007), pp. 521–540.
[16] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Trans. Math. Softw., 31 (2005), pp. 351–362.
[17] N. J. Higham, R.-C. Li, and F. Tisseur, *Backward error of polynomial eigenproblems solved by linearization*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 1218–1241.
[18] M. Hochbruck and D. Lochel, *A multilevel Jacobi-Davidson method for polynomial PDE eigenvalue problems arising in plasma physics*, SIAM J. Sci. Comput., 32 (2010), pp. 3151–3169.
[19] Feng-Nan Hwang, Zih-Hao Wei, Tsung-Ming Huang, and Weichung Wang, *A parallel additive Schwarz preconditioned Jacobi-Davidson algorithm for polynomial eigenvalue problems in quantum dot simulation*, J. Comput. Phys., 229 (2010), pp. 2932–2947.
[20] D. Kressner, *A block Newton method for nonlinear eigenvalue problems*, Numer. Math., 114 (2009), pp. 355–372.
[21] D. Kressner and J. E. Roman, *Memory-efficient Arnoldi algorithms for linearizations of matrix polynomials in Chebyshev basis*, Numer. Linear Algebra Appl., 21 (2014), pp. 569–588.
[22] D. Lu and Y. Su, *Two-level orthogonal Arnoldi process for the solution of quadratic eigenvalue problems*. Manuscript, 2012.
[23] D. S. Mackey, N. Mackey, C. Mehl, and V. Mehrmann, *Vector spaces of linearizations for matrix polynomials*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 971–1004.
[24] K. Meerbergen, *The Quadratic Arnoldi method for the solution of the quadratic eigenvalue problem*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1463–1482.
[25] J. E. Roman, C. Campos, E. Romero, and A. Tomas, *SLEPc users manual*, Tech. Report DSIC-II/24/02–Revision 3.6, D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2015.
[26] G. W. Stewart, *A Krylov–Schur algorithm for large eigenproblems*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 601–614.
[27] Y. Su, J. Zhang, and Z. Bai, *A compact Arnoldi algorithm for polynomial eigenvalue problems*. talk presented at RANMEP, 2008.
[28] F. Tisseur, *Backward error and condition of polynomial eigenvalue problems*, Linear Algebra Appl., 309 (2000), pp. 339–361.
[29] F. Tisseur and K. Meerbergen, *The quadratic eigenvalue problem*, SIAM Rev., 43 (2001), pp. 235–286.