Parallel matrix-free polynomial preconditioners with application to flow simulations in discrete fracture networks

L. Bergamaschi [a,*], M. Ferronato [a], G. Isotton [b], C. Janna [a], A. Martínez [c]

[a] Department of Civil Environmental and Architectural Engineering (ICEA), University of Padua, Italy
[b] M3E - Mathematical Methods and Models for Engineering, University of Padua, Italy
[c] Department of Mathematics and Geosciences, University of Trieste, Italy

ARTICLE INFO

Keywords:
Polynomial preconditioner
Conjugate gradient method
Discrete fracture network
Parallel computing
Scalability

ABSTRACT

We develop a robust matrix-free, communication avoiding parallel, high-degree polynomial preconditioner for the Conjugate Gradient method for large and sparse symmetric positive definite linear systems. We discuss the selection of a scaling parameter aimed at avoiding unwanted clustering of eigenvalues of the preconditioned matrices at the extrema of the spectrum. We use this preconditioned framework to solve a 3×3 block system arising in the simulation of fluid flow in large-size discrete fractured networks. We apply our polynomial preconditioner to a suitable Schur complement related with this system, which can not be explicitly computed because of its size and density. Numerical results confirm the excellent properties of the proposed preconditioner up to very high polynomial degrees. The parallel implementation achieves satisfactory scalability by taking advantage from the reduced number of scalar products and hence of global communications.

1. Introduction

Discretized PDEs and constrained as well as unconstrained optimization problems often require the repeated solution of large and sparse linear systems $\boldsymbol{Ax} = \boldsymbol{b}$, in which A is symmetric positive definite (SPD). For practical scientific and engineering applications, the use of parallel computers is mandatory, due to the large size and resolution of the considered models. The size of these systems can be of order $10^6 \div 10^9$ and this calls for the use of iterative methods, equipped with ad-hoc preconditioners as accelerators.

When the problem size grows up to several millions of unknowns, it is not possible to store the system matrix nor the preconditioner on a single machine. Furthermore, it is necessary to take advantage of several distributed resources to reduce simulation time and, ultimately, the time to market. Also, in many cases the huge size of the matrices can prevent their complete storage. In these instances only the application of the matrix to a vector is available as a routine (*matrix-free regime*). Differently from direct factorization methods, iterative methods do not need the explicit knowledge of the coefficient matrix, however they need to be suitably preconditioned to produce convergence in a reasonable CPU time. The issue is the construction of a preconditioner $P \approx A^{-1}$ which also works in a matrix-free regime. The most common (general-purpose) preconditioners, such as the incomplete Cholesky factorization or most of approximate inverse preconditioners, rely on the knowledge of the coefficients of the matrix. An exception is represented by the AINV preconditioner ([2]), whose construction is however inherently sequential. In all cases, factorization-based methods are not easily parallelizable, the bottleneck being the solution of triangular systems needed when these preconditioners are applied to a vector inside a Krylov subspace-based solver.

In this paper we are concerned with the effective development of polynomial preconditioners, i.e. preconditioners that can be expressed as $P \equiv p_k(A)$. Polynomial preconditioners are almost ideal candidates to be used as matrix-free parallel preconditioners, since, both in set-up and application, they rely solely on operations, such as the sparse matrix by vector product (SpMV), that are generally provided by highly efficient parallel linear algebra libraries such as PETSc [1], Hypre [17], etc. For instance, the application of $p_k(A)$ requires k matrix-vector products, without needing the explicit knowledge of the coefficients of matrix A. Moreover, their virtual construction requires only the computation of the coefficients of the polynomials, with negligible computational cost, and the eigenvectors of the preconditioned matrix are the same as those

* Corresponding author.
E-mail addresses: luca.bergamaschi@unipd.it (L. Bergamaschi), massimiliano.ferronato@unipd.it (M. Ferronato), g.isotton@m3eweb.it (G. Isotton), carlo.janna@unipd.it (C. Janna), amartinez@units.it (A. Martínez).
URL: https://www.m3eweb.it/ (G. Isotton).

of A. This feature can help accelerating the effect of the polynomial preconditioners by low-rank updates, which take advantage from the (approximate) knowledge of the eigenvectors of PA.

The use of polynomial preconditioners for accelerating Krylov subspace methods is not new. We quote for instance the initial works in [25,31] and [34,27] where polynomial preconditioners are used to accelerate the Conjugate Gradient and the GMRES [33] methods, respectively. However, these ideas have been recently resumed, mainly in the context of nonsymmetric linear systems, e.g. in [28,29] or in the acceleration of the Arnoldi method for eigenproblems [15]. An interesting contribution to this subject is [26] where Chebyshev-based polynomial preconditioners are applied in conjunction with sparse approximate inverses.

In this paper, starting from the work in [6], we develop a modified Newton-Chebyshev polynomial preconditioner for SPD systems, based on the choice of a parameter aimed at avoiding clustering of eigenvalues around the extrema of the spectrum. A theoretical analysis drives the choice of this parameter. This matrix-free preconditioner is employed in the solution of the discrete problem arising from flow simulations in discrete fracture network (DFN) models. DFN models represent only the fractures as intersecting planar polygons, neglecting the surrounding underground rock formation. The explicit representation of the fractures and their properties in a fully 3D structure requires the prescription of continuity constraints for the fluid flow along the linear intersections. The number of the fractures and their different size, that can change of orders of magnitude, entail a complex and multi-scale geometry, which is not trivial to address. The problem has been effectively reformulated as a PDE-constrained optimization problem in [8,10]. The formulation relies on the use of non-conforming discretizations of the single fractures and on the minimization of a functional to couple intersecting planes, with no match between the meshes of the fractures and the traces. The problem, often characterized by a huge size, can be algebraically reduced to the solution of a sequence of SPD systems, whose matrix, however, cannot be computed and stored explicitly. Nevertheless, the granular nature of the problem, which can be inherently subdivided in several local problems on the fractures with a moderate exchange of data, is particularly suitable for a massive parallel implementation.

In this work we will consider the Preconditioned Conjugate Gradient (PCG) method as iterative solver, accelerated by the modified Newton-Chebyshev polynomial preconditioner. For the parallel implementation, we rely on the Chronos library [20,24], a linear algebra package specifically designed for high performance computing. Chronos takes advantage of fine-grained parallelism through the use of openMP directives allowing for the use of multiple threads on the same MPI rank. Thanks also to the reduction of global communication required by the repeated scalar products in PCG, the parallel implementation of polynomial preconditioning turns out to be highly efficient, as it will be shown in the numerical experiments.

The rest of the paper is organized as follows. In Section 2 we briefly review the Newton-Chebyshev polynomial preconditioner and develop a strategy to avoid unpleasant clustering of eigenvalues around the endpoints of the spectrum. In Section 3 we show how to use our polynomial preconditioner in combination with other accelerators. In Section 4 we describe the test case arising from the DFN application, as well as its algebraic formulation after finite element discretization and reduction to an SPD linear system. In Section 5 we describe our parallel implementation, while Section 6 collects the numerical results of the testing. Section 7 provides some concluding remarks.

2. Polynomial preconditioners

Two alternative formulations of the optimal polynomial preconditioners for the Conjugate Gradient method for SPD linear systems are presented in [6], following the work in [30] which established a connection between an accelerated Newton method for the matrix equation $X^{-1} = A$ and the Chebyshev polynomials in the framework of matrix inversion formula. For the sake of completeness, here we shortly derive these two formulations.

2.1. Newton-based preconditioners

The Newton preconditioner can be obtained as a trivial application of the Newton-Raphson method to the scalar equation

$$x^{-1} - a = 0, \quad a \neq 0,$$

which reads

$$x_{j+1} = 2x_j - ax_j^2, \quad j = 0, \dots, \qquad x_0 \text{ fixed.}$$

The matrix counterpart of this method applied to $P^{-1} - A = 0$ can be cast as

$$P_{j+1} = 2P_j - P_j A P_j, \quad j = 0, \dots, \qquad P_0 \text{ fixed,} \tag{1}$$

which is a well-known iterative method for matrix inversion (also known as Hotelling's method [23]).

The efficiency of such a Newton method can however be increased due to the following result, whose elementary proof is in [6]:

Theorem 1. *Let α_j, β_j be the smallest and the largest eigenvalues of $P_j A$. If $0 < \alpha_j < 1 < \beta_j \leq 2 - \alpha_j$ then $[\alpha_{j+1}, \beta_{j+1}] \subset [2\alpha_j - \alpha_j^2, 1]$.*

If $\beta_j = 2 - \alpha_j$ then the reduction in the condition number from $P_j A$ to $P_{j+1} A$ is near 4 provided that α_j is small:

$$\frac{\kappa(P_j A)}{\kappa(P_{j+1} A)} = \frac{2 - \alpha_j}{\alpha_j}(2\alpha_j - \alpha_j^2) = (2 - \alpha_j)^2 \approx 4.$$

Under these hypotheses each Newton step provides an average halving of the CG iterations (and hence of the number of scalar products) as opposed to twice the application of both the coefficient matrix and the initial preconditioner. This idea can be efficiently employed setting e.g. $P_0 = I$ to cheaply obtain a polynomial preconditioner. Other choices of P_0 will be shortly discussed in Section 3.

Exploiting the findings of Theorem 1, in [6] a sequence of scalars $\{\zeta_j\}$ is defined, which modify the recurrence (1) as

$$\hat{P}_{j+1} = \zeta_{j+1}\left(2\hat{P}_j - \hat{P}_j A \hat{P}_j\right), \qquad \hat{P}_0 = \zeta_0 I$$

Application of the polynomial preconditioner to a vector $\boldsymbol{r}$ is described in step 4. of Algorithm 1.

Algorithm 1 Newton-based polynomial preconditioner of degree $2^{\texttt{nlev}-1}$.

1: Approximate the extremal eigenvalues of A: α, β.
2: Set the number of Newton steps: `nlev`
3: Set $\zeta_0 = \dfrac{2}{\alpha + \beta}$, $\zeta_1 = \dfrac{2}{1 + 2\alpha\zeta_0 - (\alpha\zeta_0)^2}$, $\zeta_i = \dfrac{2}{1 + 2\zeta_{i-1} - \zeta_{i-1}^2}$, $i = 2, \texttt{nlev}$.
4: At each CG iteration apply $P_{\texttt{nlev}}$ to the residual vector $\boldsymbol{r}$ through the following recursive procedure:

$$P_0 \boldsymbol{r} = \zeta_0 \boldsymbol{r}$$

$$P_{j+1}\boldsymbol{r} = \zeta_{j+1}\left(2P_j \boldsymbol{r} - P_j A P_j \boldsymbol{r}\right), \qquad j = \texttt{nlev} - 1, \dots, 0$$

2.2. Chebyshev preconditioners

A similar recurrence can be obtained by means of the shifted and scaled Chebyshev polynomial preconditioners. More details can be found in [32,12,6]. We denote, as before, with α and β the smallest and the largest eigenvalue of A, respectively and set

$$\theta = \frac{\beta + \alpha}{2}, \quad \delta = \frac{\beta - \alpha}{2}, \quad \text{and} \quad \sigma = \frac{\theta}{\delta}.$$

The optimal polynomial preconditioner satisfies the following recursion:

$$
\begin{aligned}
&p_{-1}(x) = 0\\
&p_0(x) = \frac{1}{\theta}\\
&p_k(x) = \rho_k\left(2\sigma\left(1-\frac{x}{\theta}\right)p_{k-1}(x) - \rho_{k-1}p_{k-2}(x) + \frac{2}{\delta}\right), \qquad k \geq 1,
\end{aligned}
\tag{2}
$$

with

$$
\rho_k = \frac{1}{2\sigma - \rho_{k-1}},\ k \geq 1 \quad \text{and} \quad \rho_0 = \frac{1}{\sigma}. \tag{3}
$$

The application of the Chebyshev preconditioner of degree m, $P_m = p_m(A)$, to a vector $\boldsymbol{r}$, satisfies a three term recurrence. In fact, defining $\boldsymbol{s}_k = P_k\boldsymbol{r},\ k \geq 0$, using (2) and exploiting the definitions of δ, σ and θ, we have

$$
\begin{aligned}
\boldsymbol{s}_0 &= \frac{1}{\theta}\boldsymbol{r}.\\
\boldsymbol{s}_1 &= \rho_1\left(2\sigma\left(1-\frac{A}{\theta}\right)p_0(A) + \frac{2}{\delta}\right)\boldsymbol{r} = \frac{2\rho_1}{\delta}\left(2\boldsymbol{r} - \frac{A\boldsymbol{r}}{\theta}\right)\\
\boldsymbol{s}_k &= \rho_k\left(2\sigma\left(1-\frac{A}{\theta}\right)p_{k-1}(A)\boldsymbol{r} - \rho_{k-1}p_{k-2}(A)\boldsymbol{r} + \frac{2}{\delta}\boldsymbol{r}\right)\\
&= \rho_k\left(2\sigma\left(1-\frac{A}{\theta}\right)\boldsymbol{s}_{k-1} - \rho_{k-1}\boldsymbol{s}_{k-2} + \frac{2}{\delta}\boldsymbol{r}\right)\\
&= \rho_k\left(2\sigma\boldsymbol{s}_{k-1} - \rho_{k-1}\boldsymbol{s}_{k-2} + \frac{2}{\delta}\left(\boldsymbol{r} - A\boldsymbol{s}_{k-1}\right)\right), \quad k > 1.
\end{aligned}
$$

The practical implementation of $P_m\boldsymbol{r}$ is described in Algorithm 2.

Algorithm 2 Computation of the preconditioned residual $\hat{\boldsymbol{r}} = P_m\boldsymbol{r}$ with Chebyshev preconditioner.

1: Compute $\rho_k, k = 1, \ldots, m_{\max}$ using (3)
2: $\boldsymbol{x}_{old} = \boldsymbol{r}/\theta$ (*if* $m = 0$ *exit with* $\hat{\boldsymbol{r}} = \boldsymbol{x}_{old}$)
3: $\boldsymbol{x} = \frac{2\rho_1}{\delta}\left(2\boldsymbol{r} - \frac{A\boldsymbol{r}}{\theta}\right)$ (*if* $m = 1$ *exit with* $\hat{\boldsymbol{r}} = \boldsymbol{x}$)
4: **for** $k = 2 : m$ **do**
5: $\quad \boldsymbol{z} = \frac{2}{\delta}(\boldsymbol{r} - A\boldsymbol{x})$
6: $\quad \hat{\boldsymbol{r}} = \rho_k\left(2\sigma\boldsymbol{x} - \rho_{k-1}\boldsymbol{x}_{old} + \boldsymbol{z}\right)$
7: $\quad \boldsymbol{x}_{old} = \boldsymbol{x};\ \boldsymbol{x} = \hat{\boldsymbol{r}}$
8: **end for**

2.3. Relation between Newton and Chebyshev polynomials

In [30,6] a relation is established between the two algorithms basically by writing a different recursion involving Chebyshev polynomials taken from the relation

$$
T_{2k}(x) = 2T_k^2(x) - 1. \tag{4}
$$

The Newton-based polynomial preconditioner is then proved equal to the Chebyshev polynomial preconditioner based on the recursion (4). Only, in the Newton case, polynomials in the sequence have degrees $k = 2^j - 1,\ j = 0, \ldots,$ while with the original Chebyshev algorithm every nonnegative integer can be used as the degree of the polynomial.

2.4. Avoiding eigenvalue clustering

A drawback of the polynomial preconditioners is that clustering may arise in the extremal parts of the eigenspectrum of the preconditioned matrix, thus limiting the acceleration of the Conjugate Gradient method. In [6] a modification of the basic algorithms is proposed in order to mitigate such an undesired occurrence. In this Section we analyze more deeply the effect of this modification.

Let us first consider the first step of the original Newton approach. The spectral interval $[\alpha, \beta]$ of A is first scaled by $\frac{2}{\alpha+\beta} = \frac{1}{\theta}$ obtaining $[\hat{\alpha}, \hat{\beta}] = \left[\frac{2\alpha}{\alpha+\beta}, \frac{2\beta}{\alpha+\beta}\right]$. Following the results of Theorem 1 with $f(t) = 2t - t^2$, the spectral interval of P_1A is $[f(\hat{\alpha}), 1]$, with a reduction of the condition number of about 4, as explained in Section 2.1. However, *the extrema of the scaled spectral interval are both mapped onto the left endpoint* $f(\hat{\alpha}) = f(\hat{\beta}) = \frac{4\alpha\beta}{(\alpha+\beta)^2}$ of P_1A thus originating a cluster around the smallest eigenvalue, which is in principle detrimental for the CG convergence.

To avoid this, in [6] a scaling parameter ξ is introduced in order to modify the definition of parameter θ in the Chebyshev/Newton algorithms as

$$
\bar{\theta} = \frac{\beta+\alpha}{2}(1+\xi). \tag{5}
$$

The parameter ξ should be small enough to apply just a slight modification of the native Chebyshev/Newton algorithm. Multiplying the original spectral interval $[\alpha, \beta]$ by $\bar{\theta}^{-1} = \eta\theta^{-1}$ with $\eta = \frac{1}{1+\xi}$, we obtain

$$
[\hat{\alpha}_\eta, \hat{\beta}_\eta] \equiv \left[\frac{2\eta\alpha}{\alpha+\beta}, \frac{2\eta\beta}{\alpha+\beta}\right]
$$

which will be now mapped by the function $f(t)$ onto $[\alpha_\eta^{(1)}, \beta_\eta^{(1)}] := [f(\hat{\alpha}_\eta), 1]$.

Let us denote by $\kappa = \frac{1}{f(\hat{\alpha})}$ and $\kappa_\eta = \frac{1}{f(\hat{\alpha}_\eta)}$ the condition numbers of the preconditioned matrix before and after the modification, respectively. We first prove that modification (5) provides a modest increment of the condition number of the preconditioned matrix at step 1, assuming ξ sufficiently small.

Theorem 2. *Let* $\xi = O(\kappa^{-1})$*, then*

$$
\frac{\kappa_\eta}{\kappa} = 1 + \xi + O(\xi^2).
$$

Proof. First we have that

$$
1 - \eta = \frac{\xi}{1+\xi} = \xi + O(\xi^2), \quad \text{and}
$$

$$
\hat{\alpha}_\eta - \hat{\alpha} = (\eta - 1)\frac{2\alpha}{\alpha+\beta} = (\eta-1)O(\kappa^{-1}) = O(\xi^2)
$$

then

$$
\begin{aligned}
\frac{\kappa_\eta}{\kappa} = \frac{f(\hat{\alpha}_\eta)}{f(\hat{\alpha})} &= \frac{f(\hat{\alpha}) + (\hat{\alpha}_\eta - \hat{\alpha})f'(\hat{\alpha}) - 2(\hat{\alpha}_\eta - \hat{\alpha})^2}{f(\hat{\alpha})}\\
&= 1 + \frac{\frac{2\alpha}{\alpha+\beta}(\eta-1)\left(2 - 2\frac{2\alpha}{\alpha+\beta}\right) + O(\xi^4)}{\frac{4\alpha\beta}{(\alpha+\beta)^2}} =\\
&= 1 + \frac{(\alpha+\beta)^2}{4\alpha\beta}\frac{2\alpha}{\alpha+\beta}(\eta-1)\frac{2(\beta-\alpha)}{\alpha+\beta} + O(\xi^3) =\\
&= 1 + (\eta-1)\frac{\beta-\alpha}{\beta} + O(\xi^3)\\
&= \eta + \frac{1}{\kappa}(1-\eta) + O(\xi^3) = \eta + O(\xi^2) = 1 + \xi + O(\xi^2).
\end{aligned}
$$

Though the condition number κ_η slightly increases with respect to κ, the favorable outcome is that now $f(\hat{\alpha}_\eta) \neq f(\hat{\beta}_\eta)$ with a consequent separation of the smallest eigenvalues. Moreover, a number $k \geq 1$ of the smallest eigenvalues are mapped onto as many of the smallest eigenvalues of the preconditioned matrix. The next theorem states that the k (with $k \geq 1$) smallest eigenvalues of the preconditioned matrix are the map (through the function f) of exactly the k smallest eigenvalues of A. This also means that the largest eigenvalues of A are no longer mapped onto the same smallest eigenvalues of P_1A, as it holds without modification.

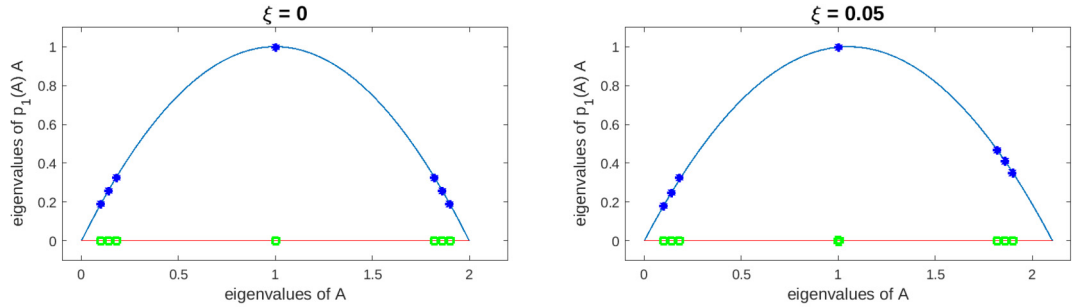

Fig. 1. Eigenvalues of A, green squares on the x-axis, and of $P_1 A$, blue stars on the y-axis. Original algorithm (left), modified algorithm with $\xi = 0.05$ (right).

Table 1
PCG iterations to solve the diagonal problem and a few of the smallest eigenvalue of the preconditioned matrices with a polynomial preconditioner of degree $k = 63$, for different values of the scaling factor ξ. The condition numbers and the partial condition numbers are also provided.

ξ	PCG iters	λ_1	λ_2	λ_5	λ_{10}	$\kappa \equiv \frac{\lambda_{\max}}{\lambda_1}$	$\frac{\lambda_{\max}}{\lambda_{10}}$
0	58	0.03987	0.03987	0.03987	0.03987	25.08	25.83
10^{-6}	57	0.03984	0.04181	0.04181	0.04180	25.10	23.92
10^{-5}	50	0.03961	0.05901	0.05901	0.05901	25.25	16.95
10^{-4}	34	0.03742	0.07388	0.17768	0.21046	26.72	4.75
10^{-3}	39	0.02511	0.04976	0.12096	0.23084	39.82	4.33
10^{-2}	62	0.00898	0.01789	0.04419	0.08664	111.31	11.54

Theorem 3. *Let η be such that $\hat{\alpha}_\eta + 2(1-\eta) < 1$. Denoting by*

$$\hat{\alpha}_\eta = \lambda_1^{(0)} \le \lambda_2^{(0)} \le \ldots \le \lambda_n^{(0)} = \hat{\beta}_\eta, \qquad \textit{and}$$
$$\hat{\alpha}_\eta^{(1)} = \lambda_1^{(1)} \le \lambda_2^{(1)} \le \ldots \le \lambda_n^{(1)} = \hat{\beta}_\eta^{(1)}$$

the eigenvalues of A and $P_1 A$, respectively, and k the integer satisfying $\lambda_k^{(0)} \le \hat{\alpha}_\eta + 2(1-\eta) \le \lambda_{k+1}^{(0)}$ then

$$\lambda_j^{(1)} = f(\lambda_j^{(0)}), \qquad j = 1, \ldots, k.$$

Proof. Since $f(t) = f(2-t),\ \forall t \in \mathbb{R}$ we have

$$f(\hat{\beta}_\eta) = f(2 - \hat{\beta}_\eta) = f\left(2\frac{\alpha + (1-\eta)\beta}{\alpha + \beta}\right) = f(\hat{\alpha}_\eta + 2(1-\eta)).$$

Taking into account that the function f is increasing in $[\hat{\alpha}_\eta, 1]$ and decreasing in $[1, \hat{\beta}_\eta]$ we have

$$f(\lambda_1^{(0)}) \le \ldots \le f(\lambda_k^{(0)}) \le f(\hat{\alpha}_\eta + 2(1-\eta)) \le \max_{j \ge k+1} f(\lambda_j^{(0)}),$$

and the thesis follows.

The situation is depicted in Fig. 1 where the clustering (unclustering) of the extremal eigenvalues is shown for $\xi = 0$ ($\xi = 0.05$). In this example we have $\lambda_1^{(0)} = 0.1,\ \lambda_2^{(0)} = 0.14,\ \lambda_2^{(0)} = 0.18$. All these three eigenvalues are less than $\hat{\alpha}_\eta + 2(1-\eta) \approx 0.195$ and therefore they are mapped onto the leftmost part of the spectrum (blue asterisks, left panel). With $\xi = 0$ the eigenvalues $\lambda_{n-2}^{(0)} = 1.82$, $\lambda_{n-1}^{(0)} = 1.86$, $\lambda_n^{(0)} = 1.9$ are mapped onto the same eigenvalues $\lambda_1^{(1)}$, $\lambda_2^{(1)}$, $\lambda_3^{(0)}$, thus creating a cluster on the leftmost part of the spectrum. By distinction, with $\xi = 0.05$ this is no longer true (blue asterisk, right panel).

Subsequent application of the Newton preconditioner will enhance this behavior, leading to a slight increase of the condition number (compared to the optimal one) at each Newton application, together with a progressive unclustering of the smallest eigenvalues. To experimentally show this behavior we consider the solution of the following linear system $A\boldsymbol{x} = \boldsymbol{b}$ with a random right hand side and a diagonal matrix A of size $n = 10^5$ such that

$$A_{ii} = i, \quad i = 1, \ldots, 10^5,$$

nlev = 6 (polynomial degree = 63), `tol` $= 10^{-10}$.

We obtained the results summarized in Table 1 where we report the extremal eigenvalues of the preconditioned matrices for different values of ξ. In addition to the condition number of $P_{63} A$ we computed a partial condition number, related to the 10th smallest eigenvalue, $\kappa_{10} = \frac{\lambda_{\max}}{\lambda_{10}}$.

Obviously the smallest condition number is provided by the non modified algorithm ($\xi = 0$). If ξ is too small, then no significant effect is observed (second row in the Table). If ξ is too large, the unclustering of the eigenvalues does not pay for the large increasing of the condition number ($\xi = 10^{-2}$ in the Table). The optimal scaling is aimed at separating the smallest eigenvalues and at the same time reducing the partial condition number κ_{10} (see last column in Table 1) which is more informative about PCG convergence, when a few outliers (roughly 10 in this test case) are present [22].

The choice of the parameter ξ is problem dependent. It is related to the degree of the polynomial, to the condition number of the original problem and to the separation of the smallest eigenvalues (to say nothing of the right-hand-side of the system).

However, the findings in Theorem 3 and the experimental results suggest to select $\xi = O(\kappa^{-1})$, but significantly larger than this value to enhance separation of extremal eigenvalues. A choice, e.g., of $\xi \in [10, 10^2]\kappa^{-1}$ will always provide a value close to the optimal one, as confirmed by the numerical experiments.

3. Polynomial acceleration of a given preconditioner

Let us now assume that a (first level) preconditioner is available in factored form as

$$P_{\text{seed}} = W W^T,$$

where P_{seed} can be the square root of the inverse diagonal of A, the inverse of the Cholesky factor $W = L^{-1}$ or the triangular factor of an approximate inverse preconditioner. In such a case the polynomial preconditioner can be applied to the symmetric matrix

$$\hat{A} = W^T A W.$$

Table 2
Results for the matrix `Cube_5317k`. The polynomial preconditioner has been modified with $\xi = 5 \times 10^{-4}$.

	P_{seed} = diagonal preconditioner				P_{seed} = IC preconditioner			
	Polynomial + spectral		Polynomial		Polynomial + spectral		Polynomial	
deg	iter	CPU	iter	CPU	iter	CPU	iter	CPU
0	8597	4481.26	9553	4083.40	1359	1270.75	1853	1476.07
1	4380	4038.17	4865	4066.76	712	1283.02	961	1499.21
3	2210	3829.54	2434	4008.73	370	1219.82	497	1599.90
7	1111	3726.72	1226	4006.06	187	**1208.19**	251	1594.50
15	563	**3716.18**	620	4042.62	97	1240.66	126	1645.59
31	292	3823.50	320	4166.52	51	1294.72	64	1605.11

If the first level preconditioner can be constructed and applied in a matrix-free environment then the whole preconditioner can still be applied in a matrix-free environment.

3.1. Low-rank acceleration

The polynomial preconditioner needs the approximation of the two extremal eigenvalues, which are usually computed together with the corresponding eigenvectors. In general, the availability of a number of the leftmost (approximate) eigenvectors can be exploited to further improve the PCG convergence provided by the polynomial preconditioner.

Let us assume that $\boldsymbol{v}_1, \dots \boldsymbol{v}_p, \boldsymbol{v}_{p+1}, \dots, \boldsymbol{v}_n$ are the eigenvectors of A (or $\hat{A}$), and $\lambda_1 \le \dots \le \lambda_p \le \lambda_{p+1} \le \dots \le \lambda_n$ the corresponding eigenvalues. Defining

$$V = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_p \end{bmatrix}, \qquad \Lambda = \text{diag}(\lambda_1, \dots, \lambda_p),$$

the polynomial preconditioner of degree m, P_0 in this section, computed for A ($\hat{A}$) can be modified to obtain a spectral preconditioner as [11,4]

$$P = P_0 + V(V^T A V)^{-1} V^T.$$

Since $\boldsymbol{v}_j, j = 1, \dots, p$ are also eigenvectors of $P_0 A$, the following properties are easily verified:

$$PA\boldsymbol{v}_j = P_0 A\boldsymbol{v}_j + \boldsymbol{v}_j = (1 + p_m(\lambda_j))\boldsymbol{v}_j, \qquad j = 1, \dots, p \tag{6}$$

$$PA\boldsymbol{v}_j = P_0 A\boldsymbol{v}_j + V(V^T A V)^{-1} \sum_{j=k}^{p} \tilde{\boldsymbol{v}}_k^T \boldsymbol{v}_j \approx p_m(\lambda_j)\boldsymbol{v}_j, \qquad j = p+1, \dots \tag{7}$$

Since Theorem 3 shows that, with the ξ-modification, the polynomial preconditioner matches the smallest eigenvalue of A on the smallest eigenvalue of $P_0 A$, the latter are incremented by one, due to (6), being shifted in the interior of the spectrum with a consequent reduction of the condition number.

3.2. Preliminary numerical results

In this section we present some results in a sequential environment showing the acceleration provided by the polynomial preconditioner applied to a first level preconditioner and modified with low-rank matrices. We consider the solution of a linear system with matrix `Cube_5317k` (available at http://www.dmsa.unipd.it/~janna/Matrices/) arising from the equilibrium of a concrete cube discretized by a regular unstructured tetrahedral grid with size $n = 5\,317\,443$ and nonzeros `nnz` $= 222\,615\,369$.

As the first level preconditioner we considered both the diagonal preconditioner and an incomplete Cholesky factorization with fill-in. In both cases we computed the 10 leftmost eigenpairs to a low accuracy (`tol` $= 10^{-3}$ on the relative residual), by the Deflation-Accelerated Conjugate Gradient, DACG [5], which employs the same preconditioner. We neglect this preprocessing time taking in mind the case in which many linear systems have to be solved with the same coefficient matrix (this is the case e.g. in linear transient problems). The sequential results provided throughout the paper have been obtained with a Matlab code running on an Intel Core(TM) i7-8550U CPU 1.80GHz. The results reported in Table 2 reveal that the combination of polynomial preconditioner and low-rank acceleration can be advantageous.

Considering for example the case with $P_{\text{seed}} = (\text{diag}(A))^{-1}$, the cost of the low-rank modification can be significant when the degree of the polynomial preconditioner is low while the relative influence of this task decreases when the degree grows, since in this case the predominant cost is that of the high number of matrix-vector products.

4. Example of application: discrete fracture network (DFN) flow model

As a relevant example of application of the proposed approach, we consider the DFN flow model developed in [8]. The flow simulation in highly-fractured rock systems is computationally very demanding, because of the complexity of the domain and the uncertainty characterizing the geometrical configuration. In this context, DFN models are usually preferred when the fracture network has a dominant impact on the fluid flow dynamics. They explicitly represent the fractures as intersecting planar polygons and neglect the surrounding rock formation, prescribing continuity constraints for the fluid flow along the fracture intersections, usually called *traces*. Here, we briefly recall the original approach for DFN models introduced in [8] and focus on its discrete algebraic formulation.

Let Ω be a connected three-dimensional fracture network consisting of the union of n_f intersecting planar polygons $\overline{\omega}_i$, $i = 1, \dots, n_f$, where $\overline{\omega}_i = \omega_i \cup \gamma_i$ is the closure of the open planar domain ω_i with its linear boundary γ_i. The fluid flow through ω_i is assumed to be laminar and governed by the standard mass balance equation coupled with Darcy's law, with appropriate essential and natural boundary conditions on γ_i to guarantee the well-posedness of the formulation:

$$-\nabla \cdot (\boldsymbol{K}\nabla h) = q, \qquad \text{in } \omega_i \in \Omega, \tag{8a}$$

$$h_{|\gamma_i^D} = h_i^D, \qquad \text{on } \gamma_i^D, \tag{8b}$$

$$\boldsymbol{K}\nabla h \cdot \vec{n}_i = g_i, \qquad \text{on } \gamma_i^N, \tag{8c}$$

where $\gamma_i^D \cup \gamma_i^N = \gamma_i$, $\gamma_i^D \cap \gamma_i^N = \emptyset$, and $\gamma_i^D \neq \emptyset$. In equations (8), the scalar function h is the hydraulic head, $\boldsymbol{K}$ is the fracture transmissibility tensor, which is assumed to be symmetric and uniformly positive definite, $\vec{n}_i$ is the outward normal to γ_i^N, q is the known discharge within the fracture, and h_i^D and g_i are the given hydraulic head and flux prescribed along the fracture boundary, respectively. Since the fracture network is connected, there is a flux exchange through the linear traces between the intersecting polygons. Let $\sigma_k^{i,j}$ denote the intersection between $\overline{\omega}_i$ and $\overline{\omega}_j$, which we assume to be represented by a single close segment, with Σ the union of the n_s traces, $\Sigma = \cup_{k=1}^{n_s} \sigma_k^{i,j}$. Indicating by h_i the restriction of h to $\overline{\omega}_i$, the continuity of the hydraulic head and the conservation of fluxes across the traces requires that:

$$h_{i|\sigma_k^{i,j}} - h_{j|\sigma_k^{i,j}} = 0, \qquad \forall\, \sigma_k^{i,j} \in \Sigma, \tag{9a}$$

$$[\![\boldsymbol{K}\nabla h_i \cdot \vec{n}_k^i]\!]_{\sigma_k^{i,j}} + [\![\boldsymbol{K}\nabla h_j \cdot \vec{n}_k^j]\!]_{\sigma_k^{i,j}} = 0, \qquad \forall\, \sigma_k^{i,j} \in \Sigma, \tag{9b}$$

with $\vec{n}_k^i$ the outer normal to the trace $\sigma_k^{i,j}$ lying on the fracture $\overline{\omega}_i$ and the symbol $[\![\cdot]\!]_{\sigma_k^{i,j}}$ denoting the jump of the quantity within brackets

through $\sigma_k^{i,j}$. The DFN flow model consists of finding the hydraulic head $h : \Omega \to \mathbb{R}$ satisfying the governing PDEs (8) under the constraints (9).

The numerical solution to the strong form (8)-(9) is re-formulated in [8] as a PDE-constrained optimization problem in weak form. Let us introduce an appropriate measurable function space $\mathcal{H}$ for the representation of h, such as, for instance:

$$\mathcal{H} = \left\{ \eta \in H^1(\omega_i) : \eta_{|\gamma_i^D} = h_i^D, \forall i = 1, \dots, n_f \right\}, \tag{10}$$

with $\mathcal{H}_0$ the corresponding counterpart with homogeneous conditions along γ_i. We use a mixed formulation where the jump $[\![\boldsymbol{K} \nabla h_i \cdot \vec{n}_k^i]\!]_{\sigma_k^{i,j}}$, living along every trace $\sigma_k^{i,j}$ for all i and j, is described by the unknown function $u_i : \sigma_k^{i,j} \to \mathbb{R}$ belonging to the proper measurable function space $\mathcal{U}_i$, which is defined according to the selection of $\mathcal{H}$. For example, for the choice (10), $\mathcal{U}_i$ can be selected as a subspace of $L^2(\sigma_k^{i,j})$, with the global space $\mathcal{U}$ including all $\mathcal{U}_i$. The set of constraints (9) can be prescribed by minimizing the functional $\psi(h,u) : \mathcal{H} \times \mathcal{U} \to \mathbb{R}$:

$$\psi(h,u) = \sum_{\sigma_k^{i,j} \in \Sigma} \left(\left\| h_i - h_j \right\|_{\mathcal{H}}^2 + \left\| u_i + u_j + \alpha \left(h_i - h_j \right) \right\|_{\mathcal{U}}^2 \right), \tag{11}$$

where $\alpha \in \mathbb{R}$ is a regularization parameter. The minimization of $\psi(h,u)$ under the conditions provided by equations (8) is enforced by using Lagrange multipliers. The weak form of (8) reads:

$$(\nabla \eta, \boldsymbol{K} \nabla h)_{\omega_i} - (\eta, u)_{\sigma_k^{i,j}} = -(\eta, q)_{\omega_i} + \left(\eta, g_i \right)_{\gamma_i^N}, \qquad \forall\, \eta \in \mathcal{H}_0,\ i = 1, \dots, n_f, \tag{12}$$

Denoting by $p \in \mathcal{P}$ the Lagrange multipliers living in the appropriate space $\mathcal{P}$, the DFN flow solution is obtained by finding $(h,u,p) \in \mathcal{H} \times \mathcal{U} \times \mathcal{P}$ that minimizes:

$$\Psi(h,u,p) = \psi(h,u) + p \sum_i \left[a_i(\eta,h) - c_i(\eta,u) - q_i(\eta) \right], \qquad \forall\, \eta \in \mathcal{H}_0, \tag{13}$$

with $a_i(\eta,h) = (\nabla \eta, \boldsymbol{K} \nabla h)_{\omega_i}$, $c_i = (\eta,u)_{\sigma_k^{i,j}}$, and $q_i = -(\eta,q)_{\omega_i} + (\eta, g_i)_{\gamma_i^N}$.

4.1. Discrete formulation

The minimization of $\Psi(h,u,p)$ in (13) is carried out approximately by replacing the function spaces $\mathcal{H}$, $\mathcal{U}$ and $\mathcal{P}$ with their discrete counterparts $\mathcal{H}^h$, $\mathcal{U}^h$ and $\mathcal{P}^h$ with finite size n^h, n^u, and n^p, respectively. A relevant advantage of this formulation is that independent computational grids can be introduced for each fracture following the standard finite element method, with no need of enforcing the mesh conformity along the traces.

The discrete counterpart of (13), $\Psi(h^h, u^h, p^h)$, with $(h^h, u^h, p^h) \in \mathcal{H}^h \times \mathcal{U}^h \times \mathcal{P}^h$, is obtained by writing the three variables as linear combinations of the respective basis functions. Denoting with $\boldsymbol{h} = \left[h_1, \dots, h_{n^h} \right]^T$, $\boldsymbol{u} = \left[u_1, \dots, u_{n^u} \right]^T$ and $\boldsymbol{p} = \left[p_1, \dots, p_{n^p} \right]^T$ the vectors collecting the components of these linear combinations we obtain the final expression of the discrete function to be minimized:

$$\Psi(\boldsymbol{h}, \boldsymbol{u}, \boldsymbol{p}) = \begin{bmatrix} \boldsymbol{h} & \boldsymbol{u} \end{bmatrix}^T \begin{bmatrix} G^h & -\alpha B \\ -\alpha B^T & G^u \end{bmatrix} \begin{bmatrix} \boldsymbol{h} \\ \boldsymbol{u} \end{bmatrix} + \boldsymbol{p}^T \left(A\boldsymbol{h} - C\boldsymbol{u} - \boldsymbol{q} \right).$$

The first order optimality conditions yield the following algebraic problem:

$$G^h \boldsymbol{h} - \alpha B \boldsymbol{u} + A \boldsymbol{p} = \boldsymbol{0}, \tag{14a}$$

$$-\alpha B^T \boldsymbol{h} + G^u \boldsymbol{u} - C^T \boldsymbol{p} = \boldsymbol{0}, \tag{14b}$$

$$A\boldsymbol{h} - C\boldsymbol{u} = \boldsymbol{q}, \tag{14c}$$

where α is usually on the order of 1, $\mathbf{h} \in \mathbb{R}^{n^h}$ is the discrete hydraulic head on fractures, $\mathbf{u} \in \mathbb{R}^{n^u}$ is the discrete flux on the traces, and $\mathbf{p} \in \mathbb{R}^{n^p}$ are the discrete Lagrange multipliers. The vector $\mathbf{q} \in \mathbb{R}^{n^h}$ includes the boundary conditions and the forcing terms. Usually, $n^p = n^h$, while according to the problem n^u can be either larger or smaller than n^h. The matrices in (14) are as follows:

- $G^h \in \mathbb{R}^{n^h \times n^h}$ and $G^u \in \mathbb{R}^{n^u \times n^u}$ are symmetric positive semi-definite (SPSD), usually rank-deficient. The matrix G^h is fracture-local, in the sense that it has a block-diagonal structure with the block size depending on each fracture dimension, while G^u has a global nature and operates on degrees of freedom related to different fractures;
- $B, C \in \mathbb{R}^{n^h \times n^u}$ are rectangular coupling blocks, whose entries are given by inner products between the basis functions of $\mathcal{H}^h$ and $\mathcal{U}^h$. The matrix C is fracture-local, with rectangular blocks whose size depends on the dimension of each fracture and the related traces, while $B = C + E$ has a global nature accounted for by the contribution E that has zero entries in the positions corresponding to the rectangular blocks of C;
- $A \in \mathbb{R}^{n^h \times n^h}$ is symmetric positive definite (SPD) and fracture-local, i.e., with a block diagonal structure. Each diagonal block arises from the discretization of the $\nabla \cdot (\boldsymbol{K} \nabla)$ operator over a fracture, hence inherits the usual structure of a 2-D discrete Laplacian.

Equations (14) can be written in a compact form as:

$$\begin{bmatrix} G^h & -\alpha B & A \\ -\alpha B^T & G^u & -C^T \\ A & -C & 0 \end{bmatrix} \begin{bmatrix} \mathbf{h} \\ \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{q} \end{bmatrix} \quad \Longrightarrow \quad \mathcal{K}_0 \mathbf{x} = \mathbf{f_0} \tag{15}$$

where $\mathcal{K}_0$ is a symmetric saddle-point matrix with a rank-deficient leading block. Solution to such problems arises in several applications and is the object of a significant number of works. For a review on methods and ideas, see for instance [3]. With an SPD leading block, as it often arises in Navier-Stokes equations, mixed finite element formulations of flow in porous media, poroelasticity, etc., an optimal preconditioner exists based on the approximation of the matrix Schur complement [14]. However, if the leading block is singular the problem is generally more difficult and the only available result is for the case of maximal rank deficiency [16].

4.2. Algebraic solver

Here we develop a preconditioning framework exploiting the nice properties of matrix A, that is SPD, block diagonal, and such that its inverse can be applied exactly to a vector at a relatively low cost, and the polynomial acceleration. First, an appropriate permutation of $\mathcal{K}_0$ is used:

$$\mathcal{K} = \left[\begin{array}{cc|c} A & 0 & -C \\ G^h & A & -\alpha B \\ \hline -\alpha B^T & -C^T & G^u \end{array}\right], \qquad \mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \mathbf{h} \\ \mathbf{u} \end{bmatrix}, \qquad \mathbf{f} = \begin{bmatrix} \mathbf{q} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \tag{16}$$

so as to avoid a singular leading block. Though the permuted matrix is no longer symmetric, the 2×2 principal submatrix has a block diagonal structure, hence it is, in principle, practically invertible. In a more compact form, the system $\mathcal{K}\mathbf{x} = \mathbf{f}$ can be written as

$$\begin{bmatrix} M & -X \\ -Y^T & G^u \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{0} \end{bmatrix} \tag{17}$$

with

$$M = \begin{bmatrix} A & 0 \\ G^h & A \end{bmatrix}, \ X = \begin{bmatrix} C \\ \alpha B \end{bmatrix}, \ Y = \begin{bmatrix} \alpha B \\ C \end{bmatrix}, \ \mathbf{x}_1 = \begin{bmatrix} \mathbf{p} \\ \mathbf{h} \end{bmatrix}, \ \mathbf{f}_1 = \begin{bmatrix} \mathbf{q} \\ \mathbf{0} \end{bmatrix}.$$

Block Gaussian elimination reduces the system (17) to:

$$\begin{bmatrix} M & -X \\ 0 & G^u - Y^T M^{-1} X \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ Y^T M^{-1} \mathbf{f}_1 \end{bmatrix}$$

$$\text{with} \quad M^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -A^{-1} G^h A^{-1} & A^{-1} \end{bmatrix}$$

whose main computational burden is in the solution of

$$S^u(\alpha)\mathbf{u} = \mathbf{r}, \qquad S^u(\alpha) = G^u - Y^T M^{-1} X, \quad \mathbf{r} = Y^T M^{-1}\mathbf{f}_1. \tag{18}$$

Direct computation easily shows that matrix S^u is symmetric:

$$S^u(\alpha) = G^u - W^T M^{-1} Z = G^u - \alpha B^T A^{-1} C - \alpha C^T A^{-1} B + C^T A^{-1} G^h A^{-1} C. \tag{19}$$

It is also positive definite under realistic conditions. In fact, matrix $S^u(\alpha)$ can always be made SPD by wisely selecting $\alpha > 0$ since $S^u(0)$ is SPD as the sum of the SPSD matrix G^u and the SPD matrix $C^T A^{-1} G^h A^{-1} C$. We assume that this assumption is verified and denote simply by S^u the Schur complement in (18). Therefore, the PCG solver can be employed. Specific theoretical results on the choice of α are not available. However, computational experience shows that an α value on the order of 1 meets the regularizing requirements for the minimization of $\psi(h,u)$ in (11) ensuring that S^u is SPD [8,9].

Explicit computation of S^u is not affordable for realistic problems, while the matrix-free application of S^u to a vector can be implemented with no need of matrix-matrix multiplications. Before starting the PCG iteration, the exact Cholesky factorization of A is computed, i.e., the lower triangular matrix L_A such that $A = L_A L_A^T$. Note that the Cholesky factor L_A preserves the block diagonal structure of A and each diagonal block arises from a 2-D discretization, hence this task is not overly expensive. Then, the application of S^u to a vector $\mathbf{r}$ can be implemented as described in Algorithm 3, whose complexity is: 6 triangular solves + 7 matrix-vector products involving block matrices B, C, G^h and G^u. Once system (18) is solved, the unknowns $\mathbf{h}$ and $\mathbf{p}$ in (16) can be readily recovered by

$$\mathbf{h} = (L_A L_A^T)^{-1}(\mathbf{q} + C\mathbf{u}), \quad \mathbf{p} = (L_A L_A^T)^{-1}\left(\alpha B\mathbf{u} - G^h\mathbf{h}\right).$$

Algorithm 3 Computation of $\mathbf{y} = S^u\mathbf{r}$.

1: $\mathbf{v} = C\mathbf{r}$;
2: $\mathbf{z} = \alpha B\mathbf{r}$;
3: Solve $L_A\mathbf{u} = \mathbf{v}$;
4: Solve $L_A^T\mathbf{t} = \mathbf{u}$;
5: Solve $L_A\mathbf{u} = \mathbf{z}$;
6: Solve $L_A^T\mathbf{w} = \mathbf{u}$;
7: $\mathbf{z} = G^u\mathbf{r} - \alpha B^T\mathbf{t} - C^T\mathbf{w}$;
8: $\mathbf{v} = G^h\mathbf{t}$;
9: Solve $L_A\mathbf{u} = \mathbf{v}$;
10: Solve $L_A^T\mathbf{w} = \mathbf{u}$;
11: $\mathbf{y} = \mathbf{z} + C^T\mathbf{w}$.

A preconditioner for the global system (15) has been proposed in [21] following the basic framework developed in [18] and [19]. The idea is to exploit the fact that the matrix $Z = A^{-1}C = (L_A L_A^T)^{-1}C$ is fracture-local and $B = C + E$, with E having zero entries in the positions of the non-zeros of C and containing the off-block diagonal connections among the fractures. The Schur complement $S^u(\alpha)$ in equation (19) can be re-written as:

$$S^u(\alpha) = G^u + Z^T\left(G^h Z - 2\alpha C\right) - \alpha\left(E^T Z + Z^T E\right) = S_D - S_E, \tag{20}$$

where S_D is block-diagonal and S_E has null diagonal blocks. The preconditioner for S^u proposed in [21] is based on replacing Z in (20) with a sparsified approximation $\widetilde{Z}$ obtained by enforcing an adaptive sparse non-zero pattern at every column $\boldsymbol{z}_i$. This solution can prove effective, but not robust with respect to the set-up parameters used for sparsifying Z. An example of the convergence rate obtained in small- to medium-size problems, taken from [21] by varying the exit tolerance τ for the adaptive sparsification strategy, is shown in Table 3. The smaller is τ, the denser is $\widetilde{Z}$. The iteration count turns out to be very sensitive to the problem-dependent τ value. If $\widetilde{Z}$ is too sparse, the preconditioner for S^u can turn to be indefinite, thus preventing from convergence. In contrast, by decreasing τ a very rapid fill-in of $\widetilde{Z}$ can be experienced. The convergence rate can be very fast, but the fill-in of the resulting preconditioner can be unacceptably large, especially in view of the aim at solving larger problems.

Table 3
Iteration count to convergence and percentage μ of non-zeros retained in $\widetilde{Z}$ with respect to Z with the preconditioner proposed in [21] for different values of the set-up parameter τ. The problems A, B and C have global size equal to 32,549, 86,795 and 205,812, respectively. – means that the solver does not converge because of the indefiniteness of the approximate S^u.

	A		B		C	
τ	iter	μ	iter	μ	iter	μ
0.100	136	12.3%	–	11.5%	–	14.3%
0.050	57	25.1%	1483	54.0%	445	32.0%
0.025	28	40.2%	8	99.5%	128	56.6%
0.001	17	60.8%	4	99.9%	41	81.0%

The fact that the coefficient matrix S^u is not explicitly available makes it attractive the use of a matrix-free preconditioner. In this work, we investigate the application and parallel implementation of the Newton-Chebyshev polynomial preconditioner previously described.

4.3. Preconditioner implementation details

Following the discussion in Section 3, we used as the *seed* preconditioner the diagonal of S^u. Note that $D_S = \mathrm{diag}(S^u)$ can be computed without forming S^u through the steps described in Algorithm 4, where with $\boldsymbol{z}_i, \boldsymbol{t}_i$ we denote the i-th column of matrices Z and T, respectively.

Algorithm 4 Computation of $D_S = \mathrm{diag}(S^u)$.

1: $Z = \left(L_A L_A^T\right)^{-1} C$
2: $T = G^h Z - 2\alpha C$
3: **for** $i = 1 : m$ **do**
4: $\quad (D_S)_i = (G^u)_{ii} + \boldsymbol{z}_i^T \boldsymbol{t}_i$
5: **end for**

The most time-consuming task in Algorithm 4 is represented by the computation of Z which requires sparse matrix inversions. However, it must be observed that these operations involve block matrices and hence do not produce a dramatic increase of the fill-in.

The polynomial preconditioner will be therefore applied to the symmetrically scaled system

$$\hat{S}^u\hat{\boldsymbol{u}} = \hat{\boldsymbol{r}}, \qquad \text{with} \quad \hat{S}^u = \sqrt{D_S^{-1}}\,S^u\sqrt{D_S^{-1}},\ \hat{\boldsymbol{u}} = \sqrt{D_S}\boldsymbol{u},\ \hat{\boldsymbol{r}} = \sqrt{D_S^{-1}}\boldsymbol{r}$$

In the sequel, we will select $S^u = S^u(1)$, i.e. $\alpha = 1$.

5. Parallel implementation

An efficient parallel implementation of the application of the Schur complement S^u and the explicit computation of its diagonal D_S is fundamental for handling large-size problems arising from realistic industrial applications.

The proposed algorithm is implemented relying on the Chronos software package, a collection of linear algebra algorithms designed for high performance computers [20]. Chronos is entirely written in C++ using the potential of object-oriented programming (OOP) to easen its use from other software. The Message Passing Interface (MPI) is used for communications among processes while OpenMP directives enhance the fine-grained parallelism through multithreaded execution. Chronos is free for research purposes and its license can be requested at the library website [20].

The high level of abstraction introduced in Chronos by the OOP allows for the use of the same distributed matrix object to store and use all the sparse matrices composing the block system $\mathcal{K}$ in eq. (16). In

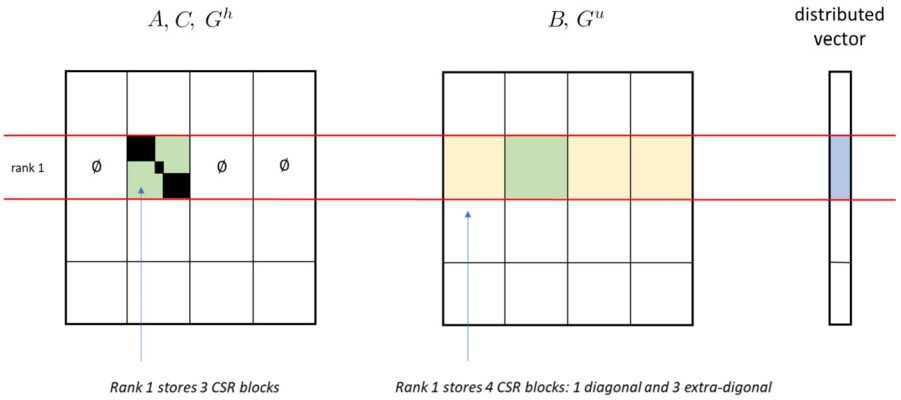

Fig. 2. Chronos DSMat storage schemes for A, C and G^h (left) and B and G^u (center) matrices partitioned into 4 MPI ranks. On the right, a corresponding distributed vector in Chronos. The portions of matrices and vector stored by MPI rank 1 are highlighted with different colors.

Table 4
Size and nonzeros of the relevant matrices for each test case.

Test case	n^u	$n^p \equiv n^h$	$nnz(\mathcal{K})$	$nnz(Z)$	$nnz(S^u)$	# fractures
#1	56375	886693	13 797084	301 879683	62 139981	395
#2	312518	221144	10 854803	59 966125	325 144680	1425
#3 (Frac16)	1 428334	502152	31 802122	–	–	15102
#4 (Frac32)	2 777378	994907	44 646710	–	–	29370

particular, Chronos adopts a Distributed Sparse Matrix (DSMat) storage scheme, where the matrix is sliced into `nprocs` horizontal stripes of consecutive rows, where `nprocs` is the number of MPI ranks involved in the computation. Each stripe is in turn subdivided into blocks stored in Compressed Sparse Row (CSR) format. This block-nested storage scheme, along with nonblocking send/receive messages, enhances the overlap between communications and computations hiding data-transfer latency and reducing wall-time.

For the particular application of DFN, the stripes are chosen taking into account the block-diagonal structure of the matrices A, C, and G^h. Each MPI rank stores a finite number of consecutive blocks and no block is split between different ranks. This subdivision then guides the partitioning of the other matrices B and G^u. A sketch of the DSMat storage scheme for the various blocks of the matrix $\mathcal{K}$ is shown in Fig. 2.

Both the multiplication by S^u and the set-up of D_S require the application of A^{-1}. To this aim, the exact Cholesky factor, L_A, of A is computed by factorizing in parallel all its diagonal blocks: since the number of blocks is very high, within each MPI rank, several OpenMP threads are used to factor a chunk of blocks. The sequential routine *cholmod_factorize*, provided by the SuiteSparse library [13], is used to factorize the single CSR blocks.

The S^u application shown in Algorithm 3 requires Sparse Matrix-by-Vector product (SpMV) calls that are provided by Chronos. At its inner level, SpMV is specifically designed according to the type of matrix. In particular, 10 SpMV products are executed with block-diagonal matrices, 6 of which through forward and backward substitutions performed block-by-block using *cholmod_solve*2 from SuiteSparse. These products do not require any communication between the MPI ranks, and on each rank the operations are executed by multiple OpenMP threads. The remaining three SpMV products, involving B and G^u, require preliminary MPI data transfer: each stripe must receive the components of the distributed vector $\boldsymbol{r}$ that correspond to the column indices of the extra diagonal CSR blocks. To hide the latency, these communications are overlapped to the application of diagonal CSR block with the portion of $\boldsymbol{r}$ owned by the rank, highlighted respectively in green and light blue in Fig. 2.

The computation of D_S is performed in matrix-free setting following Algorithm 4. Once again, the diagonal block structure allows for a highly parallel implementation that does not require communications among MPI ranks. In particular, each group of consecutive entries of D_S, corresponding to the rows of a C^T block, can be computed in parallel using several OpenMP threads.

6. Numerical results on the DFN problem

The relevant sizes and nonzeros of the test matrices are reported in Table 4.

We notice that in the first case $n^u \ll n^h$ (large fractures with few mutual intersections) implying that the intermediate matrix Z has more nonzeros than the final Schur complement S^u, due to its large row size. For this problem it is more convenient to form explicitly S^u and work with the full Schur complement matrix. In the other cases computing the whole Schur complement is not worth due to its size and nonzero number, so the computation of $\mathrm{diag}(S^u)$ and the applications of $\hat{S}^u$ to a vector are implemented as described in Algorithms 3 and 4. The (very high) nonzero number of S^u for test case #2 is reported only to reiterate that this matrix can not be formed explicitly.

6.1. Results on test case #1

The results provided for this example, have been obtained with a Matlab code running on an Intel Core(TM) i7-8550U CPU 1.80GHz.

To roughly estimate the extremal eigenvalues we used the CG-based method called Deflation-Accelerated Conjugate Gradient, DACG [5,7] with low accuracy, namely using a tolerance on the relative residual $\mathtt{tol}_{\mathtt{eig}} = 10^{-3}$. The DACG method is aimed at computing the leftmost eigenpairs of an SPD pencil (A, B) but can be also employed to assess the (reciprocal of the) largest eigenvalues of A when the input matrices are (I, A). The DACG algorithm, implemented in a matrix-free regime (no preconditioning) requiring the application of the coefficient matrix as a function, took 39 iterations for the smallest and 45 iterations for the largest eigenvalue and 6.5 seconds overall.

The results in terms of number of iterations and CPU time are provided in Table 5 for increasing polynomial degree $m = 2^j - 1, j = 0, \ldots, 6$. On the left we show the results of the polynomial preconditioner alone, on the right with a rank-one acceleration, namely using only the leftmost eigenpair, already computed for the polynomial preconditioner setting.

The optimal scaling factor is found to be $\xi = 10^{-3}$ which is in accordance with the theoretical findings as $\kappa(S^u) \approx 1.6 \times 10^4$ and with the

Table 5
Iterations and CPU time to solve $\hat{S}\hat{\boldsymbol{u}} = \hat{\boldsymbol{r}}$ with the polynomial preconditioner for various degrees and $\xi = 10^{-3}$ (left) and for different ξ-values with $m = 31$ and rank-one update (right).

m	iter	MVP	ddot	CPU	iter	MVP	ddot	CPU
	no update				rank-one update			
0	1322	1322	3966	105.59	1235	1235	4900	99.21
1	670	1340	2010	100.95	625	1250	2500	89.77
3	350	1400	1050	104.75	327	1308	1308	94.20
7	177	1416	531	105.66	166	1328	664	95.02
15	90	1440	270	108.09	85	1360	340	97.62
31	48	1536	144	114.61	45	1440	180	103.17
63	28	1792	84	133.49	27	1728	108	123.69

ξ	iter
0	63
10^{-4}	51
10^{-3}	45
3×10^{-3}	49
5×10^{-3}	53
10^{-2}	61

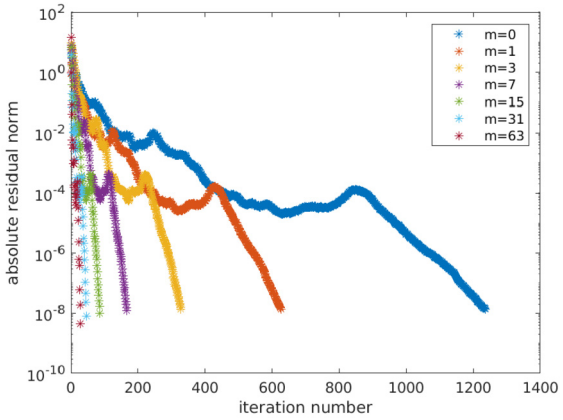

Fig. 3. PCG Convergence profiles for the DFN test case #1 and different values of the polynomial degree. Polynomial preconditioner with rank-one acceleration.

comment at the end of Section 2, in fact we have $\xi \approx 16\kappa^{-1}$. The effect of the polynomial preconditioner is to drastically reduce the scalar products, by slightly increasing the number of matrix-vector products. The low-rank correction, even using one vector only, seems to be convenient, since the additional scalar product per iteration is compensated by a significant reduction of the matrix-vector products. The convergence profile of the PCG solver with different polynomial preconditioners is shown in Fig. 3, where the steepest profiles corresponding to larger degrees can be appreciated.

6.2. Results on test case #2

We use this test case to assess the parallel efficiency of our implementation of polynomial preconditioning. We run the tests on the Marconi100 supercomputer which is installed at CINECA, the Italian supercomputing center. Marconi100 consists of 980 computing nodes each one equipped with 2 x 16 cores IBM Power9 AC922 processors at 2.6 GHz. For completeness, we add that each node can also take advantage of 4 NVIDIA V100 GPU accelerators, but we do not use GPUs in this work. The sparsity pattern of the whole 3×3 block matrix $\mathcal{K}$ is provided in Fig. 4a. Comparing this sparsity pattern with the block structure of $\mathcal{K}$ in equation (16) we can observe that the nonzeros of the coupling matrices B and G^u are spread over the entire block while A, C and G^h display a block diagonal structure. This is better shown in Fig. 4b, 4c where a zoom of matrix A and its exact Cholesky factorization L_A is provided. This nonzero distribution is due to the huge number of connections among the fractures and gives rise to a very dense S^u matrix (see Table 4, 2nd row).

Due to the large size of this problem, we solve it on 4 Marconi100 nodes involving all the available cores for a total of 128 cores. First, we experimentally determine the optimal value of ξ by varying it from 0.001 to 0.01 and keeping fixed the polynomial degree to $m = 127$. Table 6 provides the number of iterations to converge and solution time for PCG along with the minimum and maximum eigenvalues of the diagonally scaled matrix that are needed to set-up the polynomial.

Table 6
Number of iterations to converge and solution times for PCG preconditioned with a polynomial of degree $m = 127$ and 128 Marconi100 cores by varying ξ from 0.001 to 0.01. The minimum and maximum eigenvalues of the diagonally scaled matrix are 1.56×10^{-5} and 2.06, respectively.

ξ	PCG iters	Solv. time [s]
0.001	113	60.393
0.002	107	56.562
0.003	94	50.182
0.004	83	44.074
0.005	108	57.266
0.006	97	51.527
0.007	76	40.509
0.008	78	41.758
0.009	80	42.779
0.010	83	43.959

Table 7
Number of iterations to converge and solution time for PCG preconditioned with polynomials of varying degrees and 128 Marconi100 cores for $\xi = 0.007$.

m	PCG iters	Solv. time [s]
3	2940	48.633
7	1509	50.024
15	670	44.493
31	378	49.962
63	195	51.636
127	76	40.509
255	46	49.445

The choice of the polynomial degree has been made similarly by keeping $\xi = 0.007$ and varying m, again on 128 cores of Marconi100. Table 7, providing the number of iterations to converge and solution time for PCG, shows that the number of iterations always decreases with the degree of the polynomial, as expected, while the time to solution initially decreases but reaches a minimum for $m = 127$.

Finally, we provide a strong scalability test to demonstrate how polynomial preconditioning is amenable to parallelization. Using the optimal values of ξ and m found above, that is 0.007 and 127, respectively, we solve the test case #2 by using 4 Marconi100 nodes and a number of cores per node varying from 1 up to the maximum possible, 32.

In Table 8, and in the following ones, we also report the Set-up time which accounts for the computation of the extremal eigenvalues and the evaluation of the diagonal of the Schur complement matrix S^u, in view of its scaling. We observe that this cost is always less than 5% of the total CPU (Set-up + solution) time. From Table 8, it is possible to note how the number of PCG iterations remains constant, as expected, while the solution times decreases with the increase of the number of

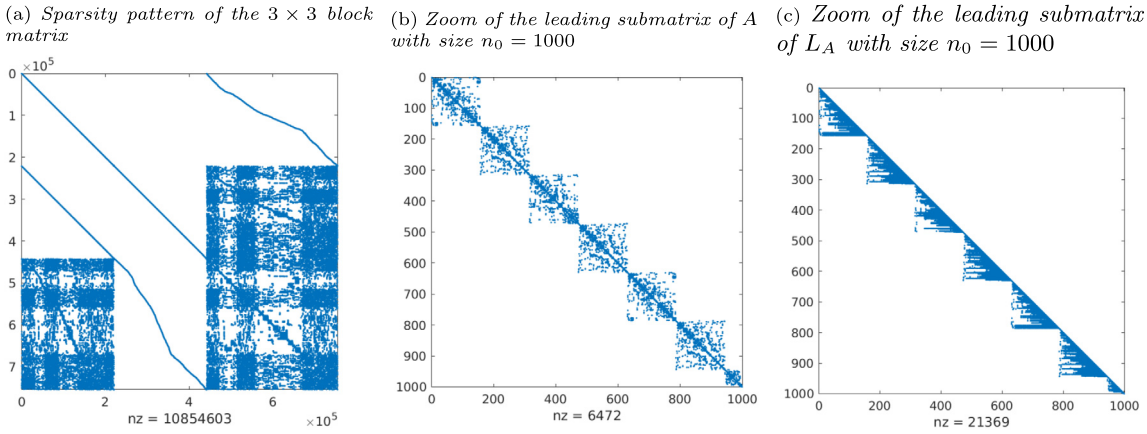

Fig. 4. Sparsity patterns of the whole matrix and subblocks.

Table 8
Number of iterations to converge, Set-up and solution times, and parallel efficiency for PCG preconditioned with a 127-degree polynomial with a varying number of cores.

# of cores	PCG iters	Set-up time [s]	Solv. time [s]	η [%]
4	76	27.7	552.0	100.00
8	76	15.3	304.0	90.80
16	76	8.8	175.3	78.75
32	76	5.5	108.0	63.91
64	76	3.2	63.8	54.05
128	76	2.4	46.8	36.84

cores. To better understand how effective polynomial preconditioning is in parallel, we also report the parallel efficiency which is defined as the ratio between real and ideal speed-up:

$$\eta(\texttt{nprocs}) = \frac{\texttt{nprocs}}{4}\frac{T_{\texttt{nprocs}}}{T_4} \tag{21}$$

where `nprocs` denotes the number of cores used in the run and $T_{\texttt{nprocs}}$ the corresponding execution time. Note that, although with 128 cores the number of unknowns binded to each core is only 2,441, we still have a reasonable efficiency which is very unlikely to reach with more complex preconditioning as approximate inverses, ILU or AMG.

6.3. Results on the largest test cases

This section presents the numerical results on the two largest test cases with a number of fractures of about 16,000 and 32,000, named `Frac16` and `Frac32`, respectively. As done for the other test cases, we first determine the optimal value of ξ, on the basis of the condition number of the (diagonally scaled) Schur complement S^u. As $\kappa^{-1}(\hat{S}^u) = 7.6 \times 10^{-6}$ we vary ξ between 10^{-4} and 5×10^{-3} with a fixed polynomial degree $m = 127$. Table 9 provides the number of iterations for the convergence of the PCG: the optimal value found is 10^{-4}, which is once again roughly one order of magnitude larger than $\kappa^{-1}(\hat{S}^u)$. Moreover, there are no significant differences in the range of $10^{-4} - 10^{-3}$ and the trend appears to be similar as the number of fractures increases.

Regarding the parallel implementation, the two cases `Frac16` and `Frac32` were solved with degree $m = 127$ and $\xi = 0.001$ by increasing the number of cores up to 32. The results are provided in Table 10 and show excellent strong scalability, with an efficiency of about 70% with 32 cores where the number of unknowns binded to each core is only 15,000 and 30,000 for `Frac16` and `Frac32`, respectively.

7. Conclusions

A high-degree polynomial preconditioner has been developed with the aim of reducing the number of scalar products in the Conjugate Gradient iteration. We have shown that the suitable choice of a scaling parameter can speed-up the PCG convergence by avoiding clustering of eigenvalues around the endpoints of the spectral interval. We have given theoretical criteria to select an appropriate value for this parameter. The proposed preconditioning approach reveals particularly useful when the coefficient matrix is not explicitly available, as in the case of the Schur complement matrix obtained in the solution of a 3× 3 block linear system arising in fluid flow simulations on fractured network models. This preconditioner is well suited to parallelization since it reduces considerably the number of scalar product, thus minimizing the collective global communications among processors. Results on the Marconi100 supercomputer show satisfactory scalability results on realistic Discrete Fracture Networks test cases with thousands of fractures.

Data availability

Data will be made available on request.

Acknowledgements

We acknowledge the CINECA award under the ISCRA initiative, for the availability of high performance computing resources and support. LB and AM also acknowledge the support of the "INdAM – GNCS Project", CUP_E53C22001930001.

References

[1] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc web page, https://www.mcs.anl.gov/petsc, 2021.
[2] M. Benzi, J.K. Cullum, M. Tůma, Robust approximate inverse preconditioning for the conjugate gradient method, SIAM J. Sci. Comput. 22 (2000) 1318–1332.
[3] M. Benzi, G.H. Golub, J. Liesen, Numerical solution of saddle point problems, Acta Numer. 14 (2005) 1–137.
[4] L. Bergamaschi, A survey of low-rank updates of preconditioners for sequences of symmetric linear systems, Algorithms 34 (2) (2020).
[5] L. Bergamaschi, G. Gambolati, G. Pini, Asymptotic convergence of conjugate gradient methods for the partial symmetric eigenproblem, Numer. Linear Algebra Appl. 4 (1997) 69–84.
[6] L. Bergamaschi, A. Martinez, Parallel Newton–Chebyshev polynomial preconditioners for the conjugate gradient method, Comput. Math. Methods 3 (2021) e1153.
[7] L. Bergamaschi, M. Putti, Numerical comparison of iterative eigensolvers for large sparse symmetric matrices, Comput. Methods Appl. Mech. Eng. 191 (2002) 5233–5247.
[8] S. Berrone, S. Pieraccini, S. Scialò, A PDE-constrained optimization formulation for discrete fracture network flows, SIAM J. Sci. Comput. 35 (2013) B487–B510.
[9] S. Berrone, S. Pieraccini, S. Scialò, On simulations of discrete fracture network flows with an optimization-based extended finite element method, SIAM J. Sci. Comput. 35 (2013) A908–A935.
[10] S. Berrone, S. Scialò, F. Vicini, Parallel meshing, discretization, and computation of flow in massive discrete fracture networks, SIAM J. Sci. Comput. 41 (2019) C317–C338.

Table 9
Number of iterations for the convergence of the PCG preconditioned with a polynomial of degree $m = 127$ by varying ξ from 10^{-4} to 5×10^{-3}.

Test case	m	ξ	PCG iters	Test case	m	ξ	PCG iters
	127	5×10^{-3}	132		127	5×10^{-3}	156
Frac16	127	1×10^{-3}	104	Frac32	127	1×10^{-3}	121
	127	3×10^{-4}	105		127	3×10^{-4}	112
	127	1×10^{-4}	103		127	1×10^{-4}	107

Table 10
Number of iterations for the convergence, Set-up and solution times, and parallel efficiency of the PCG preconditioned with polynomials of degree $m = 127$ with a varying number of cores.

Test case	# of cores	PCG iters	Set-up time [s]	Solv. time [s]	η[%]
	2	105	83.9	1678.6	100.0
	4	104	43.3	866.6	96.8
Frac16	8	104	23.0	459.6	91.3
	16	103	12.5	249.7	84.0
	32	103	7.9	157.5	66.7
	4	107	87.5	1750.2	100.0
Frac32	8	107	46.3	924.4	94.7
	16	106	25.1	501.6	87.2
	32	108	15.0	300.5	72.8

[11] B. Carpentieri, I.S. Duff, L. Giraud, A class of spectral two-level preconditioners, SIAM J. Sci. Comput. 25 (2003) 749–765 (electronic).
[12] K. Chen, Matrix Preconditioning Techniques and Applications, Cambridge Monographs on Applied and Computational Mathematics, vol. 19, Cambridge University Press, Cambridge, 2005.
[13] Y. Chen, T.A. Davis, W.W. Hager, S. Rajamanickam, Algorithm 887: cholmod, supernodal sparse Cholesky factorization and update/downdate, ACM Trans. Math. Softw. 35 (2008).
[14] H.C. Elman, D.J. Silvester, A.J. Wathen, Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics, 2nd ed., in: Numerical Mathematics and Scientific Computation, Oxford University Press, New York, 2014.
[15] M. Embree, J.A. Loe, R. Morgan, Polynomial preconditioned Arnoldi with stability control, SIAM J. Sci. Comput. 43 (2021) A1–A25.
[16] R. Estrin, C. Greif, On nonsingular saddle-point systems with a maximally rank deficient leading block, SIAM J. Matrix Anal. Appl. 36 (2015) 367–384.
[17] R.D. Falgout, U.M. Yang, Hypre: a library of high performance preconditioners, in: Proceedings of the International Conference on Computational Science-Part III, ICCS '02, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 632–641.
[18] M. Ferronato, A. Franceschini, C. Janna, N. Castelletto, H. Tchelepi, A general preconditioning framework for coupled multi-physics problems with application to contact- and poro-mechanics, J. Comput. Phys. 398 (2019) 108887.
[19] A. Franceschini, N. Castelletto, M. Ferronato, Approximate inverse-based block preconditioners in poroelasticity, Comput. Geosci. 25 (2021) 701–714.
[20] M. Frigo, G. Isotton, C. Janna, Chronos web page, https://www.m3eweb.it/chronos, 2021.
[21] L. Gazzola, M. Ferronato, S. Berrone, S. Pieraccini, S. Scialò, Numerical investifgation on a block preconditioning strategy to improve the computational efficiency of dfn models, in: Book of Extended Abstracts of the 6th ECCOMAS Young Investigator Conference, Valencia, Spain, 2021, pp. 346–354.
[22] A. Greenbaum, Iterative Methods for Solving Linear Systems, SIAM, Philadelphia, PA, 1997.
[23] H. Hotelling, Some new methods in matrix calculation, Ann. Math. Stat. 14 (1943) 1–34.
[24] G. Isotton, M. Frigo, N. Spiezia, C. Janna, Chronos: a general purpose classical AMG solver for high performance computing, SIAM J. Sci. Comput. 43 (2021) C335–C357.
[25] O.G. Johnson, C.A. Micchelli, G. Paul, Polynomial preconditioners for conjugate gradient calculations, SIAM J. Numer. Anal. 20 (1983) 362–376.
[26] I.E. Kaporin, Using Chebyshev polynomials and approximate inverse triangular factorizations for preconditioning the conjugate gradient method, Comput. Math. Math. Phys. 52 (2012) 169–193.
[27] Q. Liu, R.B. Morgan, W. Wilcox, Polynomial preconditioned gmres and gmres-dr, SIAM J. Sci. Comput. 37 (2015) S407–S428.
[28] J.A. Loe, R.B. Morgan, New polynomial preconditioned GMRES, arXiv:1911.07065, 2019.
[29] J.A. Loe, H.K. Thornquist, E.G. Boman, Polynomial preconditioned GMRES in trilinos: practical considerations for high-performance computing, in: Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP), 2020, pp. 35–45.
[30] V. Pan, R. Schreiber, An improved Newton iteration for the generalized inverse of a matrix, with applications, SIAM J. Sci. Stat. Comput. 12 (1991) 1109–1130.
[31] Y. Saad, Practical use of polynomial preconditionings for the conjugate gradient method, SIAM J. Sci. Stat. Comput. 6 (1985) 865–881.
[32] Y. Saad, Iterative Methods for Sparse Linear Systems, second edition, SIAM, Philadelphia, PA, 2003.
[33] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986) 856–869.
[34] M.B. van Gijzen, A polynomial preconditioner for the GMRES algorithm, J. Comput. Appl. Math. 59 (1995) 91–107.