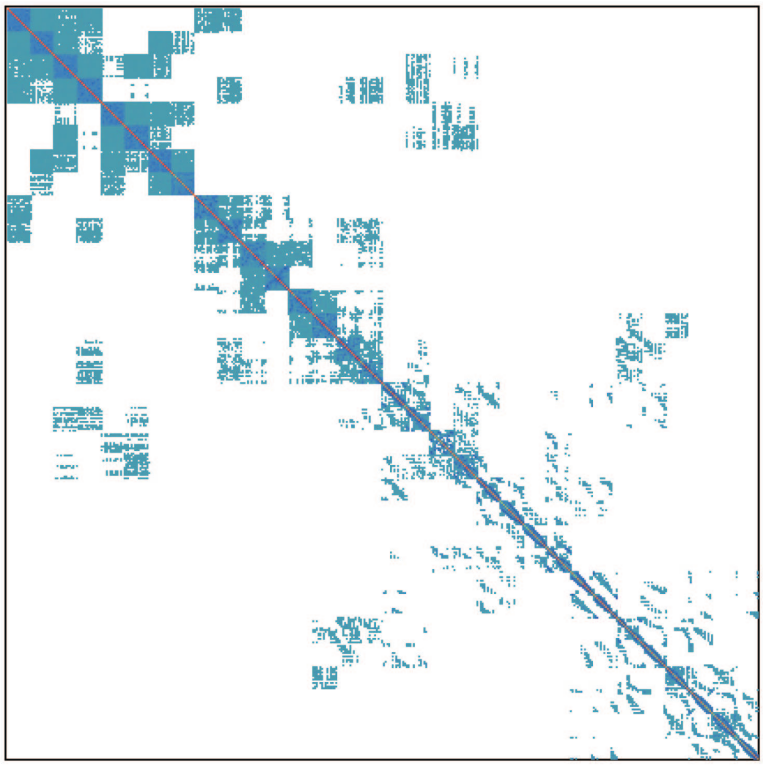
FSAIPACK: User's guide



July, 2013

Contents

1 Introduction

The parallel solution to large sparse linear systems of equations:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$, $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$, is a central task in computational science and engineering. While iterative methods based on Krylov subspaces are well-suited for an effective implementation on High Performance Computing (HPC) platforms, most preconditioning techniques are not. For a recent thorough review, see e.g. [6].

A well-known preconditioner for Symmetric Positive Definite (SPD) linear systems is the Factorized Sparse Approximate Inverse (FSAI). The native FSAI algorithm was originally introduced by [15] about 20 years ago with the aim at computing an explicit sparse approximation of the inverse of the exact lower Cholesky factor of A by means of a Frobenius norm minimization procedure. FSAI is a robust tool endowed with attractive theoretical properties, such as its optimality for a given sparsity pattern with respect to the Kaporin conditioning number of the preconditioned matrix [14]. The aspect that mainly impacts on the FSAI quality and efficiency is the selection of its sparsity pattern. Originally, the algorithm was conceived assuming that a pattern $\mathcal{S}$ for the non-zero entries of the preconditioner is statically set by the user. A typical choice based on the Neumann series expansion of A^{-1} is to use for $\mathcal{S}$ the pattern of a small power of A, i.e., A^2 or A^3. However, even small powers of A can be very dense, thus slowing down the overall algorithm. Possible remedies to such a drawback can be either using the power of a sparsified A [5] or post-filtering the FSAI preconditioner by dropping its smallest entries [16] or both. More recently, also a dynamic pattern computation has been advanced [12] using an algorithm that chooses adaptively the most significant terms to be retained for each row of the preconditioner. Another possibility is to build the preconditioner recursively, so as to compute a dense and high quality FSAI as the product of a few sparse factors [3]. The choice of the most efficient technique for generating the FSAI non-zero pattern is strongly problem-dependent. In most cases, a combination of existing strategies would probably provide the best outcome.

The software package FSAIPACK collects all existing ways for computing a FSAI preconditioner. FSAIPACK allows for combining different strategies to build $\mathcal{S}$, in both a static and dynamic way, thus thoroughly exploiting the preconditioner full potential. The combination of different strategies can be managed by direct library calls or by the FSAIPACK Command Language, that provide the user with a great flexibility and easiness of use. The strategy can be defined by a sequence of commands in ASCII format that can be interpreted by the driver. The commands follow a number of syntax rules that must be complied with for a successful run. FSAIPACK is programmed in Fortran90 using the OpenMP directives for shared memory parallel computations.

2 Theoretical background

The FSAI preconditioner M^{-1} for SPD matrices is defined as the product:

$$M^{-1} = G^T G \tag{2}$$

where G is an approximation of the inverse of the exact lower Cholesky factor L of A. Theoretically, FSAI is computed by minimizing the Frobenius norm:

$$\|I - GL\|_F \tag{3}$$

over all $G \in \mathcal{W}_{\mathcal{S}}$, where $\mathcal{W}_{\mathcal{S}}$ is the set of matrices with the lower triangular non-zero pattern $\mathcal{S}$. In contrast to other inverse ILU techniques [2], the computation of G can be performed without knowing L explicitly. Differentiating (3) with respect to the G entries g_{ij} and setting to zero yields:

$$[GA]_{ij} = [L^T]_{ij} \qquad \forall\ (i,j) \in \mathcal{S} \tag{4}$$

Since L^T is upper triangular while $\mathcal{S}$ is a lower triangular pattern, the matrix equation (4) can be rewritten as:

$$[GA]_{ij} = \begin{cases} 0 & i \neq j, \quad (i,j) \in \mathcal{S} \\ l_{ii} & i = j \end{cases} \tag{5}$$

Since L is unknown, G is not available from (5). Instead, we compute $\widetilde{G}$ such that:

$$[\widetilde{G}A]_{ij} = \delta_{ij} \qquad \forall\ (i,j) \in \mathcal{S} \tag{6}$$

where δ_{ij} is the Kronecker delta. The FSAI factor G is finally found scaling $\widetilde{G}$:

$$G = D\widetilde{G} \tag{7}$$

where D is a diagonal matrix. The choice of D is not unique. In principle, the best choice would be using the inverse of the diagonal entries of L. However, it can be shown that if the diagonal entries of GAG^T are unitary, the Kaporin number of the preconditioned matrix:

$$\kappa = \frac{\frac{1}{n}\mathrm{tr}(GAG^T)}{\det(GAG^T)^{1/n}} \tag{8}$$

is minimum over all matrices belonging to $\mathcal{W}_{\mathcal{S}}$ [13, 14]. Such a condition can be enforced by setting:

$$D = [\mathrm{diag}(\widetilde{G})]^{-1/2} \tag{9}$$

The FSAI preconditioner is very robust, as it exists for any non-zero pattern $\mathcal{S}$ that includes the main diagonal and the preconditioned matrix is SPD by construction. Moreover, the computation of each row of G can be performed independently on the others, thus the resulting setup process exhibits a high inherent parallelism. However, the actual FSAI performance is intimately connected to the selection of $\mathcal{S}$. Quite obviously, the denser the pattern, the more accurate and expensive the preconditioner. In particular, equation (6) shows that the i-th row of $\widetilde{G}$ is computed by solving a dense $m_i \times m_i$ system, with m_i denoting the number of non-zeroes retained in the i-th row. As a consequence, the setup cost turns to be proportional to m_i^3, thus growing up very quickly as m_i increases. The optimal a priori choice of an appropriate non-zero pattern is a difficult task that can be addressed either statically or dynamically, e.g. [9, 5, 10, 12]. The most effective techniques currently available for defining $\mathcal{S}$ are described in the sequel.

ALGORITHM 1: STATIC_FSAI - FSAI computation from a given static non-zero pattern.

Input: Matrix A, the non-zero pattern $\mathcal{S}$
Output: Matrix G
for $i = 1, n$ **do**
  Gather $A[\mathcal{P}_i, \mathcal{P}_i]$ from A ;
  Solve $A[\mathcal{P}_i, \mathcal{P}_i]\, \widetilde{\boldsymbol{g}}_i = \boldsymbol{e}_{m_i}$;
  $\boldsymbol{g}_i = \widetilde{\boldsymbol{g}}_i / \sqrt{\widetilde{g}_{i,m_i}}$;
  Scatter the coefficients in $\boldsymbol{g}_i$ into $G[i, \mathcal{P}_i]$
end

2.1 Static FSAI

Assume that the non-zero pattern $\mathcal{S}$ is given. Denote by $\mathcal{P}_i$ the set of column indices belonging to the i-th row of $\mathcal{S}$:

$$\mathcal{P}_i = \{j : (i,j) \in \mathcal{S}\} \tag{10}$$

and by $A[\mathcal{P}_i, \mathcal{P}_i]$ the submatrix of A collecting the entries a_{kl} such that $k, l \in \mathcal{P}_i$. Let us indicate with $\widetilde{\boldsymbol{g}}_i$ and $\boldsymbol{g}_i$ the dense vectors containing the non-zero coefficients in the i-th row of $\widetilde{G}$ and G, respectively. Using this notation, equation (6) can be re-written as:

$$A[\mathcal{P}_i, \mathcal{P}_i]\widetilde{\boldsymbol{g}}_i = \boldsymbol{e}_{m_i}, \qquad i \in \{1, 2, \ldots, n\} \tag{11}$$

where $\boldsymbol{e}_{m_i}$ is the m_i-th vector of the canonical basis of $\mathbb{R}^{m_i}$. The row $\boldsymbol{g}_i$ of G is obtained by scaling $\widetilde{\boldsymbol{g}}_i$ with the square root of its last entry:

$$\boldsymbol{g}_i = \frac{\widetilde{\boldsymbol{g}}_i}{\sqrt{\widetilde{g}_{i,m_i}}} \tag{12}$$

The procedure for computing a static FSAI given the non-zero pattern $\mathcal{S}$ is summarized in Algorithm 1.

From a computational viewpoint, for a generic row i the most expensive tasks in Algorithm 1 consist of gathering $A[\mathcal{P}_i, \mathcal{P}_i]$ from A and solving the related linear system, with the numerical complexities proportional to m_i^2 and m_i^3, respectively. In contrast, the tasks of scaling $\widetilde{\boldsymbol{g}}_i$ and scattering the coefficients of $\boldsymbol{g}_i$ into G have a linear complexity with m_i, hence their cost is negligible. Let us define the preconditioner density μ_G as the ratio between the number of non-zeroes of G and A:

$$\mu_G = \frac{\mathrm{nnz}(G)}{\mathrm{nnz}(A)} \tag{13}$$

or, equivalently, as the ratio between the average number of non-zeroes of each row of G, $\overline{m}_G$, and A, $\overline{m}_A$:

$$\mu_G = \frac{\overline{m}_G}{\overline{m}_A} \tag{14}$$

For a regularly distributed problem, i.e., $m_i \simeq \overline{m}_G$ for any i, the asymptotic cost c for the preconditioner setup turns out to be proportional to the third power of μ_G:

$$c \propto n \cdot \overline{m}_G^3 = n \cdot \overline{m}_A^3 \mu_G^3 \tag{15}$$

Therefore, the cost for computing FSAI grows very rapidly with increasing the preconditioner density.

ALGORITHM 2: MK_PATTERN - Generation of a static non-zero pattern.

Input: Matrix A, parameters τ, μ_{min}, k, μ_{max}
Output: Non-zero pattern $\mathcal{S}$
repeat
  $\widetilde{A} \leftarrow \text{Drop}(A, \tau)$;
  if $\mu_{\widetilde{A}} < \mu_{min}$ **then** $\tau \leftarrow \tau(\mu_{\widetilde{A}}/\mu_{min})$
until $\mu_{\widetilde{A}} \geq \mu_{min}$;
$B_0 = I$;
for $i = 1, k$ **do**
  $B_i \leftarrow \text{Low}(B_{i-1}\ A)$;
  if $\mu_{B_i} \geq \mu_{max}$ **then** exit the loop
end
Store the pattern of B_k as $\mathcal{S}$;

2.2 Static pattern generation

The problem of generating static non-zero patterns for the FSAI preconditioner has been already discussed by several authors. Traditionally, for approximate inverses based on a Frobenius norm minimization [1, 7] the non-zero pattern of small powers of A is suggested. However, the number of entries in $\mathcal{S}$ grows so quickly with the power of A that only the second or third power can be used in real applications.

With FSAI only a lower triangular pattern is needed, with the upper part of A^k actually useless. In [9, 10] it is shown that better patterns can be obtained with k steps of the recurrence:

$$B_i \leftarrow \text{Low}(B_{i-1}\ A) \qquad i = 1, 2, \ldots, k \tag{16}$$

starting from $B_0 = I$. In equation (16) $\text{Low}(\cdot)$ denotes the lower triangular part of a matrix. The non-zero pattern computed according equation (16) can become quite dense also for relatively small values of k. Hence, a safety parameter μ_{max} is introduced that provides an upper limit to the preconditioner density. Whenever μ_{max} is reached, the procedure (16) is stopped. On the other hand, in order to include in $\mathcal{S}$ also entries belonging to high powers of A pre-filtration of A may be recommended [5]. In the FSAIPACK implementation, a sparsified matrix $\widetilde{A}$ is computed by dropping all the entries a_{ij} of A such that:

$$a_{ij} < \tau \, |a_{ii} a_{jj}| \tag{17}$$

with τ a user-specified real parameter. Setting the optimal value for τ is strongly problem-dependent and may be a difficult task if the coefficients of A lie in a narrow interval. In particular, a large τ value may cause the loss of too many entries. For this reason the parameter μ_{min} that enforces a minimal density for $\widetilde{A}$ is also introduced.

The pre-filtration process desribed above has a linear complexity, hence repeating the procedure a few times does not impact signficantly on the overall set-up cost. Algorithm 2 describes the procedure used to generate $\mathcal{S}$ in FSAIPACK where the function Drop summarizes the pre-filtration process of equation (17).

2.3 Dynamic FSAI computation with adaptive pattern generation

Adaptive procedures have been developed to generate dynamically $\mathcal{S}$ during the FSAI computation with the aim at selecting for each row the "best" positions [8, 10, 12]. In FSAIPACK we use an approach similar to the one proposed in [12] for Block FSAI preconditioning. The use of this algorithm in the context of FSAI is new.

Suppose that an initial guess G_0 for G has already been computed and call $\mathcal{S}_\prime$ its non-zero pattern. The default initial guess is $G_0 = \text{diag}(A)^{-\frac{1}{2}}$ with $\mathcal{S}_\prime$ the diagonal pattern, however any other choice is acceptable. Then, write G_0 as:

$$G_0 = \widetilde{D}_0 \widetilde{G}_0 \tag{18}$$

where $\widetilde{D}_0 = \text{diag}(G_0)$, i.e., the diagonal entries of $\widetilde{G}_0$ are unitary. It can be proved [12] that the entries of $\widetilde{G}_0$ can be directly computed by solving the dense linear systems:

$$A[\overline{\mathcal{P}}_i^0, \overline{\mathcal{P}}_i^0]\widetilde{\boldsymbol{g}}_{0,i} = -A[\overline{\mathcal{P}}_i^0, i], \qquad i \in \{1, 2, \ldots, n\} \tag{19}$$

where $\overline{\mathcal{P}}_i^0 = \mathcal{P}_i^0 \setminus i$, $\mathcal{P}_i^0 = \{j : (i,j) \in \mathcal{S}_\prime\}$, and $\widetilde{\boldsymbol{g}}_{0,i}$ is the vector collecting the non-zero off-diagonal terms in the i-th row of $\widetilde{G}_0$. Consider the Kaporin conditioning number of the preconditoned matrix $G_0 A G_0^T$:

$$\kappa = \frac{\frac{1}{n}\text{tr}(G_0 A G_0^T)}{\det(G_0 A G_0^T)^{\frac{1}{n}}} = \frac{\frac{1}{n}\text{tr}(\widetilde{D}_0 \widetilde{G}_0 A \widetilde{G}_0^T \widetilde{D}_0)}{\det(\widetilde{D}_0 \widetilde{G}_0 A \widetilde{G}_0^T \widetilde{D}_0)^{\frac{1}{n}}} \tag{20}$$

Recalling that $G_0 A G_0^T$ has unitary diagonal entries and $\det(\widetilde{G}_0) = \det(\widetilde{G}_0^T) = 1$, it follows that:

$$\kappa = \frac{1}{\det(A)^{\frac{1}{n}}} \frac{1}{\det(\widetilde{D}_0)^{\frac{2}{n}}} = \frac{\det\left[\text{diag}(\widetilde{G}_0 A \widetilde{G}_0^T)\right]^{\frac{1}{n}}}{\det(A)^{\frac{1}{n}}} \tag{21}$$

Denoting by $\widetilde{G}_0[i,:]$ the i-th row of $\widetilde{G}_0$, the numerator of equation (21) reads:

$$\det\left[\text{diag}(\widetilde{G}_0 \widetilde{A} \widetilde{G}_0^T)\right] = \prod_{i=1}^{n} \widetilde{G}_0[i,:] A \widetilde{G}_0[i,:]^T = \prod_{i=1}^{n} \psi_{0,i} \tag{22}$$

with $\psi_{0,i} = \widetilde{G}_0[i,:] A \widetilde{G}_0[i,:]^T$. Hence, the Kaporin conditioning number of the preconditioned matrix is:

$$\kappa = \det(A)^{-\frac{1}{n}} \prod_{i=1}^{n} \psi_{0,i} \tag{23}$$

The objective is to find an augmented pattern $\mathcal{S}_\infty$ allowing for a reduction of κ in equation (23). The idea is to compute the gradient of κ with respect to $\widetilde{G}_0$ and add in $\mathcal{S}_\infty$ the positions corresponding to its largest components. Notice that each $\psi_{0,i}$ in (23) depends on the i-th row of $\widetilde{G}_0$ only, so the adaptive pattern generation can be efficiently carried out in parallel. Denote by $\nabla\psi_{0,i}$,

the gradient of $\psi_{0,i}$ with respect to the components of $\widetilde{G}_0[i,:]$. Its components read:

$$\frac{\partial \psi_{0,i}}{\partial g_{0,ij}} = 2\left(\sum_{r=1}^{n} a_{jr} g_{0,ir} + a_{ji}\right) \qquad \forall j = 1, \ldots, i-1 \tag{24}$$

with $g_{0,ij}$ the coefficients of $\widetilde{G}_0[i,:]$, i.e., $\nabla\psi_{0,i}$ can be computed at the cost of a sparse matrix by vector product. After computing $\nabla\psi_{0,i}$ using equation (24) the non-zero pattern $\overline{\mathcal{P}}_i^0$ is enlarged by adding the s positions corresponding to the largest entries of $\nabla\psi_{0,i}$, thus obtaining $\overline{\mathcal{P}}_i^1$ and then $\widetilde{\boldsymbol{g}}_{1,i}$ from equation (19). The above procedure can be repeated k times to find all the rows of $\widetilde{G}_2$, $\widetilde{G}_3$, ..., $\widetilde{G}_k$. The current preconditioner quality can be monitored quite inexpensively by computing the value of $\psi_{k,i}$. Therefore it is possible to stop the adaptive procedure for every row whenever a relative exit tolerance is met:

$$\frac{\psi_{k,i}}{\psi_{0,i}} = \frac{\widetilde{G}_k[i,:]A\widetilde{G}_k[i,:]^T}{\widetilde{G}_0[i,:]A\widetilde{G}_0[i,:]^T} \leq \varepsilon \tag{25}$$

with ε a user-specified parameter. To reduce the computational cost of each iteration it is also possible to enforce some dropping by neglecting those entries $g_{k,ij}$ such that:

$$|g_{k,ij}| \leq \tau \|\widetilde{\boldsymbol{g}}_{k,i}\|_2 \qquad j = 1, \ldots, i-1 \tag{26}$$

with τ another user-specified parameter. When either $\psi_{k,i}$ has been reduced below the prescribed tolerance ε or a maximum number of iterations k_{iter} are performed, the adaptive procedure is stopped and each row $\widetilde{G}_k[i,:]$ is properly scaled back to ensure that:

$$G_k[i,:]AG_k[i,:]^T = 1 \tag{27}$$

A summary of the above adaptive procedure as implemented in FSAIPACK is provided in the Algorithm 3.

As to the computational cost, each iteration requires the same operations as Algorithm 1 plus the computation of $\nabla\psi_{k,i}$. If we denote by $m_{k,i}$ the number of non-zeroes of the i-th row of $\widetilde{G}_k$, the cost for computing $\nabla\psi_{k,i}$ is on the average proportional to $\overline{m}_A \cdot m_{k,i}$, while the cost for gathering and solving the dense linear subsystem are proportional to $m_{k,i}^2$ and $m_{k,i}^3$, respectively, as already pointed out previously. After k_{iter} iterations of the above procedure starting from $\widetilde{G}_0 = I$ with $\tau = 0$, the cost c_i for computing the i-th row is:

$$c_i \propto \sum_{k=1}^{k_{iter}} \left[\overline{m}_A \cdot sk + (sk)^2 + (sk)^3\right] = (\alpha_0 \overline{m}_A + \alpha_1)\, k_{iter} + (\alpha_0 \overline{m}_A + \alpha_2)\, k_{iter}^2 + \alpha_3 k_{iter}^3 + \alpha_4 k_{iter}^4 \tag{28}$$

where α_i, $i = 0, \ldots, 4$, are five scalar coefficients depending on s. For k_{iter} large enough, i.e., approximately $k_{iter} \geq \sqrt{\overline{m}_A}$, the overall cost of the adaptive procedure turns out to be proportional to the fourth power of the preconditioner density:

$$c = \sum_{i=1}^{n} c_i \propto n \cdot k_{iter}^4 = n \cdot \overline{m}_A^4 \mu_G^4 \tag{29}$$

The large computational cost has to be compensated for by a faster PCG convergence.

ALGORITHM 3: ADAPT_FSAI - Dynamic FSAI with adaptive pattern generation.

Input: Matrix A, Matrix G_0 (optional), Parameters k_{iter}, s, τ, ε
Output: Matrix G
if G_0 *is present* **then**
  $\widetilde{G}_0 \leftarrow \text{diag}(G_0)^{-1} G_0$;
else
  $\widetilde{G}_0 = I$;
end
for $i = 1, n$ **do**
  for $k = 0, k_{iter} - 1$ **do**
    Compute $\nabla\psi_{k,i}$;
    Compute $\overline{\mathcal{P}}_i^{k+1}$ by adding to $\overline{\mathcal{P}}_i^{k}$ the indices of the s largest components of $\nabla\psi_{k,i}$;
    Gather $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}]$ and $A[\overline{\mathcal{P}}_i^{k+1}, i]$ from A;
    Solve $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}]\widetilde{\boldsymbol{g}}_{k+1,i} = -A[\overline{\mathcal{P}}_i^{k+1}, i]$;
    if $\psi_{k+1,i} \leq \varepsilon\, \psi_{0,i}$ **then** exit the loop over k;
    Drop the entries of $\widetilde{\boldsymbol{g}}_{k+1,i}$ such that $\widetilde{g}_{k+1,ij} \leq \tau \|\widetilde{\boldsymbol{g}}_{k+1,i}\|_2$;
  end
  $\widetilde{d}_{ii} = (-\widetilde{\boldsymbol{g}}_{k+1,i}^T A[\overline{\mathcal{P}}_i^{k+1}, i])^{-\frac{1}{2}}$;
  $\widetilde{\boldsymbol{g}}_{k+1,i} \leftarrow \widetilde{d}_{ii}\widetilde{\boldsymbol{g}}_{k+1,i}$;
  Scatter the coefficients of $\widetilde{\boldsymbol{g}}_{k+1}$ into $G_{k+1}[i, \mathcal{P}_i^{k+1}]$
end

2.4 Fully iterative FSAI construction

The adaptive pattern generation can be computationally expensive, with the most expensive task being the repeated solution to dense linear subsystems with size $m_{k,i}$. The objective is to minimize $\psi_{k,i}$ for each row i, i.e., the quadratic form (22). To do this a steepest descent algorithm can be implemented. Let us define α such that:

$$\psi_{k,i}\left(\widetilde{G}_k[i,:] + \alpha\nabla\psi_{k,i}\right) \rightarrow \min \tag{30}$$

with α reading:

$$\alpha = -\frac{\nabla\psi_{k,i}^T \nabla\psi_{k,i}}{\nabla\psi_{k,i}^T A \nabla\psi_{k,i}} \tag{31}$$

Then, the current i-th row of $\widetilde{G}_k$ is updated as:

$$\widetilde{G}_{k+1}[i,:] = \widetilde{G}_k[i,:] + \alpha\nabla\psi_{k,i}^T \tag{32}$$

Actually, after only a few iterations $\widetilde{G}_k[i,:]$ may be full, so that some dropping is mandatory. In the FSAIPACK implementation a dual dropping strategy is enforced by means of the user-specified parameters τ and m_{max} representing, as usual, the relative threshold tolerance and the maximum number of entries to be retained, respectively [18]. As in Algorithm 3, the iterative procedure for building FSAI is controlled by two user-specified parameters, k_{iter} and ε, defining the maximum number of iterations and the exit tolerance of the steepest descent algorithm, respectively. Because of the dropping, this procedure is actually an incomplete steepest descent algorithm.

ALGORITHM 4: PROJ_FSAI - Fully iterative FSAI computation.

Input: Matrix A, k_{iter}, m_{max}, τ, ε, Matrix G_0 (optional), Matrix M^{-1} (optional)
Output: Matrix G
if G_0 *is present* **then**
    $\widetilde{G}_0 \leftarrow \mathrm{diag}(G_0)^{-1} G_0$;
else
    $\widetilde{G}_0 \leftarrow I$;
end
if M^{-1} *is not present* **then** $M^{-1} \leftarrow I$;
for $i = 1, n$ **do**
    for $k = 0, k_{iter} - 1$ **do**
        Compute $\nabla\psi_{k,i}$;
        $\boldsymbol{p} = M^{-1}\nabla\psi_{k,i}$;
        $\alpha = -\nabla\psi_{k,i}^T \boldsymbol{p} / \boldsymbol{p}^T A \boldsymbol{p}$;
        $\widetilde{G}_{k+1}[i,:] = \widetilde{G}_k[i,:] + \alpha\nabla\psi_{k,i}$;
        Select the m_{max} largest entries of $\widetilde{G}_{k+1}[i,:]$ larger than $\tau\|\widetilde{G}_{k+1}[i,:]\|_2$;
        if $\psi_{k+1,i} \leq \varepsilon\, \psi_{0,i}$ **then** exit the loop over k;
    end
    $\widetilde{d}_{ii} = (\widetilde{G}_k[i,:] A \widetilde{G}_k[i,:]^T)^{-\frac{1}{2}}$;
    $G_k[i,:] = \widetilde{d}_{ii}\widetilde{G}_k[i,:]$;
end

The steepest descent convergence can be accelerated by using an inner preconditioner M^{-1}, e.g., a previously computed FSAI, at the expense of an additional computational cost. The fully iterative FSAI computation procedure is provided in Algorithm 4, that is reminiscent of the idea proposed in [4] where a self-preconditioned descent-type method is used to compute a non-factored approximate inverse.

All iterations in Algorithm 4 have approximately the same computational cost because a fixed number of entries are retained for each row. The more expensive operations are the computation of $\nabla\psi_{k,i}$ and α, both requiring a sparse matrix by a sparse vector multiplication. At most, the computational complexity of these tasks is proportional to $\overline{m}_A \cdot m_{max}$ and $\overline{m}_{M^{-1}} \cdot \overline{m}_A \cdot m_{max}$, where $\overline{m}_{M^{-1}}$ is the average number of non-zeroes per row of M^{-1}. The overall cost for the fully iterative construction of the i-th row is therefore:

$$c_i \propto k_{iter} \cdot (1 + \overline{m}_{M^{-1}}) \cdot \overline{m}_A \cdot m_{max} \tag{33}$$

with the total cost estimated as:

$$c \propto n \cdot k_{iter} \cdot (1 + \overline{m}_{M^{-1}}) \cdot \overline{m}_A \cdot m_{max} = n \cdot k_{iter} \cdot (1 + \overline{m}_{M^{-1}}) \cdot \overline{m}_A^2 \cdot \mu_G \tag{34}$$

Hence, the cost for computing G with Algorithm 4 is approximately linear with the density μ_G, with the proportionality constant depending linearly on k_{iter} and $\overline{m}_{M^{-1}}$. For such a reason, the use of a preconditioned iteration may have a strong impact on the overall cost, and should be used only when a quite sparse and effective M^{-1} is available.

ALGORITHM 5: POST_FILT - FSAI post-filtration.

Input: Matrix A, Matrix G, Parameters τ, m_{max}
Output: Matrix G
for $i = 1, n$ **do**
  Select the m_{max} largest entries of $\boldsymbol{g}_i$ such that $|g_{ij}| \geq \tau \|\boldsymbol{g}_i\|_2$;
  Store the selected entries in $\widetilde{\boldsymbol{g}}_i$ and the discarded entries in $\boldsymbol{\varepsilon}_i$;
  Create the set of discarded column indices $\mathcal{E}_\rangle$;
  Gather $A[\mathcal{E}_\rangle, \mathcal{E}_\rangle]$ from A;
  $\widetilde{d}_{ii} = \left(1 + \boldsymbol{\varepsilon}_i^T A[\mathcal{E}_\rangle, \mathcal{E}_\rangle]\boldsymbol{\varepsilon}_\rangle\right)^{-1/2}$;
  $\widetilde{\boldsymbol{g}}_i \leftarrow \widetilde{d}_{ii}\widetilde{\boldsymbol{g}}_i$;
  Scatter the coefficients of $\widetilde{\boldsymbol{g}}_i$ into $G[i, \mathcal{P}_i \setminus \mathcal{E}_\rangle]$;
end

2.5 FSAI post-filtration

Experience suggests that FSAI may have several small entries, independently of the strategy used to generate $\mathcal{S}$. In order to make the FSAI application cheaper with no important detrimental effects on the PCG convergence rate an a posteriori sparsification of G can be performed. This technique, introduced in [16], is called post-filtration and proceeds as follows. Assume that G has already been computed. The off-diagonal non-zero entries of the i-th row, stored in the dense vector $\boldsymbol{g}_i$, are processed using a dual threshold strategy similar to the one suggested in [18] in the context of ILU factorizations. Two user-specified parameters m_{max} and τ are needed, such that only the largest m_{max} off-diagonal entries of $\boldsymbol{g}_i$ are retained with an absolute value larger than $\tau\|\boldsymbol{g}_i\|_2$. Each vector $\boldsymbol{g}_i$ is therefore decomposed as:

$$\boldsymbol{g}_i = \widetilde{\boldsymbol{g}}_i + \boldsymbol{\varepsilon}_i \qquad i = 1, \ldots, n \tag{35}$$

where $\widetilde{\boldsymbol{g}}_i$ and $\boldsymbol{\varepsilon}_i$ collect the sparsified row and the discarded entries, respectively. All vectors $\widetilde{\boldsymbol{g}}_i$, $i = 1, \ldots, n$, form the matrix $\widetilde{G}$ which however no longer satisfies the condition that the preconditioned matrix $\widetilde{G}A\widetilde{G}^T$ has unitary diagonal entries. This condition helps accelerate the PCG convergence [21], hence to restore it the actual post-filtered FSAI factor G is defined by scaling $\widetilde{G}$ with the aid of the diagonal matrix $\widetilde{D}$:

$$G = \widetilde{D}\widetilde{G} \tag{36}$$

Let us define the set $\mathcal{E}_\rangle$ collecting the column indices of the discarded entries $\boldsymbol{\varepsilon}_i$:

$$\mathcal{E}_\rangle = \left\{ | : \}_{\rangle|} \in \boldsymbol{\varepsilon}_\rangle \right\} \tag{37}$$

Then the coefficients of $\widetilde{D}$ can be computed as:

$$\widetilde{d}_{ii} = \left(1 + \boldsymbol{\varepsilon}_i^T A[\mathcal{E}_\rangle, \mathcal{E}_\rangle]\boldsymbol{\varepsilon}_\rangle\right)^{-1/2} \tag{38}$$

The post-filtration procedure is summarized in Algorithm 5.

As to the computational cost, filtering a row with m_i non-zeroes and selecting the largest entries have a linear and superlinear complexity, respectively, while the cost for the computation of $\widetilde{d}_{ii}$ is proportional to m_i^2 as it requires the

ALGORITHM 6: REC_FSAI - Recurrent FSAI computation.

Input: Matrix A, n_ℓ
Output: Matrices G_1, G_2, ..., G_{n_ℓ}
$A_0 = A$;
$G_0 = I$;
for $k = 0, n_\ell - 1$ **do**
    $A_{k+1} = G_k A_k G_k^T$;
    Compute G_{k+1} as the FSAI factor of A_{k+1};
end

gathering of a submatrix from A. Hence, the average post-filtration cost c can be estimated as:

$$c \propto n \cdot \overline{m}_G^2 = n \cdot \overline{m}_A^2 \cdot \mu_G^2 \tag{39}$$

and is therefore usually negligible relative to the cost of computing G. The post-filtration procedure generally produce sparse FSAI factors with no significant loss in the convergence rate.

2.6 Recurrent FSAI computation

Another way to reduce the FSAI cost though retaining a large number of non-zero entries is to compute G as the product:

$$G = \prod_{k=1}^{n_\ell} G_k \tag{40}$$

where G_k, $k = 1, \ldots, n_\ell$, is the k-th level FSAI. The final G implicitly inherits a large number of non-zeroes even though each G_k is rather sparse. This idea, first introduced in [16] and later developed by [22, 3], looks similar to multilevel preconditioners, e.g., [20, 11].

The k-th term G_k is computed as the FSAI factor of A_k:

$$A_k = G_{k-1} A_{k-1} G_{k-1}^T \tag{41}$$

by any of the previous techniques. The procedure starts with $G_0 = I$ and $A_0 = A$, and proceeds through n_ℓ levels. A summary is provided in Algorithm 6.

Quite obviously, the matrix G in (40) is never formed explicitly and its density is generally much larger than the sum of the G_k densities. Hence, the sum of the costs for computing each factor G_k is generally smaller than the cost for computing the final G in just one step. The overall cost of the recurrent FSAI computation depends on both the strategy used for G_k at each level and the sparse matrix-matrix product $A_{k+1} = G_k A_k G_k^T$. The full computation of A_{k+1} is not feasible, hence a dropping is implemented using again the dual strategy defined by m_{max} and τ, and preserving the symmetry of A_{k+1}.

A sketch of the procedure used to compute $B = GAG^T$ is provided in Algorithm 7. The cost c_i for computing the i-th row of B with Algorithm 7 is proportional to:

$$c_i \propto m_{G,i} \cdot \overline{m}_A + m_{max} \cdot \overline{m}_{G,i} \tag{42}$$

ALGORITHM 7: PREC_MAT - Computation of a FSAI preconditioned matrix.

Input: Matrix A, Matrix G, m_{max}, τ
Output: Matrix $B = GAG^T$
for $i = 1, n$ **do**
  $\boldsymbol{v} = AG[i,:]^T$;
  Select the m_{max} largest entries of $\boldsymbol{v}$ such that $|v_j| \geq \tau \|\boldsymbol{v}\|_2$, $j = 1, 2, \ldots, n$;
  $\boldsymbol{w} = G\boldsymbol{v}$;
  Select the m_{max} largest entries of $\boldsymbol{w}$ such that $|w_j| \geq \tau \|\boldsymbol{w}\|_2$, $j = i, i+1, \ldots, n$;
  Scatter $\boldsymbol{w}$ into $B[i,:]$;
end
Transpose the strict upper part of B into its strict lower part;

with the overall average cost:

$$c \propto n \cdot \overline{m}_G \cdot (\overline{m}_A + m_{max}) = n \cdot \overline{m}_A \cdot (\overline{m}_A + m_{max}) \cdot \mu_G \tag{43}$$

3 Package decription

The FSAIPACK software package is a collection of functions and subroutines that implement the Algorithm 1 through 7 for shared memory parallel computers with the Fortran90 programming language and OpenMP directives for multi-thread execution [17]. As all the algorithms exhibit a high degree of parallelism, an MPI implementation for distributed memory computers is not theoretically difficult and is currently under development.

A number of derived data types and classes are defined and can be accessed at the user level with the aim at making the FSAIPACK use easier (Table 1):

- **Pattern**: a data structure used to store the pattern of a matrix in the compressed sparse row storage mode [19]. **Pattern** includes two member integer variables, NROWS and NTERM, with the size and the number of non-zeroes of the matrix, respectively, and two member integer arrays, IAT and JA, storing the pointer to the beginning of each row and the column indices of the non-zeroes of the matrix, respectively;

- **CSRMAT**: a derived data type that extends the **Pattern** type so as to include also the coefficients of the matrix. **CSRMAT** includes as members a **Pattern** variable PATT and a double precision array COEF collecting the matrix coefficients in row-major order;

- **FSAIP_Prec**: a derived data type containing two lists of pointers to **CSRMAT** type variables, LEFT and RIGHT. The definition of this data type has been selected to store a FSAI preconditioner in the most general form (40). LEFT and RIGHT are the list in ascending and descending order of the G_k and G_k^T factors, respectively. The **FSAIP_Prec** type variable is accessed by a function that includes new factors in the LEFT and RIGHT lists, and by a routine that applies the preconditioner to a vector.

The derived data types are supplemented with the respective constructors and destructors along with other specific functions.

Derived Data Type	Member Variables	Member Functions
Pattern	NROWS, NTERM, IAT, JA	*new_Pattern*, *dlt_Pattern*
CSRMAT	PATT, COEF	*new_CSRMAT*, *dlt_CSRMAT*, *new_CSRCOEF*, *dlt_CSRCOEF*, *copy_CSRMAT*
FSAIP_Prec	LEFT, RIGHT	*append_LEFT*, *append_RIGHT*, *delete_LEFT*, *delete_RIGHT*, *apply_FSAIP_Prec*, *dlt_FSAIP_Prec*

Table 1: Summary of the user-accessible derived data types

The other functions accessible to the user are (Table 2):

- *MK_patt*: creates a pattern for the static FSAI computation by Algorithm 2;
- *CPT_static_FSAI*: computes the coefficients of a static FSAI by Algorithm 1;
- *CPT_adaptive_FSAI*: computes a FSAI factor with adaptive pattern generation of Algorithm 3;
- *CPT_projection_FSAI*: computes a fully iterative FSAI by Algorithm 4;
- *CPT_preconditioned_Matix*: computes the sparse-sparse product of three matrices of Algorithm 7;
- *FILTER_FSAI*: post-filters a FSAI factor by Algorithm 5;
- *TRANSPOSE_FSAI*: transposes a FSAI factor.

The member functions *APPEND_LEFT* and *APPEND_RIGHT*, that append a FSAI factor and its transposed to the lists of a **FSAI_Prec** variable, are indirectly accessible by using a specific interpreted command, as it will be shown later. All functions return a **CSRMAT**-type variable, except *MK_patt*, that returns a **Pattern** type variable, and *APPEND_LEFT* and *APPEND_RIGHT*, that return a **FSAIP_Prec** variable.

4 FSAIPACK Command Language

FSAIPACK allows for the different algorithms to be combined in order to optimize the preconditioner performance. For instance, an initial pattern created by the *MK_patt* function can be improved by a few steps performed with either *CPT_adaptive_FSAI* or *CPT_projection_FSAI*, potentially obtaining a better preconditioner than that computed using a static or a dynamic technique

Function	Input parameters	Output parameters
MK_patt	CSRMAT: A integer: k double: τ, μ_{min}, μ_{max} Pattern: $patt_{in}$ (optional)	Pattern: $patt_{out}$
CPT_static_FSAI	CSRMAT: A Pattern: $patt_{in}$	CSRMAT: G
CPT_adaptive_FSAI	CSRMAT: A integer: k_{iter}, s double: τ, ε CSRMAT: G_0 (optional)	CSRMAT: G
CPT_projection_FSAI	CSRMAT: A integer: k_{iter}, m_{max} double: τ, ε CSRMAT: G CSRMAT: G_p, G_p^T (optional)	CSRMAT: G
CPT_preconditioned_Matrix	CSRMAT: A integer: m_{max} double: τ CSRMAT: G	CSRMAT: B
FILTER_FSAI	CSRMAT: A integer: m_{max} double: τ CSRMAT: G	CSRMAT: G
TRANSPOSE_FSAI	CSRMAT: G	CSRMAT: G^T
APPEND_LEFT	CSRMAT: G	FSAI_Prec: $FSAI$
APPEND_RIGHT	CSRMAT: G^T	FSAI_Prec: $FSAI$

Table 2: Summary of the functions directly accessible at the user level with the Input and Output parameters

Type	Character
Comment	#
Command	>
Data	Any number

Table 3: First non-blank character of each line type.

only. To improve the easiness of use of FSAIPACK, a command language is defined that allows for the implementation of a user-specified strategy to compute FSAI. The commands are stored as an array of character variables of maximal length 100. Each element of such array represents a single line of the strategy and must comply with a set of syntax rules. The strategy array is used as input for the FSAIPACK driver with the Command Language interpreter provided as an open source code.

4.1 Syntax rules

Strategy lines belong to three types:

1. comment type,
2. command type,
3. data type.

The type of each line is identified by the first non-blank character, according to Table 3.

The objects handled by the strategy are CSRMAT type variables identified by a user-specified string of characters, except the final preconditioner that is a FSAIP_Prec type variable. The CSRMAT objects are managed in a fully dynamic way by the command language, allocating and deallocating automatically the user-specified data structures whenever needed. Mandatory object identifiers are:

1. `A`: system matrix identifier,
2. `PREC`: final preconditioner identifier.

`A` and `PREC` must be used at least once in each strategy. The maximum length of the string identifying each object is 11 characters.

A comment line can be introduced at any point of a strategy. FSAIPACK Command Language discards any character located after the comment type identifier `#`. A comment can be also introduced at the end of a command or a data type line.

A command line is a string with at most 100 characters with the syntax:

> *keyword* `[` *input* `,` *input* `:` *output* `]` `-`*character* `-`*character* `-`*...*

The *keyword* is the identifier of a function implemented in the FSAIPACK library. The symbols `[` and `]` include the identifiers of the CSRMAT objects used as *input* and *output* of the function identified by the *keyword*. A function can have more than one input objects separated by the symbol `,`. All functions

have only one output object. The symbol `:` is the separator between input and output objects. All functions depend on a set of user-specified parameters. If the user does not specifies a parameter value, a default is automatically assumed. The parameters set by the user are identified by the symbol `-` followed by a *character* associated to the parameter. Each command type line must be followed by a number of data type lines equal to the number of parameters set by the user. Each data type line contains a numerical value that is assigned to a parameter according to the order of the parameters declared in the command type line. Blank characters can be inserted wherever in a command or data type line and are discarded by the interpreter.

The available keywords, required input and output objects, user-specified parameters and default values are summarized in Table 4. The symbols A and B denote CSRMAT type variables, *patt* a non-zero pattern included in a CSRMAT type variable, G a CSRMAT type variables containing a FSAI preconditioner factor, and $FSAI$ the FSAIP_Prec type variable with the final preconditioner, which is always denoted by `PREC` in our syntax.

4.2 Examples

This is an example strategy:

```
# This is an example strategy
> MK_PATTERN [A:patt] -k -t # Two out of four parameters are set
2 # Number of steps of power recurrence
0.05 # Pre-filtration tolerance
> STATIC_FSAI [A,patt:G]
# The static FSAI is improved dynamically
> ADAPT_FSAI [A:G] -n -e # G is an Input/Output object
10 # Number of adaptive steps
1.e-3 # Exit tolerance
# Post-filtering a FSAI factor is generally recommended
> POST_FILT [A:G] # Default parameters are used
# It is necessary to transpose G and append the left and right factors
> TRANSP_FSAI [G:Gt]
> APPEND_FSAI [G,Gt:PREC] # PREC is a mandatory identifier
```

The strategy above performs the following operations:

- the system matrix is used to build the static pattern `patt` by `k`=2 steps of the power recurrence and `t`=0.05 as pre-filtration tolerance
- the system matrix and the pattern `patt` are used to compute statically the FSAI factor `G`
- the factor `G` is improved dynamically by `n`=10 steps of the adaptive procedure with exit criterion `e`=10^{-3}
- the factor `G` is post-filtered according to the entries of the system matrix
- the factor `G` is transposed into `Gt`
- the factors `G` and `Gt` are appended in the final preconditioner

Function, Keyword		Input/Output		Parameters, identifier, default		

MK_Patt	`MK_PATTERN`	In:
		In/Out:
			A
			patt
				τ
				k
				μ_{min}
				μ_{max}
					`t`
					`k`
					`m`
					`M`
						0.05
						3
						0.20
						5.00

CPT_static_FSAI	`STATIC_FSAI`	In: Out:	A, *patt* G			

CPT_adaptive_FSAI	`ADAPT_FSAI`	In:
		In/Out:
			A
			G
				k_{iter}
				s
				τ
				ε
					`n`
					`s`
					`t`
					`e`
						30
						1
						0.00
						1e-3

CPT_projection_FSAI	`PROJ_FSAI`	In:
		Opt. in:
		In/Out:
			A
			G_p, G_p^T
			G
				k_{iter}
				m_{max}
				τ
				ε
					`n`
					`s`
					`t`
					`e`
						10
						10
						0.00
						1e-8

CPT_preconditioned_Matrix	`PREC_MAT`	In: Out:	A, G, G^T B	m_{max} τ	`n` `t`	n 0.00
FILTER_FSAI	`POST_FILT`	In: In/Out:	A G	m_{max} τ	`n` `t`	n 0.05
TRANSPOSE_FSAI	`TRANSP_FSAI`	In: Out:	G G^T			
APPEND_LEFT and *APPEND_RIGHT*	`APPEND_FSAI`	In: In/Out:	G, G^T $FSAI$			

Table 4: Keywords, input/output objects, parameter identifiers and defualt values.

Recall that the `APPEND_FSAI` keyword is mandatory at the end of any strategy.

References

[1] Benson, M. W. 1973. Iterative solution of large scale linear systems. M.Sc. Thesis, Lakehead University, Thunder Bay (ON), 1973.

[2] Benzi, M. and Tůma, M. 1999. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics 30*, 305-340.

[3] Bergamaschi, L. and Martinez, A. 2012. Banded target matrices and recursive FSAI for parallel preconditioning. *Numerical Algorithms 61*, 223–241.

[4] Chow, E.i and Saad, Y. 1998. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput. 19*, 995–1023.

[5] Chow, E. 2000. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput. 21*, 1804–1822.

[6] Ferronato, M. 2012. Preconditioning for sparse linear systems at the dawn of the 21st century: history, current developments, and future perspectives. *ISRN Applied Mathematics 2012*, Article ID 127647, doi: 10.5402/2012/127647

[7] Frederickson, P. O. 1975. Fast approximate inversion of large sparse linear systems. Mathematical Report 7, Lakehead University, Thunder Bay (ON), 1975.

[8] Grote, M. and Huckle, T. 1997. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.* **18**, 838–853.

[9] Huckle, T. 1999. Approximate sparsity patterns for the inverse of a matrix and preconditioning. *Appl. Num. Math.* **30**, 291–303.

[10] Huckle, T. 2003. Factorized sparse approximate inverses for preconditioning. *J. Supercomput.* **25**, 109–117.

[11] Janna, C., Ferronato, M., and Gambolati, G. 2009. Multilevel incomplete factorizations for the iterative solution of non-linear FE problems. *Int. J. Num. Meth. Eng. 80*, 651–670.

[12] Janna, C. and Ferronato, M. 2011. Adaptive pattern research for Block FSAI preconditioning. *SIAM J. Sci. Comput. 33*, 3357–3380.

[13] Kaporin, I. E. 1990. A preconditioned conjugate gradient method for solving discrete analogs of differential problems *Diff. Equat. 26*, 897–906.

[14] Kaporin, I. E. 1994. New convergence results and preconditioning strategies for the conjugate gradient method. *Num. Lin. Alg. Appl. 1*, 179–210.

[15] Kolotilina, L. Yu. and Yeremin, A. Yu. 1993. Factorized sparse approximate inverse preconditioning. I. Theory. *SIAM J. Mat. Anal. Appl. 14*, 45–58.

[16] Kolotilina, L. Yu. and Yeremin, A. Yu. 1999. Factorized sparse approximate inverse preconditioning. IV. Simple approaches to rising efficiency. *Numerical Linear Algebra with Applications* **6**, 515–531.

[17] OpenMP website. http://www.openmp.org.

[18] Saad, Y. 1994. ILUT: a dual threshold incomplete ILU factorization. *Num. Lin. Alg. Appl. 1*, 387–402.

[19] Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems* (2nd edn.). SIAM, Philadelphia (PA).

[20] Saad, Y. and Suchomel, B. 2002. ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Num. Lin. Alg. Appl. 9*, 359–378.

[21] Van Der Sluis, A. 1969. Condition numbers and equilibration of matrices. *Numer. Math.* **14**, 14–23.

[22] Wang, K. and Zhang, J. 2003. MSP: a class of parallel multistep successive sparse approximate inverse preconditioning strategies. *SIAM J. Sci. Comput. 24*, 1141–1156.