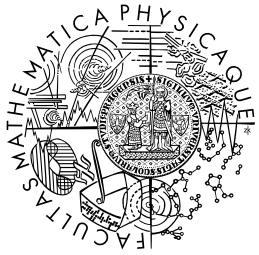

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Outrata

Approximations by low-rank matrices and their applications

Department of Numerical Mathematics

Supervisor of the master thesis:	prof. Ing. Miroslav Tůma, CSc.
Study programme:	Mathematics
Study branch:	Numerical and Computational Mathematics

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank to my supervisor, who provided me with every support, inspiration and guidance one could possibly ask for. I would like to express equal gratitude to all of my family and friends who supported me throughout my studies.

Title: Approximations by low-rank matrices and their applications

Author: Michal Outrata

Department: Department of Numerical Mathematics

Supervisor: prof. Ing. Miroslav Tůma, CSc., Department of Numerical Mathematics

Abstract: Consider the problem of solving a large system of linear algebraic equations, using the Krylov subspace methods. In order to find the solution efficiently, the system often needs to be preconditioned, i.e., transformed prior to the iterative scheme. A feature of the system that often enables fast solution with efficient preconditioners is the *structural sparsity* of the corresponding matrix. A recent development brought another and a slightly different phenomenon called the *data sparsity*. In contrast to the classical (structural) sparsity, the data sparsity refers to an uneven distribution of extractable information inside the matrix. In practice, the data sparsity of a matrix typically means that its blocks can be successfully approximated by matrices of low rank. Naturally, this may significantly change the character of the numerical computations involving the matrix. The thesis focuses on finding ways to construct Cholesky-based preconditioners for the conjugate gradient method to solve systems with symmetric and positive definite matrices, exploiting a combination of the data and structural sparsity.

Methods to exploit the data sparsity are evolving very fast, influencing not only iterative solvers but direct solvers as well. Hierarchical schemes based on the data sparsity concepts can be derived both from applications or algebraically (see, e.g., Hackbusch, 1999; Enquist and Ying, 2011). As for the Cholesky factorization as a subtask of such schemes, while the general trend seems to move towards recurrently-based approaches, possibly combined with nested-dissection reorderings that introduce incompleteness via low-rank approximations (see, e.g., Grasedyck, Kriemann and Le Borne 2008 or Kriemann and Le Borne 2014). The focus in this thesis is on the column-oriented approach that allows for combination of the two mentioned types of approximations. On one hand, one can approximate factor blocks using low-rank approximations. On the other hand classical concepts of incomplete factorizations can be used. But in order to put these ideas into practice, new algorithmic approaches have to be introduced. Our starting point is the classical sparse incomplete column-oriented Cholesky factorization that is significantly modified in the above spirit.

Keywords: low-rank matrix approximations, sparse matrices, iterative methods for solving large linear algebraic equations, preconditioning

Contents

1. Introduction

1.1 Basic notation and preliminaries

Let us first recall some of the basic notions, mostly in order to unify terminology and notation. The work follows the commonly used notation that can be found in, e.g., the textbook of Demmel [22] or in the clasical book of Golub and van Loan [40].

The whole work will assume the real n-dimensional Euclidean space $\mathbb{R}^n$. Vectors will be denoted by small letters, e.g., $x \in \mathbb{R}^n$. If not stated otherwise, $\|x\|_2 \equiv \|x\|$ is to the Euclidean norm of the vector x. Linear operators from $\mathbb{R}^n$ to $\mathbb{R}^m$ will be represented by m-by-n real matrices, denoted by capital letters. The identity operator from $\mathbb{R}^n$ to $\mathbb{R}^n$ will be denoted I_n or simply I. Having an m-by-n matrix A, one often uses either its Frobenius norm $\|A\|_F$ or the spectral norm (2-matrix-norm) denoted by $\|A\|_2$ or simply $\|A\|$. The condition number of a matrix with respect to some given norm $\|\cdot\|$ is denoted by $\kappa(A)$ and defined as $\kappa(A) = \|A\|\|A^{-1}\|$. If not stated otherwise, the norm is again considered to be the spectral norm as above. The j-th column of A will be denoted by $A_{*,j}$ (i-th row by $A_{i,*}$); its entry at the position (i, j) is denoted by a_{ij}. Furthermore, the square j-by-j matrix formed by the first j rows of the first j columns of A is called the *j-th leading principal submatrix*. If another submatrix of A is considered, the classical MATLAB notation will be used, e.g., $A_{1:j,1:j}$ is the j-th leading principal submatrix and $A_{i:n,j}$ is composed from the entries of the j-th column with row indices i and higher.

There are several important classes of matrices that should be mentioned here. First, square matrix A is called *symmetric*, provided $A^T = A$, where A^T stands for the *transposed matrix to A*. Any square matrix U is called *unitary*, provided $U^T U = UU^T = I$. Special subclass of unitary matrices are *permutation matrices*, i.e., matrices that have in each column and row only one nonzero element equal to one. A square matrix A is called *symmetric, positive-definite* (SPD), provided that $A^T = A$ and satisfies $x^T Ax > 0$ for any $x \neq 0$.

The number of linearly independent columns of A is the *rank of A*, denoted by $\mathrm{rank}(A)$. The linear hull of the columns of A is called the *image of A* or the *range of A* and is denoted by $\mathcal{R}(A)$. The set of the preimages of the zero vector with respect to the matrix A is called the *null space of A* or the *kernel of A* and is denoted by $\mathcal{N}(A)$. If $\mathcal{N}(A) = \{0\}$, A is said to be *regular* or *nonsingular*, otherwise A is said to be *singular* or *rank-deficient*. If all of the leading principal submatrices of A are regular, then A is called *strongly regular*.

It is often the case that a given matrix is nonsingular but its columns are *almost linearly dependent*. In practice, this often means that one has to work with the matrix as though it is rank-deficient. This leads to the notion of the *numerical rank of A*, which captures the *amount of information* present in A. To be more specific, if a matrix A is very accurately approximated by a matrix of exact rank p, then the numerical rank of A is said to be p. It is, for a small $\varepsilon > 0$ the *numerical rank of A* is defined as

$$\min_{\|E\| \leqslant \varepsilon} \mathrm{rank}(A - E).$$

Here ε is a desired numerical precision. The lower the numerical rank, the less information the matrix contains and vice versa. One can analogously introduce *numerical range of A* and *numerical kernel of A*.

Having an m-by-n matrix A of (exact) rank p, one can always find its *p-rank decomposition*, i.e., find rectangular matrices U and V (m-by-p and n-by-p) such that $A = UV^T$, as sketched in Figure 1.1.

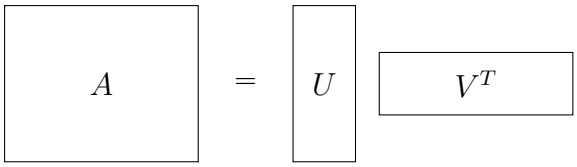

Figure 1.1: Low-rank decomposition of a rectangular matrix.

The main advantage of working with p-rank decompositions instead of the original (possibly large) matrices lies in potential massive reduction of computational and memory costs. Assuming $p \ll \min(m, n)$, the low-rank decomposition requires only $p(n + m)$ numbers to be stored instead of the usual mn numbers. Substantial savings can be also achieved when it comes to standard operations, e.g., matrix-matrix multiplication. Computing the product of $A = UV^T$ with a general n-by-m matrix, the result can be achieved in $(m+n)mp$ operations instead of m^2n of the classical case. These easy observations have been exploited to a considerable level by many authors (see, e.g., [2] or [6]) and have been motivation for this work as well.

1.1.1 Matrix sparsity

Although working with rank deficient matrices (numerically or exactly) brings the above mentioned benefits, their presence may not be apparent in some applications. However, there are surprisingly many applications, where the system matrix itself, even when it is not rank deficient, *can be divided into blocks such that most of the blocks are low-rank* (either exactly or numerically). This was first observed and exploited by Greengard and Rokhlin in [44], where the *fast multipole method* was introduced. This method later inspired many authors and consequently lead to the term of *data-sparse* matrices as it is commonly used now.

Definition 1.1.1. *Each n-by-m matrix can be characterized by $n \times m$ numbers. If the matrix with generally $\mathcal{O}(n \times m)$ nonzero numbers can be described also by significantly less numbers, e.g., only $n + m$ or $C(n + m)$ with reasonable constant C, then we refer to that matrix as* data-sparse*.*

Sometimes the data-sparsity is identified with the blockwise rank deficiency. This definition follows the work of Hackbusch [46], where we have found the first notion of the *data-sparsity*. We believe that it captures the spirit of the data-sparsity phenomenon accurately. While it is clear that the blockwise low-rank matrices fit the definition, one can also observe that other specific classes of matrices are data-sparse as Toeplitz matrices, Hankel matrices or circulant matrices[1]. Here

[1]See [53, p. 26-27] for definition of these classes of matrices

we will focus *exclusively* on the *blockwise low-rank matrices* as representatives of the data-sparse matrices. However, we wanted to stress out this distinction and we will try to distinct between these clearly and as much as possible.

Second, we would like to emphasize the difference of the just defined data-sparsity of a matrix and standard *structural sparsity* of a matrix.

Definition 1.1.2. *An n-by-m matrix is called* structurally sparse*, provided the structure of its nonzero entries can be exploited to develop a more efficient algorithm for a given purpose, usually a matrix factorization or multiplication.*

Both of the defined sparsity properties are similarly motivated, i.e., exploiting particular features of a given matrix so that the computation and storage costs of the given matrix are *cheap* in comparison to the naive standard approach. At the same time though, the techniques and methods that take these two sparsity properties into account are quite different and sometimes not compatible. In this work we will try to bridge this gap to some extent and combine both of the approaches for solving a particular problem.

Having introduced the basic notions, let us recall basic matrix decompositions. In the four following subsections are briefly summarized the three most relevant factorizations and decompositions. The summary follows Golub and van Loan [40] and Demmel [22]. As the LU factorization and the Cholesky factorization are closely related to the topic of this thesis, they are considered in more depth than the other two.

1.1.2 The Singular Value Decomposition (SVD)

For any m-by-n matrix A the *singular value decomposition* (SVD) of A consists of square unitary matrices U (m-by-m) and V (n-by-n) and an m-by-n matrix Σ such that

$$A = U\Sigma V^T, \qquad \text{with} \quad \Sigma = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_{\min(m,n)} \\ & & \end{bmatrix} \in \mathbb{R}^{m\times n},$$

where $\sigma_1 \geqslant \ldots \geqslant \sigma_{\min(m,n)} \geqslant 0$ are nonnegative real numbers called *singular values of A*. The column vectors u_i and v_i of U and V are called the *i-th left* and *right singular vectors of A*, respectively. Moreover, the singular values of A correspond to eigenvalues of the matrix A^TA.

It is easy to see that A has (exact) rank p if and only if $\sigma_p > 0$ and $\sigma_{p+1} = 0$. One of the reasons why SVD is so important can be viewed by the well-known *Young-Mirsky Theorem*[2] that is a powerful tool connected to the *numerical rank-deficiency*.

Theorem 1.1.1 (Young-Mirsky Theorem, [40, Corollary 2.3-3., p.19])**.** *Having an m-by-n matrix A with SVD of the form* $A = U\Sigma V^T$ *and given any* $1 \leqslant p \leqslant$

[2] Also known as Schmidt-Mirsky Theorem.

$\min(m,n)$, the best p-rank approximation of A with respect to either Frobenius or Euclidean norm is the matrix $\tilde{A}$ given by

$$\tilde{A} = \sum_{i=1}^{p} \sigma_i u_i v_i^T = U \Sigma_p V^T, \quad \textit{with} \qquad \Sigma_P = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_p \\ & & \end{bmatrix}.$$

A substantial drawback of SVD is number of operations necessary to compute the decomposition even for a structurally sparse A. The clasical proces of computing the SVD of a given m-by-n matrix involves *bidiagonalization proces* followed by the *QR algorithm*, resulting together in $\mathcal{O}(mn \min(m,n))$ see [22, Section 5.4, p. 237 - 254]. Great deal of work has been devoted to speed the computation up. For further details on the classical approach one can see the original work of Golub and Reinsch [39].

1.1.3 The QR factorization

Another important matrix factorization is the *QR factorization*. Assuming $m \geqslant n$ and having an m-by-n matrix A (i.e., it is "tall"), the QR-factorization of A consists of an m-by-n matrix Q and an n-by-n upper triangular matrix R with nonnegative diagonal such that $A = QR$. If A has full column rank, then Q has orthonormal columns. There are three basic approaches to obtain this factorization - *Gram-Schmidt orthogonalization process* (classical - CGS, or modified - MGS), *Householder reflections* or *Givens rotations* - all of which can be found in the cited references. Each of the approaches has certain advantages and disadvantages, e.g., time and memory costs or numerical behaviour in the sense of the loss of orthogonality.

In general, the QR-decomposition amounts to $\mathcal{O}(mn \min(m,n))$ operations. However, this can be relaxed for some classes of matrices and many applications allow for such savings. A prime example would be the problem of solving linear systems with an *upper Hessenberg matricx*[3], where one can make use Givens rotations, see [22, Section 4.4.8] or [22, p. 320]. The QR-decomposition can be modified also for other classes of matrices that have a particular structure of the nonzero entries or have only few of them, see [21] or [19].

1.1.4 The LU factorization

The third standard matrix factorization we will mention here is the *LU factorization*, closely related to the Gaussian elimination. We will restrict ourselves to the *square matrix* case here, following the work of Demmel [22, Section 2.3]. For LU-decomposition of general m-by-n matrix, one can refer, e.g., to [40, Chapter 4]. Having a strongly regular n-by-n matrix A, the LU factorization computes an n-by-n unit[4] lower triangular matrix L and an n-by-n upper triangular matrix U such that $A = LU$.

[3] Any square matrix with zeros below the first subdiagonal is called *upper Hessenberg*.

[4] A square lower triangular matrix is called *unit* provided it has all diagonal entries equal to one.

A natural way of computing the decomposition is to eliminate all the subdiagonal entries of A column by column. Provided the elimination can be done by adding multiples of rows to the ones below them, one indeed obtains the desired factorization. The elimination of the first column can be given as

$$\begin{bmatrix} 1 & a_{12} & \dots & a_{1n} \\ -\frac{a_{21}}{a_{11}} & & & \\ \vdots & & \ddots & \\ -\frac{a_{n1}}{a_{11}} & & & 1 \end{bmatrix} A = \begin{bmatrix} a_{11} & & \\ 0 & & \\ \vdots & & \tilde{A}_{22} \\ 0 & & \end{bmatrix}, \tag{1.1}$$

provided $a_{11} \neq 0$. After elimination of the first k columns one can write the partial factorization as

$$\begin{bmatrix} \tilde{L}_{11} & 0 \\ \tilde{L}_{21} & I_{n-k} \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{21} \\ 0 & \tilde{A}_{22} \end{bmatrix}, \tag{1.2}$$

where $\tilde{L}_{11}$ is k-by-k unit lower triangular and U_{11} is k-by-k upper triangular. In both cases, $\tilde{A}_{22}$ denotes the part of A yet to be factorized, which, however, has been already changed comparing to the original part of the matrix A. Once all n columns are processed, one arrives at

$$\tilde{L}A = U, \qquad \text{i. e.,} \qquad A = LU,$$

with $L = \tilde{L}^{-1}$. The Gaussian elimination can be viewed as an application of this decomposition. Indeed, solving $Ax = b$ and having $A = LU$, one can proceed in two steps

- **forward substitution** - solve $Ly = b$ for y;
- **backward substitution** - solve $Ux = y$ for x.

As for the factorization $A = LU$, the procedure sketched above can be written down, resulting in Algorithm 1.

Algorithm 1 LU factorization

Input: A strongly regular matrix A.

```
1: function LU
2:     Initialize: L ← I_n;
3:     for j = 2, ..., n do
4:         for s = j, ... n do
5:             l_{s,j-1} ← a_{s,j-1} / a_{j-1,j-1};
6:         end for
7:         for k = 1, ..., j − 1 do
8:             for i = k + 1, ..., n do
9:                 a_{ij} ← a_{ij} − l_{ik} a_{kj};
10:            end for
11:        end for
12:    end for
13:    Return the factors L and U (U is stored in the upper triangle of A);
14: end function
```

Observing the above Algorithm 1, there are several remarks. First, the entries of both of the factors are usually stored *instead of the original entries of the matrix* A (the diagonal is reserved for the U factor). Second, the computation of the entries of L on lines 4-6 can be moved inside the *for*-loop on lines 7-11. The difference between the consequent variations gives the name to the *right-looking* and *left-looking* versions of the LU factorization (see, e.g., [3, Section 1.2.4]). Let us now also point out the convenient notation of [3], which we will adopt later on. The *for*-loop on lines 4-6 is referred to as *factorize(*$A_{*,j}$*)* procedure and the *for*-loop on lines 7-11 is referred to as *update(*$A_{*,j}, A_{*,k}$*)* procedure. It is also worth noting that the three main *for*-loops of indicies j,k and i can be in principle reshuffled, resulting into *row-*, *column-* or *submatrix-based version of LU factorization*, see, e.g., [63, Appendix 1]. This classification is even more general than just distinguishing the left- and right-looking variants.

The fourth straightforward observation regarding Algorithm 1 is that it breaks down if and only if any of the *pivots* is equal to zero, i.e., if somewhere during the factorization $\tilde{A}_{22}$ has its first diagonal entry equal to zero. It is well-known that this occurs if and only if at least one of its leading principal submatrices is singular (see [22, Theorem 2.4, p. 39]). This can obviously happen even though the matrix is nonsingular. However, it is also known that for any nonsingular matrix A there exists a permutation matrix P such that PA has all the principal leading submatrices nonsingular (see [22, Theorem 2.5, p. 39]).

Last but not least, even if all the principal leading submatrices are nonsingular the above procedure may still result into a numerically unsound algorithm. Indeed, nonzero pivots may be much smaller (in absolute value) then the rest of the entries of the column. This allows for (possible) cancellation of valid digits and also propagation of the errors in the matrix entries. This issue is usually addressed by the so-called *partial pivoting procedure*. Dealing with the k-th step, i.e., with the k-th column, the procedure will search in the subdiagonal part of the column (entries k to n) for the largest entry. Once it has been found, the procedure swaps the corresponding rows so that the pivot (diagonal entry of the column) is the largest element (in absolute value) in the subdiagonal part of the column. In practice, the LU factorization with partial pivoting is observed to be numerically sound, but one can construct counterexamples showing that even the LU factorization with partial pivoting may fail due to the *factor growth*. For more details on the numerical errors analysis of LU decomposition (or Gaussian elimination) with and without the pivoting strategies, see [22, Section 2.4] or [40, Section 4.3 and 4.4]. Following the above sketched approach would amount to $\mathcal{O}(n^3)$ operations. This has been relaxed on multiple levels in the general case and even more so in particular practical cases as we will see throughout this thesis.

1.1.5 The Cholesky factorization

This section will focus on the particular case of the LU decomposition when A is symmetric, positive definite. Although one might find the SPD property quite restrictive, in many cases of interest one obtains such matrices. To name at least a few, linear problems involving SPD matrices often arises from discretizatiuon of PDE problems (see [28, Chapter 2 and 6]), optimization methods [29] or the linear regression models. The basics were laid down above in the LU factorization. If

A is symmetric, positive-definite, the decomposition enjoys number of additional preferable properties.

The *Cholesky factorization* of an n-by-n SPD matrix A corresponds to the LU factorization in which $U = L^T$ (i.e., $A = LL^T$), where L is no longer unit lower triangular, but it has positive entries on diagonal. The existence of Cholesky factorization is in fact equivalent to the SPD property of a given matrix. One sometimes also encounters the so-called *square root free Cholesky factorization* that corresponds to $A = LDL^T$, where the *unit* property of L is restored.

First obvious benefit of the Cholesky factorization in comparison to the general LU factorization is that one needs only half of the numbers to store the factorization. Another easy observation is that any SPD matrix has all its leading submatrices symmetric, positive-definite as well[5] and hence nonsingular. Consequently the proposed procedure cannot break down (in exact arithmetic). The following result is less obvious and touches upon the *numerical stability* and unnecessity of the pivoting for the Cholesky factorization. The original result is due to Wilkinson [82] and reads as follows.

Theorem 1.1.2 (Backward stability of Cholesky factorization, [82])**.** *Computation of the Cholesky factorization of a given A in finite precision on level ε_{mach} without pivoting yield the Cholesky factor L such that there exists a perturbation matrix E satisfying (in exact arithmetic)*

$$\begin{aligned} A + E &= LL^T, \\ \|E\|_2 &\leqslant 2.5\sqrt{n^3} \cdot \varepsilon_{mach} \|A\|_2. \end{aligned}$$

In other words, the finite precision result can be interpreted as an exact arithmetic result for *a matrix very close to the original one*, i.e., the Cholesky decomposition is *backward stable*, see [40, Section 3.2, p. 36-37]. The importance of this result cannot be underestimated, as it both considerably reduces the overall costs of the computation and substantially simplifies the analysis.
Bearing in mind the unnecessity of pivoting and that A is SPD, one can easily write down the code for the column oriented Cholesky factorization as follows.

[5] See the Sylvester criterion [22, Proposition 2.2(2)]

Algorithm 2 Column-oriented, left-looking Cholesky factorization

Input: A square n-by-n SPD matrix A.

function CHOL
  for $j = 2, \ldots, n$ **do**;
    $l_{j-1,j-1} \leftarrow \sqrt{a_{j-1,j-1}}$;
    for $s = j, \ldots n$ **do**
      $l_{s,j-1} \leftarrow \frac{a_{s,j-1}}{l_{j-1,j-1}}$;
    end for
    for $k = 1, \ldots, j-1$ **do**;
      for $i = k+1, \ldots, n$ **do**;
        $a_{ij} \leftarrow a_{ij} - l_{jk} l_{ik}$;
      end for
    end for
  end for
  Return the factor L;
end function

Algorithm 2 can be further modified to eb more efficient for many practical classes of matrices. In the coming chapters we focus on the *structurally sparse* and *data sparse* matrices in particular.

1.2 Structurally sparse Cholesky factorization

In many cases of interest the system matrix is *structurally sparse*, meaning that its nonzero entries have a particular structure so that a substantially more efficient algorithm can be achieved, provided this structure is taken into account. A thoroughly studied class of such matrices are, e.g., *banded matrices*[6], or, to be even more specific, *tridiagonal matrices* or *pentadiagonal matrices*. Those matrices are often encountered when discretizing simple differential operators with a three- or five-point stencils, e.g., for solving the Poisson equation $\Delta u = f$.

Some authors, e.g., [66], define sparse matrices as n-by-n matrices with number of the nonzero entries proportional to n (i.e., with $\mathrm{nnz}(A) = \mathcal{O}(n)$) or with fixed small number of nonzero entries per row and column (i.e., with $\mathrm{nnz}(A_{*,j} = \mathrm{nnz}(A_{j,*} = \mathcal{O}(1))$. However, a common definition of *structural sparsity* nowadays is wider, i.e., given matrix is structurally sparse provided one can *strongly benefit from the structure of its nonzero elements to obtain a more efficient procedure to solve the given problem*. Since neither [40] nor [22] consider this topic with more details[7], the main reference source for this section will be the book of George and Liu [33] (and the work cited there).

In general, there are two main goals - to reduce the time and the memory costs, i.e., to not store zero entries and to not carry out multiplications by zeros and additions of zeros. In the Cholesky factorization case, this is achieved by splitting the overall computation into two parts, *symbolic* and *numerical*. First,

[6] A square matrix A is called *banded*, provided there is k such that $a_{ij} = 0$ for all i, j such that $|i - j| > k$. Minimal such number k is called *bandwidth of* A.

[7] Although in [22, Section 2.7.4, p. 83 - 92] one can find brief remarks on the overall problematic of sparse matrices.

the *symbolic computation* is carried out, i.e., the sparsity structure of the factor L is computed assuming the *non-cancellation*[8]. The symbolic part often involves suitable symmetric permutations to reduce the amount of the *fill-in*, i.e., to reduce the amount of the nonzero entries of L that were zero in the original matrix. Once the structure of the Cholesky factor L is computed, the necessary memory is allocated and the numerical factorization is carried out. Let us emphasize that this approach is not applicable if the matrix are not SPD. In general case one has to account for *pivoting* that is *necessary to achieve numerical stability of the LU decomposition*. However, this is a dynamical process that *depends on the computed values and cannot be computed in advance*.

The following subsection is concerned with the first part of the Cholesky decomposition - the symbolic factorization, i.e., how to determine the fill-in entries and how to minimize their amount. Note that we have not discussed *how to exploit the structural sparsity in the computation*, e.g., particular schemes of how to store the matrix, how to carry out the multiplication etc, so far. For this, interested reader is referred to, e.g., [33] or [66].

1.2.1 Graph theory point of view

This subsection will be focused on the Cholesky factorization from the point of view of the graph theory. There are two basic and different ways of how to look on the Cholesky factorization from this point of view. Both of them have their pros and cons and both of them can be extremely effective in certain cases. For further elaboration on this topic, one can see the book by George and Liu [33] and the book by Duff, Erisman and and Reid [23] and the works cited in those.

Adjacency graph

Having an n-by-n SPD matrix A, one can define the *graph associated with A*, denoted by $G(A)$ or G^A, as a graph $G(A) = (V^A, E^A)$ on n vertices, i.e., $V^A = \{1, \ldots, n\}$ and with edges corresponding to the nonzero entries of A, i.e., (i, j) is an edge in the graph $G(A)$, provided $a_{ij} \neq 0$, i.e., $E^A = \{(i, j) \,|\, a_{ij} \neq 0\}$. Note that the necessary and sufficient condition for this graph to be well-defined is *structural symmetry of A*, i.e., symmetry in sense $a_{ij} \neq 0 \iff a_{ji} \neq 0$. In case that this property is not satisfied, one can still work with concepts of *directed graphs*, i.e., graphs in which each edge *has a direction*. We will work with directed graphs in later part of this work as well.

As the rest of this section deals with the symbolic computation *only*, the values will not be written down into the matrices and only zeros and nonzeros will be distinguished, although the SPD property is still assumed. An example of this notation and a particular pair $A, G(A)$ is in Figure 1.2 below. Considering a node $i \in V^A$, one can define the *set of vertices adjacent to i in $G(A)$* as the set of vertices that are connected with i by some edge in E^A, i.e., $Adj_{G(A)}(i) = \{j \,|\, (i, j) \in E^A\}$. The notion of adjacency can be extended to any subset of V^A and not only a single vertex. Considering the example from Figure 1.2, $Adj_{G(A)}(3) = \{1, 2, 4\}$ and $Adj_{G(A)}(\{1, 4\}) = \{5, 3\}$.

[8] This is a common assumption stating that sum of any two nonzero numbers is again nonzero. Consequently, the computed sparsity structure is only *upper bound*, which is, however, usually very tight.

$$A = \begin{bmatrix} * & & * & & * \\ & * & * & & \\ * & * & * & * & \\ & & * & * & \\ * & & & & * \end{bmatrix}$$

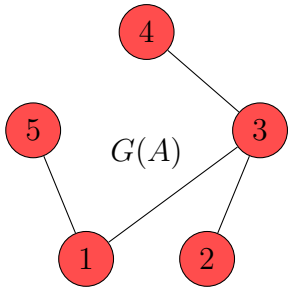

Figure 1.2: On left is an 5-by-5 SPD matrix A and on the right is its graph.

The following observation highlights the fact that the above example is a rather convenient one in the sense that the graph $G(A)$ has only one component.

Observation 1.2.1 ([33, Section 3.1, p.39])**.** *In some cases, the graph $G(A)$ is not connected, i.e., there are vertices that has no path between them, i.e., the graph has more than one component. This happens if and only if A can be permuted to a block-diagonal form. If this is the case, one can treat each of the diagonal blocks* separately *. Therefore, it is usually assumed that the graph $G(A)$ has only one component*[9]*. Throughout this thesis we adopt this assumption as well, i.e., A is assumed to be* irreducible*.*

A simple yet crucial step is to interpret one step of the Cholesky factorization in terms of the graph $G(A)$. First, one needs a suitable notation. For a given n-by-n SPD matrix, let us first carry out the first $j-1$ steps of the Cholesky factorization, as in Algorithm 2. At this point we have computed the first j columns of L (since the j **for**-loop in Algorithm 2 starts with $j = 2$). $A^{(j)}$ denotes the current matrix stored during the j-th step of the factorization. Its lower triangular part contains the first j columns of the Cholesky factor and the remaining $n-j$ columns of $A^{(j)}$ are columns of A yet to be factored. The upper triangle is filled symmetrically. Moreover, denote the square $(n-j)$-by-$(n-j)$ bottom-right most submatrix of $A^{(j)}$ by H^j, i.e., H^j corresponds to the part of A that is yet to be factored (in 1.1-1.2 in the first sections those were labelled $\tilde{A}_{22}$, regardless of the step of the Gaussian elimination). Consider A from Figure 1.2. $A^{(1)} \equiv A$ and $A^{(3)}$ are given below, with H^3 emphasized by red color.

$$A^{(1)} = \begin{bmatrix} * & & * & & * \\ & * & * & & \\ * & * & * & * & \\ & & * & * & \\ * & & & & * \end{bmatrix}, \qquad A^{(3)} = \begin{bmatrix} * & & * & & * \\ & * & * & & \\ * & * & * & * & * \\ & & * & * & \\ * & & * & & * \end{bmatrix}$$

Changes of the matrices throughout the elimination are described by the following lemma.

[9] If $G(A)$ has only one component, the matrix A is called irreducible, otherwise it is called reducible.

Lemma 1.2.1 (The fill-in lemma, [33, p. 94-95]). *Let A be an n-by-n SPD matrix and assume j steps of the Cholesky factorization were carried out, yielding matrices $A^{(j)}$ and H^j as above. Denote the row indicies of all of the* subdiagonal *nonzero entries of $(j+1)$-th column of $A^{(j)}$ as $i_1 < \ldots < i_k$, i.e., $Adj_{G(H^j)}(j+1) = \{i_1, \ldots, i_k\}$.*
Consider the following step of the factorization, i.e., elimination of the $(j+1)$-th column of $A^{(j)}$. The elimination of the element $\left(A^{(j)}\right)_{l,j+1}$ for $l \in \{i_1, \ldots, i_k\}$ modifies the graph $G(H^j)$ in the following way.

- *The edge $(l, j+1)$ is deleted;*
- *New edges are created to connect the vertex l with every vertex that is both larger than l and is connected with $j+1$ in $G(H^j)$.*

Consequently, carrying out the $(j+1)$-th step of the factorization will modify the graph $G(H^j)$ to $G(H^{j+1})$ by deleting the vertex $(j+1)$ and connecting pairwise all of the vertices from $Adj_{G(H^j)}(j+1)$.
Working with the graph $G(A^{(j)})$, the deletion step *is simply omitted as the Cholesky factor automatically inherits the structure of A, i.e., only new edges are created in the way described above, corresponding to the newly created nonzeros, i.e., to the* fill-in.

A simple consequence of the above **lemma** is the classical result stating that an element l_{ij} of the Cholesky factor is nonzero if and only if $a_{ij} \neq 0$ or there exists an index $k < \min\{i, j\}$ such that $l_{ik} \neq 0$ and $l_{kj} \neq 0$, see [33, Lemma 5.1.1, p.95]. The second case, i.e., creation of a fill-in, is depicted in the Figure 1.3.

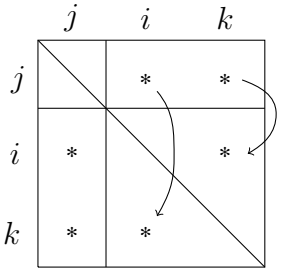

Figure 1.3: Illustration of the fill-in in a single position (symmetrically). The diagonal is always nonzero, but for simplicity the stars are replaced by the diagonal line here.

The evolution of the graph of the Cholesky factor together with the corresponding matrices for the matrix from Figure 1.2 is given below in Figure 1.4.

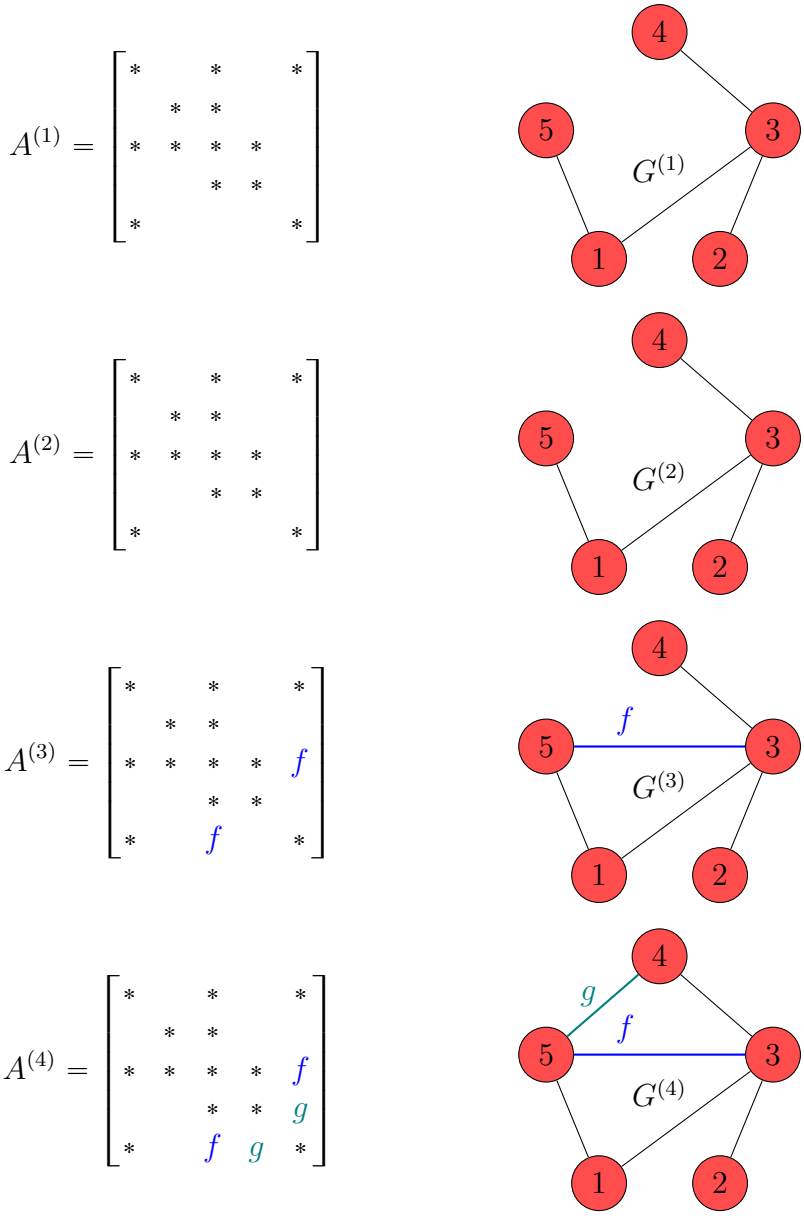

Figure 1.4: The symbolic Cholesky factorization is carried out for some 5-by-5 SPD matrix. The fill-in is illustrated both in the matrix and in the corresponding graphs by symbols f and g.

In order to formulate the result *at once* and not recurrently, it is convenient to introduce the *reachable sets*. Having vertices i and j in $G(A)$ and some additional set of vertices S, one says that j is reachable from i through S, provided there exists a path from i to j using only vertices from S. The set of all reachable vertices from i through S is denoted $Reach_{G(A)}(i, S)$. Having this notion, one can connect the graph of $A^{(n-1)}$ (i.e., graph of the filled-in matrix $F = L + L^T$, i.e., the graph of the Cholesky factor - copied symmetrically above the diagonal) and the graph of A, as follows.

Theorem 1.2.1 ([33, Theorem 5.1.2, p.98]). *Having an n-by-n SPD matrix A and its Cholesky factorization $A = LL^T$, denote $F = L + L^T$ the matrix carrying the structure of the Cholesky factor. Then $G(F)$ is created from $G(A)$ by connecting every two vertices in $G(A)$ that have a path between them using only* lower indexed vertices*. In other words, the graph $G(F)$ is graph on vertices $\{1, \ldots, n\}$ with edges given by*

$$E^F = \left\{(i,j) \,|\, i \in Reach_{G(A)}(j, S_{ij})\right\}, \text{ with } \quad S_{ij} = \{k < \min\{i,j\}\}\,.$$

Elimination tree

Another important way of looking at the Cholesky factorization is through the optic of *elimination trees*. As this topic is not considered in [33] nor in [23], we refer to the overview paper by Liu [60]. Nice summary can eb also found in the paper by George [32].

Having an n-by-n SPD matrix A and its Cholesky factorization $A = LL^T$, the matrix L induces a particular rooted tree on vertices $\{1, \ldots, n\}$ called *elimination tree of A*, denoted by $\mathcal{T}(L)$. The root is always the vertex n and the tree structure is induced by the first nonzero entries of the columns of L. To be more specific, vertex k is the *parent* of vertex j (or j is a son of k), provided the first nonzero subdiagonal entry of the j-th column $L_{*,j}$ is in the k-th row, i.e.,

$$k = parent(j) \quad \Longleftrightarrow \quad k = \min\{i \,|\, l_{ij} \neq 0\}. \tag{1.3}$$

Note that this definition relies on the non-cancellation assumed earlier. An example of the elimination tree for the previously considered 5-by-5 matrix A is given below in Figure 1.5.

$$F = L + L^T = \begin{bmatrix} * & & * & & * \\ & * & * & & \\ * & * & * & * & * \\ & & * & * & * \\ * & & * & * & * \end{bmatrix}$$

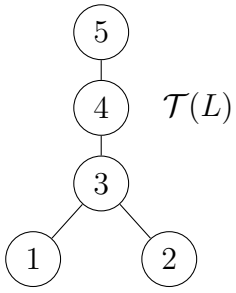

Figure 1.5: Consider the 5-by-5 SPD matrix A from 1.3. The fill-in structure of the Cholesky factor L is given on the left by matrix $F = L + L^T$. The elimination tree $\mathcal{T}(L)$ is given on the right.

The following observation highlights the fact that the above example was a rather convenient one in some sense, analogously to Observation 1.2.1.

Observation 1.2.2 ([60, Section 2.2, p.137]). *In some cases, the elimination tree $\mathcal{T}(L)$ can be in fact an* elimination forest*, i.e., constructing the graph using* (1.3) *results in several rooted trees. This can occur if and only if one of the columns of the Cholesky factor has* no off-diagonal nonzero entries*. It is possible that this cannot happen if A is irreducible. Hence, throughout this thesis is $\mathcal{T}(L)$ assumed to be a rooted tree.*

Having the elimination tree, it is easy to define the j-th subtree for $1 \leqslant j \leqslant n$ as the the subtree of $\mathcal{T}(L)$ rooted in vertex j, denoted by $\mathcal{T}[j]$. Another notion commonly used in this area is the *ancestor* of a vertex as a generalization of the term parent. The vertex i is an *ancestor* of the vertex j, provided that there are vertices $k_1, \ldots, k_l$ such that

$$k_1 = parent(j),\ k_2 = parent(k_1), \ldots, k_l = parent(k_{l-1}), j = parent(k_l).$$

In other words, there is an oriented path in the elimination tree from one of the vertices to another (and the orientation is important). As stated above, the elimination tree is a powerful tool to determine the structure of the Cholesky factor as it *completely captures the structure of the fill-in propagation during the factorization* as is highlighted in the following theorem.

Theorem 1.2.2 ([32, Theorem 3.1, Theorem 3.2])**.** *Let A be an n-by-n SPD matrix with the Cholesky factorization $A = LL^T$ and denote the elimination tree of A by $\mathcal{T}(L)$. Then $l_{ij} \neq 0$ if and only if at least one of the following holds.*

- $a_{ij} \neq 0$*;*
- *The vertex j is an* ancestor *of some vertex k in $\mathcal{T}(L)$ such that $a_{ik} \neq 0$.*

On the other hand, each nonzero entry a_{ik} creates a sequence of (potential) fill-in in the i-th row in columns corresponding to the ancestors of the vertex i in the elimination tree $\mathcal{T}(L)$.

The result of the Theorem 1.2.3 can be further reformulated to provide a deeper insight as well as a computational tool to take advantage of the elimination tree. In order to do so it will be useful to introduce yet another definition - the *structure of a given vector*. Given a vector $x \in \mathbb{R}^n$, the vector $Struct(x)$ is defined as set of indices of nonzero entries in x. For example, considering $x = (-1, 0, 0, 3, 0)^T$, one has $Struct(x) = \{1, 4\}$. The key is the observation that fill-in process does not propagate (or copy) only the nonzeros in the i-th row, but rather the entire nonzero structure of the k-th column below the i-th row.

Theorem 1.2.3 ([32, Theorem 2.5], [60, Theorem 3.6])**.** *Let A be an n-by-n SPD matrix with the Cholesky factorization $A = LL^T$ and consider the elimination tree $\mathcal{T}(L)$. The (nonzero) structure of the j-th column of the Cholesky factor L of A is equal to the lower-triangle structure of the j-th column of A united with the structure of already computed columns k_l of L such that $j = parent(k_l)$, from row index j below. In short, one can write*

$$Struct(L_{*,j}) = Struct(A_{j:n,j}) \bigcup \left(\bigcup_{j=parent(k)} Struct(L_{j:n,k}) \right)$$

The above result is sometimes called the *column structure replication* and is key for structural computations. However, in some instances, one might not have the elimination tree available. Therefore, another reformulation may be useful. Namely, one can write

$$Struct(L_{*,j}) = Struct(A_{j:n,j}) \bigcup \left(\bigcup_{k \in Struct(L_{j,1:j})} Struct(L_{j:n,k}) \right). \tag{1.4}$$

This reformulation allow us to avoid traversing through the elimination tree structure, i.e., through the *parent* vector, *provided we know the structure of the j-th row of L*. This also makes the term *column structures replication* more visible, see Figure 1.6.

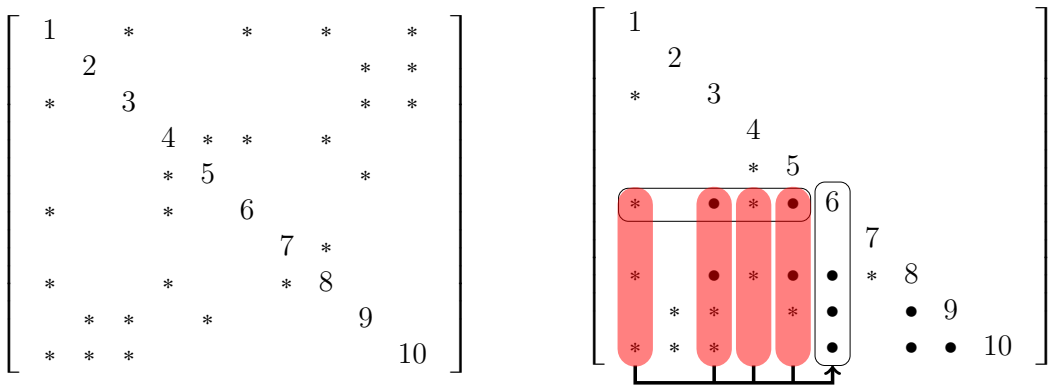

Figure 1.6: Consider the 10-by-10 sparse matrix (left) and its Cholesky factor (right), where the fill-in entries are highlighted by the symbol •. The right-hand side of the Figure illustrates the column structures replication in the sixth step of the Cholesky factorization.

This naturally leads to the definition of the *j-th row subtree* $\mathcal{T}_r[j]$ of $\mathcal{T}(L)$, which is the subtree of $\mathcal{T}(L)$ consisting of vertices $Struct(L_{j,1:j)})$. Let us point out here the difference between the j-th rooted subtree $\mathcal{T}[j]$ and the j-th row subtree $\mathcal{T}_r[j]$. Due to Theorem 1.2.2 we easily obtain $\mathcal{T}_r[j] \subset \mathcal{T}_r[j]$. However, the equality usually does not take place as illustrated in the Figure below.

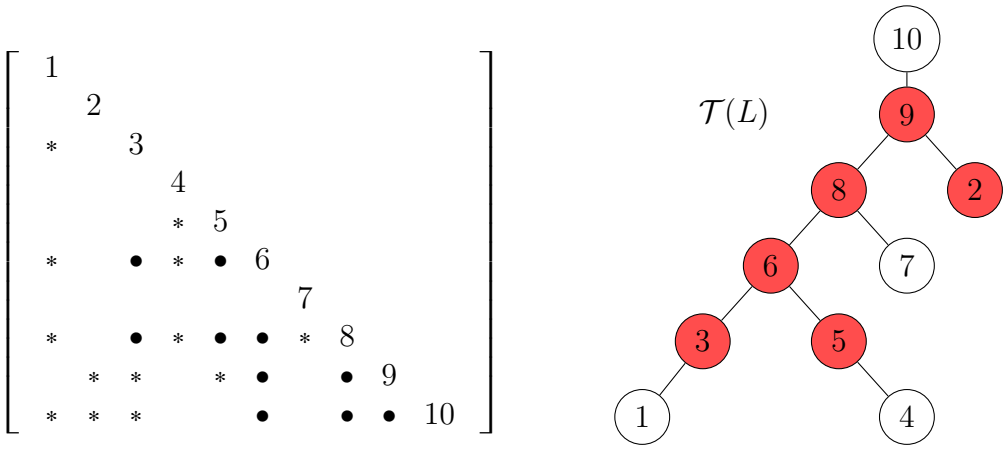

Figure 1.7: Consider the same 10-by-10 sparse matrix as in Figure 1.6. Its Cholesky factor is on the left (the fill-in is highlighted by the symbol •). On the right is the elimination tree $\mathcal{T}(L)$. The 9-th rooted subtree $\mathcal{T}[9]$ contains vertices $\{1, \ldots, 9\}$, whereas the 9-th row subtree is highlighted and contains only vertices $\{2, 3, 5, 6, 8, 9\}$.

Also, it might not be quite obvious at the first sight that vertices corresponding to $Struct(L_{j,1:j)})$ form a rooted subtree of $\mathcal{T}(L)$ but this can be easily observed from Theorem 1.2.2 and Theorem 1.2.3 as well. For further references one can consult either [60] or [32]. The main contribution of row subtrees lies in the

formulation above, i.e., as long as we can cheaply obtain the row subtrees (in fact their leaves already characterize the whole tree), the above formula (1.4) can be used to quickly determine the nonzero structure of the j-th column.

Hence, the important question is, whether one can cheaply compute the leaves of the row subtrees. Another related issue that has not been adressed is how to *construct* the elimination tree itself. Note that $\mathcal{T}(L)$ has been defined using the Cholesky factor L, which is not available in advance. Considerable amount of work has been devoted to this problem and we omit this topic here simply not to stretch into too many directions. Nonetheless, this topic is considered quite in-depth in the paper of George [32, Section 4-7]. The row subtrees are considered in detail in [32, Chapter 3].

1.2.2 Fill-in and reorderings

In the previous subsection we mentioned how the fill-in in the Cholesky factorization can be analysed by means of the graph theory. The natural following step is to use the analysis to *reduce* or *minimize* the amount of the fill-in. The well-known *arrow matrix* example and its fill-in minimizing permutation are shown in Figure 1.8.

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{bmatrix} \qquad F = L + L^T = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

$$PAP^T = \begin{bmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{bmatrix} \qquad \tilde{F} = \tilde{L} + \tilde{L}^T = \begin{bmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{bmatrix}$$

Figure 1.8: The first row shows the original matrix A and the fill-in matrix $A^{(5)}$ of its Cholesky factor. The second row considers the same matrix, that is symmetrically permuted by switching the first and the last columns and rows as well as its fill-in matrix.

Having an n-by-n SPD matrix A, the basic idea is to apply *symmetric* permutation to the matrix A, i.e., find a suitable n-by-n *permutation matrix* P and factorize PAP^T instead of A as shown above in Figure 1.8. In practice, one usually gets the permutation matrix at the first step of the symbolic part of the Cholesky factorization. It is important to bear in mind that finding the *fill-in minimizing* permutation is in general NP-complete problem for a general structurally symmetric matrix (see [22, Section 2.7.4, p. 84]) and one needs to consider heuristic procedures to minimize the fill-in. Among the most popular approaches are, e.g., minimal degree reorderings, nested dissection reordering or reverse Cuthill-McKess reordering, see [8, Section 3.3, p.437]. Once again, this area of research as a whole goes far beyond the scope of this text and we will not consider it in more details.

To conclude this section, we present a left-looking column-oriented structurally sparse version of the Algorithm 2 without symbolic considerations. Algorithm 2 would be *ridiculously inefficient* even for a structurally sparse input.

Algorithm 3 Structurally sparse column-oriented left-looking Cholesky factorization

Input: A square n-by-n SPD matrix A.

1: **function** SCHOL
2: Determine a permutation matrix P;
3: Determine the structure of the resulting factor L;
4: In particular, determine the row and column structures $Struct(L_{*j}), Struct(L_{j*})$;
5: **for** $j = 2, \ldots, n$ **do**
6: $l_{j-1,j-1} \leftarrow \sqrt{a_{j-1,j-1}}$;
7: **for** $s \in Struct(L_{j:n,j})$ **do**
8: $l_{s,j-1} \leftarrow \frac{a_{s,j-1}}{l_{j-1,j-1}}$;
9: **end for**
10: **for** $k \in Struct(L_{j,1:j-1})$ **do**
11: **for** $i \in Struct(L_{j:n,k})$ **do**
12: $a_{ij} \leftarrow a_{ij} - l_{jk} l_{ik}$;
13: **end for**
14: **end for**
15: **end for**
16: Return the factor L;
17: **end function**

1.3 Iterative methods and preconditioning

The previous section was devoted to the Cholesky factorization for the *structurally sparse* matrices, yielding substantially more efficient procedures for solving the linear systems. Indeed, the Cholesky factorization (regarded as a direct method for solution of $Ax = b$) is used in many cases in practice, mainly because of its *robustness* and predictable time and memory requirements. Nonetheless, for an increasing amount of problems the method is *simply not feasible* due to its poor scalability with respect to the ever growing dimension of many problems. For such problems (not exclusively), either an *iterative method* or perhaps a *multigrid method* is a natural choice of a numerical method for solving $Ax = b$. The multigrid methods will not be considered in this text, though they are very popular for particular applications (see, e.g., [13]). But note that a typical way to use multigrid methods is to consider them as auxiliary procedures to *improve the behaviour of the iterative methods* as we will mention later on. On the other hand, the iterative methods, and the *Krylov subspace methods* in particular, open new ways how to utilize the Cholesky factorization and *its approximations, in particular*. The following section will be refering mainly to the book of Liesen and Strakoš [58], when it comes to the Krylov subspace methods and to the survey paper of Benzi [8], regarding the preconditioners.

The best-known iterative method in the modern history is without any doubts

the *conjugate gradient method* (CG). This method was introduced almost simultaneously by Hestenes and Stiefel [49] and Lanczos [57] and was at first regarded as a *direct method*. Only after twenty years or so, the method gain a large amount of interest in the numerical comunity (see the very important paper of Reid [67]) and started the skyrocketing development of the Krylov subspace methods, continuing with the methods as MINRES and SYMMLQ by Paige and Saunders [64] , the GMRES method by Saad and Schultz [71] or the Bi-CGSTAB method by van der Vorst [81]. As this text deals exclusively with square SPD matrices, we will restrict ourselves to the CG method, described in the Algorithm 4 below (see [58, Algorithm 2.5.1, p.41]).

Algorithm 4 The Conjugate Gradient Method

Input: A square SPD matrix A, right-hand side vector b, initial guess x_0, maximal number of iterations $k_{\max}$, stopping criterion.

function CG
  Initialize: $r_0 = b - Ax_0$, $p_0 = r_0$.
  for $k = 1, 2 \ldots, k_{\max}$ **do**
    $\alpha_{k-1} \leftarrow \|r_{k-1}\|^2 / p_{k-1}^T A p_{k-1}$;
    $x_k \leftarrow x_{k-1} + \alpha_{k-1} p_{k-1}$;
    $r_k \leftarrow r_{k-1} - \alpha_{k-1} A p_{k-1}$;
    Stop the iteration provided the stopping criterion is satisfied;
    $\omega_k \leftarrow \|r_k\|^2 / \|r_{k-1}\|^2$;
    $p_k \leftarrow r_k + \omega_n p_{k-1}$;
  end for
  Return the approximation x_k of the solution of $Ax = b$;
end function

The theorem below summarizes some of the vital properties of the CG mathod in exact arithmetic.

Theorem 1.3.1 (The CG method, [58, Theorem 2.3.1, p.23])**.** *Let A be an n-by-n SPD matrix, $b \in \mathbb{R}^n$ a given right-hand side vector and $x_0 \in \mathbb{R}^n$ a given initial guess of the solution $Ax = b$. Consider the initial residual $r_0 = b - Ax_0$ and for $k = 1, 2, \ldots$ define the* k-th Krylov subspace associated with A and r_0 *as*

$$\mathcal{K}_k(A, r_0) = span\left\{r_0, Ar_0, \ldots, A^{k-1} r_0\right\}.$$

The CG method at the k-th step yields an approximation x_k to x such that

$$x - x_k \perp_A \mathcal{K}_k(A, r_0),$$

or, equivalently,

$$\|x - x_k\|_A = \min_{z \in x_0 \mathcal{K}_k(A, r_0)} \|x - z\|_A,$$

where $\|\cdot\|_A$ is the so-called energy norm given by A, induced by the energy inner product given by A, i.e., $\|v\|_A^2 = \langle v, v \rangle_A = v^T A v$. The orthogonality in the first characterization is considered with respect to the A-inner product $\langle \cdot, \cdot \rangle_A$.

The performance of this method is usually measured by the *convergence rate*, i.e., by the quantity $\|x-x_k\|_A/\|x-x_0\|_A$ (for theoretical purposes) or by $\|r_k\|/\|r_0\|$ (often used in the actual computation). Both of these quantities are closely related to the *distribution of the eigenvalues of* A. The usual beneficial property of A that suggests a good performance of CG is having a *very small condition number*. A natural idea to improve the convergence is to transform the matrix A so that the transformed one has considerably decreased condition number. This transformation is in the terminology of the iterative methods called *preconditioning* and corresponds to one of the following transformation.

- *left preconditioning* using a regular n-by-n matrix M:

$$M^{-1}Ax = M^{-1}b;$$

- *right preconditioning* using a regular n-by-n matrix M:

$$AM^{-1}y = b \quad \text{and} \quad x = M^{-1}y;$$

- *split preconditioning* using regular n-by-n matrices M_1, M_2:

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b \quad \text{and} \quad x = M^{-1}y.$$

The matrix M (or matrices M_1 and M_2) is called *left* or *right preconditioner* (or *split preconditioners*), respectively. In practice, one usually obtains M rather than M^{-1} and therefore reformulates applications of M^{-1} as a linear problem involving the matrix M and a particular right-hand side. Combining this transformation with the CG method, i.e., assuming *preconditioned CG*, one can see that in each iteration one has to solve a linear system, if M^{-1} is not provided explicitly. This implies that solving problems with matrix M *needs to be very fast* in order to have a competitive method. On the other hand, M *needs to capture well the spectral information of* A in order to fulfil the original goal, i.e., to decrease the conditional number of the matrix of the preconditioned system and/or cluster the eigenvalues of that matrix, hopefully around 1. Note that those two properties required from M (or M_1 and M_2) are *competing with each other* and one has to strive for a suitable balance of those two requirements.

Some of the possible ways how to obtain such matrices, commonly used in practice, are considered in the following section.

1.4 Preconditioners

Focusing on the case of preconditioned conjugate gradients method, an important observation is that if one would like to use preconditioner for the CG method, the preconditioned system *has to be SPD again*. A possible way to ensure this is to consider *split preconditioned CG* and take $M_1 = L, M_2 = L^T$ (or any $M_1 = M_2^T$ regular in general). In practice the preconditioner can be applied as right (or left) preconditioner with $M = LL^T$. The equivalence guarantees the positive definiteness and symmetry of the right (or left) preconditioned system and justifies the usage of the CG method. For detailed reasoning see, e.g., [70, Section 9.2].

Assuming $A = LL^T$ is the Cholesky factorization of A, it is easy to observe that taking $M_1 = L, M_2 = L^T$ as the split preconditioners results into a system with identity matrix. It is also easy to check that this exactly split preconditioned CG will converge in one iteration. On the other hand, the purpose of the iterative methods is to *avoid the computation of the complete Cholesky factorization*, i.e., avoiding the direct method. Consequently, one can try to only *approximate* the complete Cholesky factorization, i.e., alter the algorithm so that the *computation is significantly cheaper* but at the same time the preconditioned matrix is close to the identity matrix, or at least will enjoy of the key properties mentioned previously, at least to some extent. One usually refers to such technique as to the *incomplete Cholesky factorization* and it is one of the most common and easiest ways to obtain a preconditioner for a given problem.

Since the method of conjugate gradients has become extremely popular, there is an immense amount of preconditioning techniques, based on plethora of different approaches and viewpoints of the original problem. The center of this subsection is to point out the classical, widely used ways of preconditioning, while briefly mentioning other possibilities as well. The references are mainly from the survey paper by Benzi [8], Chapter 3 in particular, and the works cited there.

1.4.1 Cholesky-based preconditioners

The commonly used idea is to *drop some elements of the factor L during the computation* and thereby obtain a matrix $\bar{L}$, the *incomplete Cholesky factor* $\bar{L}$ such that $\bar{L} \approx L$. The challenges are, among others, to determine the criterion according to which the entries will be kept or dropped and also to establish the existence and stability theory, as the dropping makes the so far presented results not applicable. In particular, *breakdowns may occur*, i.e., diagonal elements may become zero or negative during the incomplete factorization, resulting in a preconditioner that is not SPD.

There are several different ideas how to determine the dropping criterion, but one always needs to bear in mind that the resulting factor $\bar{L}$ has to be *nonsingular* in order to keep the SPD property for the preconditioned system

$$\bar{L}^{-1} A \left(\bar{L}^{-1}\right)^T y = \bar{L}^{-1} b \qquad \text{and} \qquad \left(\bar{L}^{-1}\right)^T y = x.$$

The most commonly known method to obtain an incomplete factorization is the *level of fill-in* method. Each entry of the original matrix is assigned an integer defining its *level* as

$$lev_{ij} = \begin{cases} 0, & a_{ij} \neq 0 \text{ or } i = j \\ \infty, & \text{otherwise} \end{cases}$$

And each time an element is modified by the classical Cholesky factorization (see Algorithm 2, 3) its level lev_{ij} is updated to

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}.$$

The integer lev_{ij} represents how many level of fill-in were used to *fill-in this particular entry of the Cholesky factor*. The incompleteness is then imposed by assuming only entries up to some *fixed level*.

$$l_{ij} \leftarrow 0, \qquad \text{for all } (i,j) \text{ such that } lev_{ij} > p.$$

The classical notation is $IC(p)$ standing for incomplete Cholesky factorization of p-th level.

A different approach is to impose a certain *sparsity structure* to the Cholesky factor (given by the user) and whenever an entry outside the pattern should be filled-in it is dropped instead. Both of these methods can be effective in many cases, but in general they are held back by the *independence on the values of the entries* and also by reappearing possibility of a breakdown.
A possible alternative is to introduce a *drop tolerance* $\tau > 0$ and during the computation drop any entry of the (approximate) Cholesky factor that has absolute value smaller than τ - either absolutely, i.e., in the elimination of the i-th row (or column), a fill-in a_{ij} is accepted if and only if

$$|a_{ij}| \leqslant \tau,$$

or relative to the rest of the considered row or column, i.e., in the elimination of the i-th row (or column), the entry a_{ij} is discarded if and only if

$$|a_{ij}| \leqslant \tau \|A_{i,*}\|.$$

Instead of the row (or column) norm one can also use only the diagonal element norm, i.e.,

$$|a_{ij}| \leqslant \tau |a_{ii}|. \tag{1.5}$$

This approach can be backtracked back to paper of Jennings and Tuff [79] and have been later proven to be quite useful, especially for some classes of matrices, e.g., M-matrices[10]. This criterion can be coupled with requiring only some *fixed amount of fill-in in each row or column*. The resulting preconditioner can be quite powerful, but it requires two user-defined parameters that heavily affect the performance of the preconditioner.

A possibly better option than simply discard the dropped entries is to add their absolute value to the diagonal entry, arriving at the so-called *modified incomplete Cholesky factorization*. This can substantially improve the performance of the preconditioner in practice as well as allowing for some theoretical results, e.g., one can show in some cases that such incomplete factorization is breakdown-free.

Different possibility is to establish an incomplete Cholesky factorization theory (based on one of the previous ideas) only for certain class of matrices, for which there are promising theoretical results. For matrices outside the class, one can first take approximation $\hat{A}$ from the considered matrix class and then carry out the incomplete factorization $\bar{L}\bar{L}^T \approx \hat{A}$. This has been done for, e.g., M-matrices or H-matrices[11] (therefore also diagonally dominant matrices as a special case of H-matrices).

[10] Matrices that has all nondiagonal entries nonzero and their inverse exists and has all entries nonnegative.

[11] Matrices, for which there exists an M-matrix so that those two have equal diagonal elements and the off-diagonal elements of the H-matrix are given by the negative magnitude of the entries of the M-matrix.

1.4.2 Alternative approaches

All of the above mentioned approaches were based on *the classical Cholesky factorization computation, i.e., Algorithm 2, 3.* The difference is in the *dropping criterions inside the algorithm.* There are also qualitatively different approaches such as *robust inverse factorization* methods (RIF) based on the A-orthogonalization process (see [9, Chapter 3]), *sparse approximate inverse* methods based on directly approximating the inverse A^{-1} (see [9, Chapter 5]) or *Schur complement*-based methods that exploit a different formulation of Algorithm 2, namely the submatrix one (see [9, Chapter 5,6]).

The preconditioners need not to be constructed only in purely algebraic way. Search for *analytical* preconditioners for particular problems can usually outperform the general preconditioners, see [30]. In general, focusing on linking the algebraic preconditioner back to the original infinite-dimensional problem is a powerful tool for better analysis and understanding of the preconditioning process overall, see [61]. One can also use the mentioned *multigrid methods as preconditioners for the CG method.* The preconditioner is for this purpose generalized to *any routine or procedure that transforms the input vector u to an output vector v so that $Au \approx v$.* This routine is then called in each iteration of CG instead of the solver for the classical preconditioning, see [78].

A completely different idea is to use the complete Cholesky factorization as in Algorithm 2, but in *different precision*, i.e., computing the preconditioner using either single or even only half precision (eight and four valid digits respectively) and then use the result as a preconditioner in double or quadruple precision CG, see [15].

As one can see, the amount of new ideas is still increasing, although the idea of preconditioning is quite old[12]. This work will hopefully add another drop into the ocean by exploring a particular algebraic preconditioning technique, based on *data-sparsity* and *column-oriented Cholesky algorithm.* However, it is important to stress out that there is quite a lack of understanding of the effect of most of the currently available preconditioners. Detailed analysis of the techniques in terms of *efficiency of the preconditioned CG* is lacking or it is unsatisfactory in most cases, posing new challenges and questions.

As mentioned above, it is common for the complete sparse Cholesky factorization to *symmetrically permute A* prior to the incomplete factorizations in hope of minimizing the fill-in or to achieve a particular structure of the factor. However, the interaction of the reordering and the incomplete factorization is neither easy nor a well-understood matter. In many cases the reordering has a *devastating effect on the performance of the resulting preconditioner in comparison to the one without the prior reordering.* This phenomenon is characteristic, for example, for the fill-in minimizing reorderings coupled with zero-level fill-in incomplete Cholesky, see, e.g., the paper of Duff and Meurant [24]. Indeed, minimizing the amount of the fill-in intuitively means that each of the present fill-in entries carries more information. Consequently, dropping those may be more harmful than dropping more entries in the original ordering of the matrix. For more detailed discussion see [9, Section 3.3].

[12] In [9], Benzi ties the origins of the concept of preconditioning to Jacobi and his paper from 1845. The term *preconditioning* in the contemporary sense was according to Benzi used first Evans in 1968.

2. The need for approximation

The previous chapter recapitulated several well-known methods for tackling systems of linear algebraic equations, including the preconditioned CG method. As we have already mentioned, the goal of this work is to combine the techniques used in data-sparsity and structurally sparse preconditioning together, hopeful to obtain a preconditioner that inherits the positive features of both. To do that, an overview of techniques that are commonly used to handle the data-sparse matrices is in order. Since the data-sparsity and the blockwise rank-deficiency theory are not quite as well established as the classical structural sparsity theory, a separate chapter will be devoted to that. This chapter will first comment on important features of the area - the role of blocks and hierarchy in the methods - and then particular methods and techniques of low-rank approximations and hierarchical and non-hierarchical matrix formats will be described with more details.

2.1 The role of blocks

The idea of *block formulation* of the classical factorizations, i.e., idea of working with *submatrices* instead of scalars, is much older than the data-sparsity area and therefore the motivation was originally not the the low-rank approximation. For the Cholesky, QR as well as LU factorizations have been proposed blocked versions, see [21]. There are several reasons why these versions have been becoming increasingly more popular in practice.

It is key to realize that the numerical operations of a given implementation are *not the only time-consuming part of the computation*. The data movement and assignment during the actual computation also requires a certain amount of time. As of now, the *communication* and *data movement* often pose a significant bottleneck bottleneck of the computations of large scale problems (see, e.g., [14]). At the present time, it is much more efficient to compute with matrices than with scalars because of *data movements* and *memory assignments*. Arithmetic operations with matrix blocks instead of scalars require more computational time and therefore the bottleneck of *fetching* and *dispatching the data* can be somewhat relaxed. This comes hand in hand with parallelization of the computers and foremost the *parallelization of the algorithms and the need for their reformulation*. In order to obtain efficiency, matrix blocks have to fit into the fast local memory of the processors, i.e., the blocks have to be of *moderately small size and structurally dense*. Without going into more details of (high-performance) computer architecture and parallelization, this has been an important argument for developing block variants of the known algorithms. Another observed phenomenon that advocates the block algorithms is their *numerical stability*. The block versions of basic factorizations as well as of other important procedures or algorithms have experimentally proven to be often more stable, making them preferable also in this direction.

This being said, one still needs to strive for *efficient* block algorithms. From the point of view of techniques dealing with structurally sparse matrices, one usually has to apply some (possibly symmetric) row and column reordering in order to establish or enhance some block structure with *dense* blocks (in the

sense of only small number of zero elements) inside the matrix. A particularly well-known and widely used example is the *multifrontal method* of Duff and Reid (see [25]) for structurally sparse matrices and the theory and usage of *supernodes* (not exclusively) in this method and other sparse decompositions (see [32, Section 6] and also [60] and the work cited there). However, it should be pointed out that the multifrontal method is usually used as a *direct method* rather than a preconditioning technique. Other blocking techniques have been proposed as well, see, e.g., [69] and also [8, Section 3.4] and work cited there. Once the matrix is permuted, the blocks often allow for close to peak level performances and may make the process more robust as pointed out above.

The techniques working with blockwise rank deficient matrices (possibly structurally dense) are centered around the blocks mainly because they approximate the blocks by low-rank decompositions. This is usually captured by *specific matrix formats*, where the format corresponds to a certain blocking of the matrix such that ranks of the blocks are uniformly bounded by some small constant $p_{\max}$. This further enhances the advantages mentioned for the structurally sparse matrices to a considerable degree, see, e.g. [46]. At the same time one automatically introduces an *approximation error* (although possibly small), which makes the solution of the new system always only an approximation to the original one. Consequently, these techniques are often used only as preconditioners or possibly direct methods in cases where the required accuracy is not too high. Also, the blocking techniques are often more complex and *case dependent* as they have to fetch not only dense but also *rank deficient* blocks.

2.2 The role of hierarchy

The data-sparse or blockwise low-rank class of matrices have attracted a lot of attention also thanks to the concept of the *hierarchical techniques*, e.g., $\mathcal{H}$-matrix and $\mathcal{H}^2$-matrix formats[1]. However, the idea of incorporating hierarchy and recursion into the model is again much older.

There are many examples of methods working with structurally sparse matrices and using hierarchy or recursion. Some of them are not strongly related to the preconditioning, e.g., multigrid methods [13], some other are often used as preprocessing for either direct method or a preconditioner construction, e.g., nested dissection reordering [31] and some are designed to construct a preconditioner, e.g., the multilevel ILU preconditioners, see, e.g., [72] for brief overview and more elaborate references in Section 1.4. The overall goal is often to introduce as much parallelism into the computation as possible and reduce the communication necessary by making it hierarchical.

As was already pointed out, the data-sparse techniques are usually *based on* (or at least very tightly tied to) the *hierarchy and recursion*, aiming for similar goals as above, i.e., parallelism and reduction in the communication. This has been quite successful in many cases as the hierarchy and the blockwise rank deficiency can together arrange for *almost linear*[2] complexities for the basic operations. Due to a rather specific structure of the used matrix formats, the

[1] The calligraphic $\mathcal{H}$ stands for *hierarchical*.

[2] *Almost linear* linear complexity refers to complexity of $\mathcal{O}(n \log^{\alpha}(n))$ for some small α.

techniques to compute a preconditioner for a matrix in such a format are almost all *recurrent*, utilizing, e.g., the Schur complement formulation of the Cholesky factorization , see [83], or [56] and related works mentioned there, but other options are available as well, see [43] and [42] for recurrent schemes based on the row-oriented Cholesky scheme. Simultaneously with the theory, the development in the implementation and parallelization took place as well. That resulted in number of libraries that focus on particular hierarchical structures in matrices, e.g., STRUMPACK library by Chysels et al. (see [34] and also [35] and [68]) and also $\mathcal{H}$-Lib by Kriemann et al. (see [56]).

As one can see, hierarchical preconditioners (not necessarily the data-sparse ones) often incorporate incompleteness by inexact or different arithmetic or by applying the "classical incomplete techniques" (e.g. threshold dropping) at the bottom level of the hierarchy. Analysis of the preconditioners has consequently become quite challenging, even in exact arithmetic, as a result of the *non-linearity of the scheme*. Let us also emphasize that the hierarchy and recursion make the methods qualitatively different from the common preconditioning techniques based on either column- or row-oriented Cholesky factorization.

In some cases, the hierarchy and recursion can be omitted, emphasizing the blockwise low-rank setting. This is the approach we want to explore here. Some work in this direction has been done already, e.g., low-rank updates in the multifrontal method, see [3], but overall this branch of methods offers still a lot of challenges.

2.3 Algebraic low-rank decompositions and approximations

There are many obvious instances, where one can exploit algebraic low-rank approximations, e.g., numerical range approximation [47], approximation of dominant eigenpairs or singular triplets [47], or, in general, time and memory costs reduction [6]. The favourable interpretation of these methods can differ from one application to another to some extent. On the other hand, the way the methods achieve the goal can differ a lot from applying suitably changed versions of basic algorithms (truncated SVD, RRQR and RRLU) through applying heuristic models (ACA, ACA+) to non-deterministic approaches that bring randomness by employing results from the probability theory (randomized techniques). Naturally, accuracy, robustness and time/memory costs vary from one approach to another and they are the deciding factors when it comes to choosing a particular method for a given setting.

Rank-revealing decompositions Methods in this group have been developed from the previously mentioned (basic) decompositions (SVD, QR and LU) to reveal the *numerical rank* of a given matrix. The simplest method to derive (and the most expensive one in terms of computation costs) is the *truncated SVD*. This method is a trivial consequence of the Young-Minsky Theorem (see Theorem 1.1.1) about the best p-rank approximation of a given matrix. The *truncated SVD* corresponds to approximating the matrix $A = U\Sigma V^T$ by $U\Sigma_p V^T$ as in Theorem 1.1.1. The rank p of the approximation is set by some problem-

dependent accuracy threshold ε so that $\sigma_p >> \sigma_{p+1} \approx \varepsilon$ or possibly $\sigma_{p+1}/\sigma_1 \approx \varepsilon$. The obvious disadvantage of this approach is the number of operations required, which for general m-by-n matrix amounts to $\mathcal{O}(mn\min(m,n))$. This might be unimportant for small problems (e.g., coming out of discrete inverse problems, see [48, Chapters 1-3]). But for many problems, this cost makes the truncated SVD often *not feasible*. Nonetheless, it stays to be an essential tool for the *error analysis* of other low-rank methods.

It has been mentioned that the two other decompositions can also reveal the numerical rank. If the matrix is exactly rank-deficient, one could arrange its columns (and rows) so that the bottom-right-most block of R (in the QR factorization) or U (in the LU factorization) is zero. When it comes to only *numerically* rank-deficient matrices, a natural idea is that the matrices R and U should have the bottom-right-most block only *nearly zero* compared to its upper-left-most square block. Algorithms to obtain such decompositions are called *rank-revealing*, i.e., one gets *rank-revealing QR* (RRQR) and *rank-revealing LU* (RRLU) factorizations. The above idea can be formulated precisely. For example, RRQR corresponds to the following scheme.

Lemma 2.3.1 (RRQR, [52, Lemma 1.2])**.** *Consider an m-by-n matrix A ($m \geqslant n$) and its QR-decomposition $A = QR$ with*

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix},$$

with R_{11} being square p-by-p matrix. If $\sigma_{\min}(R_{11}) >> \|R_{22}\| \approx \varepsilon$, then A has numerical rank p.

The essence of the rank-revealing algorithms lies in *finding a suitable permutation matrix* (or matrices) so that the classic algorithm (either QR or LU) applied to the *permuted matrix* will end up with the bottom-right-most block having a small norm in the sense above. Unfortunately, both RRQR and RRLU are computationally demanding, requiring asymptotically $\mathcal{O}(mn\min(m,n))$ operations. The algorithms and their analysis were developed around 1990's and for more details see, e.g., [16], [52], [65], [45] but also [38] and the references cited there.

CUR or pseudo-skeletal decomposition The pseudo-skeletal decomposition (also known as CUR decomposition) uses a different approach. Assume that an m-by-n matrix A has rank p and that its first p columns and also its first p rows are linearly independent. Let A be partitioned as

$$A = \begin{bmatrix} U & R \\ C & M \end{bmatrix}, \tag{2.1}$$

with U being a regular p-by-p square matrix. Then it is easy to show that in fact $A = \begin{bmatrix} U \\ C \end{bmatrix} U^{-1} \begin{bmatrix} U & R \end{bmatrix}$ (see, e.g., [2, p.8 (2.9-2.11)]). This group of methods is based on the following extension of this result for the *numerical* rank by Goreinov, Tyrtyshnikov and Zamarashkin. Its statement follows.

Lemma 2.3.2 (CUR, [41, Theorem 3.1 and Corollary 3.1])**.** *Consider an m-by-n matrix A and some $\varepsilon > 0$. Assuming that the ε numerical rank of A is p, then there exist p columns of A*
forming an m-by-p matrix C, p rows of A forming an p-by-n matrix R
and a square p-by-p matrix G such that

$$\|A - CGR\| \leqslant \varepsilon \left(1 + \sqrt{pn} + \sqrt{pm}\right).$$

However, the theorem does not give a computational way how to find the matrices C, R and G. Some results in this direction are known, aiming at choosing the columns and rows so that U has as large absolute value of the determinant as possible. The problem of choosing such rows and columns is an NP-hard problem (see [18]). Consequently, some greedy approaches are following this direction with mixed results, see, e.g., [7, Section 2, Lemma 2 - 5] or [80, Section 4]. In the end, the most popular methods are based on heuristics derived from the result $A = CU^{-1}R$ (in case $\operatorname{rank}(A) = p$). The basic method is called *cross approximation* (CA) and proceeds as follows.

Assuming we have already constructed the matrices C, U and R of dimensions m-by-k, k-by-k and k-by-n respectively as in 2.1, the question is which row and column to include into the current approximation next. The CA method first finds the largest entry (in absolute value) in the remaining part of A (columns and rows that have not been added yet) and then adds the column and row determined by this entry to C and R and update the middle matrix $G = U^{-1}$, where U is again formed by the mutual elements of the considered rows and columns as in (2.1). As for the implementation, one does not assemble G, but only updates the matrix U and then solves a linear systems instead of applying G. The procedure is repeated until either the required accuracy is reached or until the number of columns (rank of the approximation) has reached some fixed a-priori given bound p_{max}. The CA method can be linked to the classical LU factorization, see [6, Algorithm 1, p. 576].

The search for the largest entry is computationally demanding, which leads to modifications that are cheaper, but with no performance gurantees. Among the best-known variations are adaptive cross approximation (ACA) and improved adaptive cross approximation (ACA+). The ACA method searches for the largest entry only in a random subset of the columns and adaptively determines the rank p based on some a-posteriori bounds on the approximation error. The ACA+ method uses a slightly more sophisticated way of determining the starting column subset. Although no guarantee is here in general (regarding the approximation quality), the counterweight is the low number of operations needed. For the above mentioned methods, one can find simple examples that shows $\mathcal{O}(mn)$ operations is necessary (see [12, Chapter 4]), but in many practical cases one can obtain good results after $\mathcal{O}(p^2(m+n))$ operations. For more detailed overview one can see, e.g., [2, Section 2.1.1], [12, Chapter 4] and also the work of Bebendorf and his colleagues [6] and [7].

Iterative approaches Not surprisingly, iterative methods have been considered to provide a low rank approximation as well, although they are not as popular at the moment. The summary for this class of methods in the overview paper of Kressner and Ballani [5, Section 2.3, p.9] is the following. Given an n-by-n

matrix A, one can apply the first k steps of Golub-Kahan-Lanczos iterative bidiagonalization process [40, Sect. 9.3.3, p.495] to obtain an k-by-k upper bidiagonal matrix B_k and p-by-n matrices U_k, V_k with orthonormal columns such that

$$AV_k = U_k B_k.$$

Then one can compute the *truncated singular value decomposition* $B_k \approx \widehat{U}\widehat{\Sigma}_p\widehat{V}^T$ and form the rank p approximation A_p of A as

$$A_p = U_k\widehat{U}\widehat{\Sigma}_p\widehat{V}^T V_k^T.$$

For further details, including error bounds and numerical examples, one can see the above mentioned work of Ballani and Kressner. A similar approach can be utilized also for other projection processes, e.g., Lanczos or Arnoldi (see [58, Chapter 2]). Although the iterative approaches are not as widely used as some other approaches, the idea of first projecting the matrix to reduce the dimension first is a useful one.

Randomized techniques The use of randomness in linear algebra has become increasingly popular, as it can significantly improve the numerical stability and robustness in many cases. The progress in the field was nicely summarized by Halko, Martinsson and Tropp [47] in 2011. Here one can find most of the important results in one place, supplemented by necessary theory and numerical examples.

The basic idea is based on approximating the numerical range of a square n-by-n matrix A and then projecting A onto this space, see [47, Proto-algorithm, Section 1.3, p. 224]. First, one takes a *random Gaussian matrix*[3] Ω and computes $Y = A\Omega$. This is followed by the QR factorization $Y = QR$, i.e., extraction of the orthonormal base of the range of Y, and finally one forms the approximation $\widetilde{A} := QQ^TA$, i.e., orthogonal projection of A onto the range of Y. Using the scheme of Figure 1.1, $\widetilde{A}$ is in a low-rank format with $U = Q$ and $V^T = Q^TA$. Consequently, if one aims for p rank approximation, then it should hold $\mathrm{rank}(Y) = \mathrm{rank}(\Omega) \approx p$. Randomized approximation of the SVD is also available by computing SVD of $Q^TA = \widehat{U}\widehat{\Sigma}\widehat{V}^T$ and setting $\widetilde{A} \approx Q\widehat{U}\widehat{\Sigma}\widehat{V}^T$, see [47, Section 1.5, p. 226].

In practice one almost always uses so-called *oversampling* (see, e.g., [47, Theorem 1.1, p. 225]), i.e., aiming at rank p approximation, one takes Ω as an n-by-$(p+k)$ gaussian matrix, where k is the *oversampling factor* (usually $k = 5 \sim 10$). This oversampling can dramatically improve the performance, e.g, increase the probability, with which the result is very close to the solution. The performance can be further improved by applying A (or A^TA for nonsymmetrical case) to Ω multiple times, with possible reorthogonalization in between, building on results of the power method and QR algorithm (see , see [47, Section 1.5, (1.8 - 1.9), p. 225]). In general, one could also use not only powers of those matrices but *polynomials in* A or A^TA. Overall, randomized techniques seem to be very robust and also relatively modest in terms of time complexity, see [47, Section 1.4.1, p. 224]. For more details, one can see not only the mentioned work of Halko, Martinsson and Tropp, but also the references therein.

[3] Gaussian matrices are such that have all their entries identically independently distributed corresponding to the Gaussian (normal) distribution

2.4 Data-sparse block matrix formats

The second group of methods - *Techniques of effective discretization and data storage* - consists of matrix block-partitioning schemes that enable exploitation of the *blockwise low-rank nature* of the matrix. The partitionings are either *hierarchical*, meaning that the block-partitioning has a certain recursively repeating structure (e.g., $\mathcal{H}, \mathcal{H}^2$-matrix formats), or *non-hierarchical or flat*, meaning that the blocks of the partitioning were not chosen in such a way (e.g., BLR format). The goal of these methods is to partition the matrix into blocks (possibly after a suitable permutation of the matrix in order to discover the block structure first) so that all of them have rank at most $p_{\max}$, where $p_{\max}$ is fixed in advance. This significantly reduces the memory requirements. It can also introduce new ways for parallelism and decrease time and memory costs even more.

Intuitively, in order to achieve efficiency, the original matrix has to be *data-sparse* in the first place, so that the introduced error is modest. And indeed, these techniques become *very efficient*, provided the structure they impose is already present in the matrix (or possibly in its factors or inverse, depending on what we will decide to approximate). To validate a particular blocking (in the sense of presence of the low-rank blocks), one often needs to return to the real-world problem, its infinite-dimensional model and the discretization scheme and analyse all of these together. For example, by neglecting (some of) the weak or inferior interactions inside the real-world model. A practical example is to neglect the diffusion effect far from the source or to neglect coulombic interaction of two charges far away from each other. Then one can obtain the rank-deficiency in some blocks of the matrix. This can be made more specific considering particular integral equations or PDE formulation of the problem and particular discretization scheme, e.g., Lagrangian or Gaussian quadrature rule or finite elements or finite differences scheme. Then one can prove useful statements for a certain blocking scheme in certain applications, see [12] or the already mentioned paper of Greengard and Rokhlin [44]. One can sometimes find the expression *analytical low-rank methods* depicting the particular discretization schemes that should yield a data-sparse matrix for classes of problems, see, e.g., [2, Section 2.1.2] or [12, Chapter 3 and 5]). Even though these techniques are often the key to effective usage of the formats and in many cases have been the motivation for their development, they will not be discussed here and the interested reader may look into the above mentioned works for references.

If there is no a-priori information about the matrix or the matrix come from completely different area, the *low-rank property can be on the matrix blocks imposed*. The price to pay is a hardly predictable approximation error, which may be quite large or at least definitely not negligible and can, and in many cases does, destroy the efficiency. On the other hand, as mentioned above, our aim is to use these techniques in the context of blockwise incomplete Cholesky preconditioners. Therefore, the question of *required accuracy* is perfectly valid and is yet to be fully explored. A different approach is to first compute the (blockwise) low-rank approximations to some desired accuracy,. This in turn yields the rank of the approximation $p_{\max}$ (not necessarily low), instead of imposing the rank a-priori and uniformly.

2.4.1 Hierarchical matrix formats

As the above preview suggested, this paragraph focuses on the hierarchical matrix block-partitionings. They can be introduced in many ways, but most of them can be viewed in the framework of the *cluster trees* and *block-cluster trees*. We will follow the exposition of the lecture notes of Börm, Grasedyck and Hackbusch, see [12, Chapter 1 and 2]. The general idea is to introduce hierarchical partitionings of the column and row index sets, which in turn introduces block partitioning of the matrix.

Starting with an n-by-n matrix A, one can define the rooted graph tree with vertices corresponding to *subintervals* of $\{1, \ldots, n\}$ of natural numbers[4] that posses a *hierarchical structure*, i.e., such that it holds that

- the root corresponds to $\{1, \ldots, n\}$;
- the union of the sons of a nonleaf vertex is equal to the vertex itself;
- the intersection of two vertices is empty if and only if one is not an ancestor of the other.

Such trees are called *labelled cluster trees* and two particular examples for $n = 8$ are given in Figures 2.1 and 2.2 below.

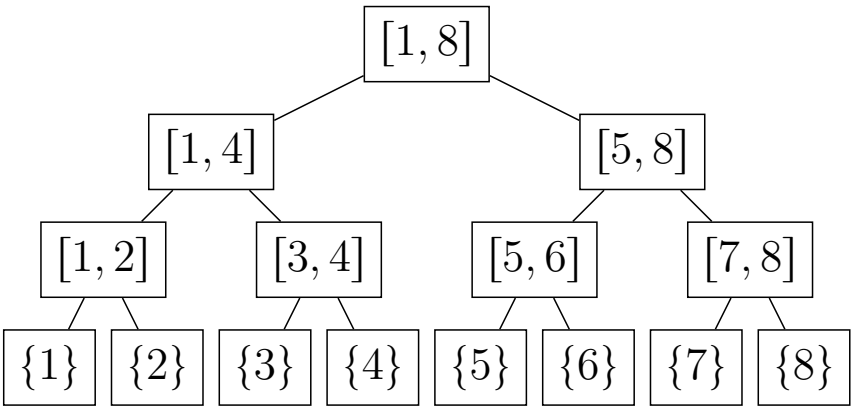

Figure 2.1: This figure shows the most natural label cluster tree $\mathcal{T}_1$ that corresponds to full binary tree for $n = 8$. The intervals in the vertices are to be understood in natural numbers, e.g., $[1, 3] \equiv \{1, 2, 3\}$, etc.

Given any labelled cluster tree $\mathcal{T}$, one can construct a *block-cluster tree*, a tree with vertices consisting of product of two index sets (note that product of two index sets corresponds to some particular block in the original matrix). The idea is to take the *direct product tree* $\mathcal{T} \times \mathcal{T}$, i.e., tree with root $\{1, \ldots, n\}^2$, in which sons of each nonleaf vertex $t \times s$ are given by product of sons of t and s in the original cluster tree $\mathcal{T}$. Having a block cluster tree, one can *adopt the matrix partitioning induced by the leaf vertices*, i.e., partition the matrix to blocks given by the leaves, see Figures 2.3-2.4 and Figures 2.5-2.6. Due to the hierarchical demands above, such partitioning of $\{1, \ldots, n\}^2$ is always disjoint, i.e., the blocks in the matrix partitioning do not overlap. However, this product might have large

[4] An *interval of natural numbers* is set of consecutive natural number, e.g., $[1, 3] = \{1, 2, 3\}$.

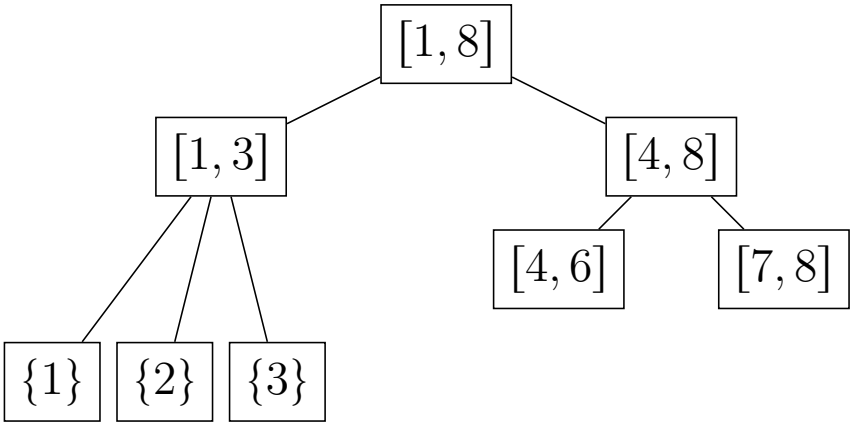

Figure 2.2: This figure gives an example of a less intuitive labelled cluster $\mathcal{T}_2$ tree (yet perfectly valid with respect to the above conditions). The intervals in the vertices are to be understood in natural numbers, e.g., $[1,3] \equiv \{1,2,3\}$, etc.

number of leaves and the resulting matrix block-partitioning might be *too fine* in the sense of yielding too small blocks. This could deny the rank deficiency if, e.g., most of the blocks ware 2-by-2 or 3-by-3. To address this issue, one can introduce an *admissibility condition* and once a vertex $t \times s \in \mathcal{T} \times \mathcal{T}$ meets the condition, its sons are discarded and the vertex is proclaimed a leaf (the vertex is then said to be *accepted* or *admissible*). Such procedure creates the *block-cluster tree* induced by $\mathcal{T}$ and the given admissibility condition. In this way one can further coarsen (or refine) the matrix blocking, while keeping the same tree structure in the background. A simple example of an admissibility condition is captured in the mentioned figures as well. Here a vertex in $\mathcal{T} \times \mathcal{T}$ is accepted if and only if the index sets do not overlap.

Let us emphasize the difference between the blocks that do not overlap in the matrix - general property due to the definition - and the two index sets *forming the vertex in block-cluster tree that do not overlap - this, in particular, means that the accepted blocks will be either strictly below or strictly above the diagonal.* Note that without any condition, $\mathcal{T} \times \mathcal{T}$ would have in total $4^3 = 64$ leaf vertices, whereas the examples below posses only 22 and 16 leaf vertices, respectively.

In practice, the admissibility condition usually compare the sizes of index sets and their distance. Classical examples are

- **strong admissibility condition** - vertex $\overbrace{\{i,\ldots,j\}}^{t} \times \overbrace{\{k,\ldots,l\}}^{s} \equiv t \times s$ meets the strong admissibility condition, provided that

$$\max\{\mathrm{diam}(t), \mathrm{diam}(s)\} \leqslant 2\mathrm{dist}(t,s); \tag{2.2}$$

- **weak admissibility condition** - vertex $\overbrace{\{i,\ldots,j\}}^{t} \times \overbrace{\{k,\ldots,l\}}^{s} \equiv t \times s$ meets the weak admissibility condition, provided that

$$\min\{\mathrm{diam}(t), \mathrm{diam}(s)\} \leqslant 2\mathrm{dist}(t,s).$$

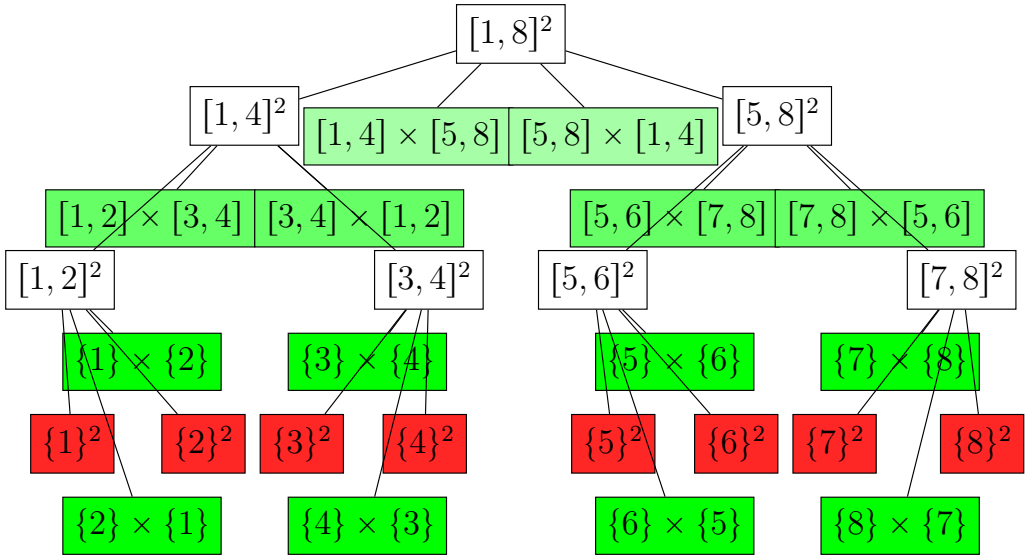

Figure 2.3: The figure gives the block-cluster tree corresponding to the product $\mathcal{T}_1 \times \mathcal{T}_1$ ($\mathcal{T}_1$ taken as in Figure 2.1) equipped with the "non-overlapping" admissibility condition. The corresponding blocking of the matrix is in Figure 2.4. The leaves are in different shades of green based on what tree level they were accepted or they are red if they were not accepted (so called *inadmissible leaves)* but couldn't be partitioned any further. Non-leaf vertices are blank. The coloring matches with the partitioning in Figure 2.4.

In other words, the blocks needs to be either far away from diagonal (i.e. t and s are far from each other) or the blocks are small (i.e. t and s are small). Example of block-cluster tree induced by the cluster tree $\mathcal{T}_1$ and the weak admissibility condition is shown in Figure 2.7 and the corresponding matrix partitioning is given in Figure 2.8.

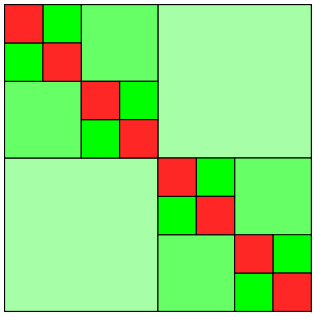

Figure 2.4: The figure gives the matrix blocking according to the block-cluster tree in Figure 2.3. The coloring corresponds to the one in Figure 2.3 as well. To have $\mathcal{H}$-matrix format, each of the blocks in the scheme has to have rank at most p. The red ones are usually small enough so that their rank is automatically smaller than p (in this case they are 1-by-1, i.e., scalars).

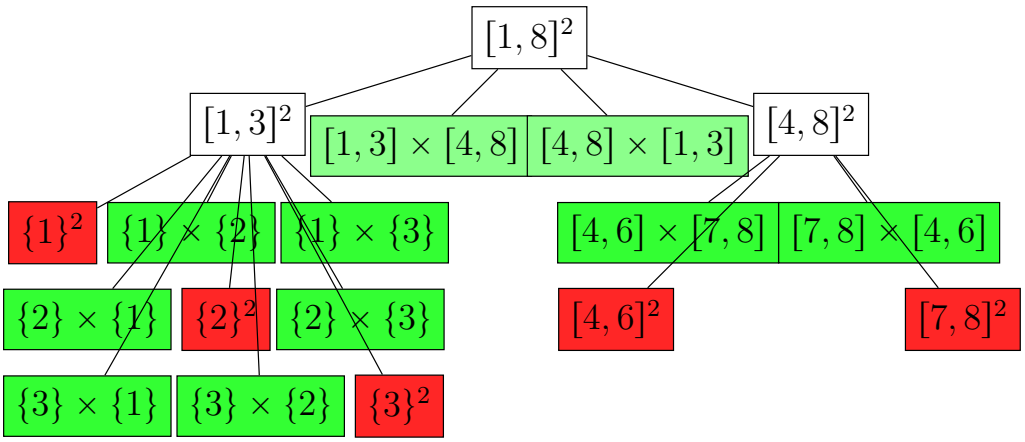

Figure 2.5: The figure gives the block-cluster tree corresponding to the product $\mathcal{T}_2 \times \mathcal{T}_2$ ($\mathcal{T}_2$ taken as in Figure 2.2) equipped with the "non-overlapping" admissibility condition. The corresponding blocking of the matrix is in Figure 2.6. The leaves are in different shades of green based on what tree level they were accepted or they are red if they were not accepted (so called *inadmissible leaves)* but couldn't be partitioned any further. Non-leaf vertices are blank. The coloring matches with the partitioning in Figure 2.6.

Note that for Figures 2.3-2.4 and 2.7-2.8 we have used the same cluster tree, but the partitioning is different, because the admissibility condition was different. Also, one does not need to use the same cluster tree for partitioning of the row and column index sets. However, those have to be *compatible* with each other in some sense. For more details see [12, Chapter 2]. Another possible generalization is to consider also rectangular matrices, i.e., matrices with different column and row index sets. Having a blocking based on some block-cluster tree and an admissibility condition, one can add the condition that each of the matrix blocks has rank at most $p_{\max}$, one obtains the $\mathcal{H}$-matrix format.

Having two $\mathcal{H}$-matrices induced by the same block-cluster tree and with the same bound $p_{\max}$ on the maximal rank of the induced matrix block, one can consider also their sum, product, inverse and possibly also their basic decompositions. It is easy to see that these operations may and often will violate the

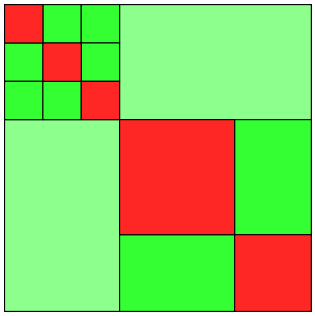

Figure 2.6: The figure gives the matrix blocking according to the block-cluster tree in Figure 2.5. The coloring corresponds to the one in Figure 2.5 as well. To have $\mathcal{H}$-matrix format, each of the blocks in the scheme has to have rank at most p. The red ones are usually small enough so that their rank is automatically smaller than p (in this case they are 1-by-1, i.e., scalars).

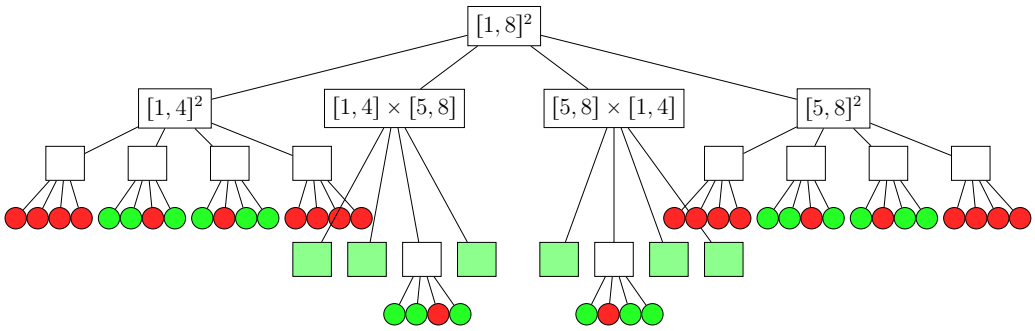

Figure 2.7: The block-cluster tree induced by the product tree $\mathcal{T}_1 \times \mathcal{T}_1$ equipped with the *weak admissibility condition*. Comparing with Figure 2.3, different admissibility condition can alter the block-cluster tree and also the number of the leaf vertices substantially. The coloring is analogous as above. The labelling of the vertices of the tree from the second level on is omitted, but it is obvious based on the block-cluster tree from Figure 2.3. The only difference here is that the admissibility condition does not accept the same vertices. The vertices themselves are identical.

blockwise rank bound $p_{\max}$ and thereby the result may no longer be in the same $\mathcal{H}$-matrix class. However, one can project the result back to the set of $\mathcal{H}$-matrices given by the considered block-cluster tree and original $p_{\max}$. In this way one can introduce entire $\mathcal{H}$-arithmetic based on computing in the classical way and then projecting the blocks back to the class of $p_{\max}$-rank matrices. It is, in general, not equivalent to the classical one, but it can be useful anyway. *The main advantage is the reduction of the time and memory costs* that asymptotically amounts to $\mathcal{O}(n \log(n)^2)$. Results from such arithmetic can be consequently used as, e.g., cheaply available preconditioners as already we have already touched upon (see, e.g., [43], [42] or [56]). The rest of the formats used in practice can be viewed as a particular instance of $\mathcal{H}$-matrices.

One of them has been already indirectly introduced - the hierarchical off-diagonal low-rank (HODLR) format. It corresponds to Figure 2.4, i.e., the block-cluster tree is product of a complete balanced binary cluster tree with itself when all off-diagonal blocks are accepted (i.e., "non-overlapping" admissibility condi-

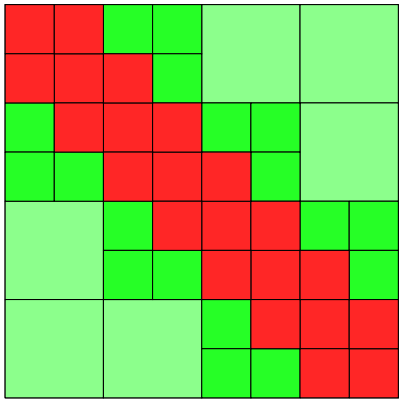

Figure 2.8: The figure shows the matrix partitioning given by the block-cluster tree from Figure 2.7. The green blocks correspond to accepted leaf vertices of the block-cluster tree (shade implies, on which level those vertices were accepted and became leaves), whereas the red ones correspond to inadmissible leaf vertices of the tree. To get the $\mathcal{H}$-matrix format, each of the blocks in the scheme has to have rank at most p. The red ones are usually small enough so that their rank is automatically smaller than p.

tion is used).

The other commonly used block formats are $\mathcal{H}^2$-matrices and *hierarchical semi-separable* format (HSS), both of which *further strengthen the hierarchy*. Each vertex $t \times s \in \mathcal{T} \times \mathcal{T}$ can be associated with a block of the matrix - let us denote that block by $A|_{t\times s}$. Assume that $t, s \in \mathcal{T}$ are a non-leaf vertices and assume that they have sons t_1, T_2 and s_1, s_2 respectively (the generalization for more sons will be straight forward) and moreover assume that in $\mathcal{T} \times \mathcal{T}$ the vertices $t_1 \times s_1$ and $t_2 \times s_2$ are non-leaf vertices and $t_1 \times s_2$ and $t_2 \times s_1$ are leaf vertices (based on some particular admissibility condition). Therefore $A|_{t\times s}$ is rank deficient and it admits the low-rank decomposition

$$A|_{t\times s} = V_{t,s} S_{t,s} W_{t,s}^T. \tag{2.3}$$

For simplicity, we do not specify dimensions of the matrices. One can view the decomposition (2.3) as the truncated SVD. Consider the two additional conditions.

- The *basis matrices* $V_{t,s}$ and $W_{t,s}$ can be written as $V_{t,s} = V_t$ and $W_{t,s} = W_s$ and they have *orthonormal columns*;
- There exist *basis transformation matrices* T_t and U_s such that

$$T_t \equiv \begin{bmatrix} T_{t_1} \\ T_{t_2} \end{bmatrix} \quad \text{such that} \quad V_t = \begin{bmatrix} V_{t_1} & 0 \\ 0 & V_{t_2} \end{bmatrix} \begin{bmatrix} T_{t_1} \\ T_{t_2} \end{bmatrix},$$
$$U_s \equiv \begin{bmatrix} U_{s_1} \\ U_{s_2} \end{bmatrix} \quad \text{such that} \quad W_s = \begin{bmatrix} W_{s_1} & 0 \\ 0 & W_{s_2} \end{bmatrix} \begin{bmatrix} U_{s_1} \\ U_{s_2} \end{bmatrix}.$$

If all of the above holds *for each non-leaf vertex* $t \times s \in \mathcal{T} \times \mathcal{T}$ we obtain the $\mathcal{H}^2$-matrix format (in general) or the hierarchical semi-separable (HSS) format(for particular choice of admissibility condition and block-cluster tree). The main

advantage is in *an additional decrease in the memory costs*, since to contain all of the blocks, one only need to store the "*middle* matrices"(which should be small, at most $p_{\max}$-by-$p_{\max}$) and the *transformation matrices* in contrast to the whole low-rank decomposition in the case of $\mathcal{H}$-matrix and HODLR formats. Because the general definition requires considerably more effort we will not present it here and the interested reader may consider, e.g., [12, Chapter 9].

It should be emphasized that it is not possible to reformulate the classical row- or column-oriented Cholesky factorization for all of the above mentioned block formats in the naive way to utilize the blocks (see Algorithm 2, 3) since the block dimensions do not allow for it. This opens up a lot of space for new research. Some of the currently used approaches to overcome this issue are summarized in the next chapter (Section 3.1 in particular), devoted to the data-sparse incomplete Cholesky factorization.

2.4.2 Non-hierarchical matrix formats

The general representative of these matrix formats is the *block low-rank* format (BLR). To be more specific, one says that A is in the BLR format with block partitioning

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{kk} \end{bmatrix},$$

provided that at least one of the blocks is a low-rank matrix[5]. Naturally, in practice one desires that most of them, if not all, to be of rank at most $p_{\max}$ for a suitably small $p_{\max}$. However, letting some blocks to be of the full rank allows considering also cases, which posses an almost nice structure. For example, considering

$$A = \begin{bmatrix} A_{11} & X_{12}^T Y_{12} & \cdots & X_{1k}^T Y_{1k} \\ X_{21}^T Y_{21} & A_{22} & \cdots & \vdots \\ \vdots & & \ddots & \vdots \\ X_{k1}^T Y_{k1} & \cdots & \cdots & A_{kk} \end{bmatrix},$$

A is in the BLR format, even though the diagonal blocks may be of considerable size with respect to the rest of the matrix and may have full rank. According to the definition there is generally no restriction on the blocks. However in order to perform the classical Cholesky algorithm (assuming A is SPD), one needs to assume the following.

B1 The blocking has compatible block-columns and block-rows, i.e., the number of columns in the blocks $A_{1j}, A_{2j}, \ldots, A_{pj}$ is constant and the same holds for the block-rows.

B2 The diagonal blocks are square.

For dense matrices, one can often consider a uniform blocking. But for structurally sparse matrices, this could be quite inefficient since one would have to work with a large amount of zero elements. Therefore, blocking a (structurally)

[5]This definition is taken from the work of Amestoy et. al. [3]

sparse matrix requires the blocks to be either completely zero or relatively densely populated, provided one would like to keep the benefits of structural sparsity also for the blocked matrix. Consequently, one usually has to first construct a suitable permutation matrix P so that PAP^T has only *moderate number of relatively small and dense blocks*. One of the first algorithms focusing on finding such reordering is the *graph compression algorithm* proposed by Ashcraft [4] for (at least structurally) symmetric matrices. The idea is to use the graph model of the symmetric matrix and characterize the vertices, for which the corresponding rows (and due to symmetry also columns) have the same nonzero structure after a convenient permutation. Permuting these symmetrically next to each other will result into PAP^T having only fully populated blocks and completely zero blocks. On the other hand, there is no guarantee for the size of these blocks, i.e., in the worst case scenario, those blocks could all be possibly 1-by-1, i.e., no blocking could be achieved at all.

From theoretical point of view, it is easy to see that two vertices are in the same class (i.e., the corresponding rows will have the identical structure) if and only if the sets of directly connected vertices[6] of these vertices are identical as well. However, to check this in the naive way is computationally not feasible. Ashcraft proposed a particular *hashing function* for the vertices, i.e., function that assigns a value (label) to each vertex and it is guaranteed that if two vertices have different labels, then also the corresponding rows has to have different structures. However, the opposite usually doesn't hold, i.e., it is not a bijection. The Ashcraft's hashing function sums the indices of the direct adjacent vertices (i.e., sums the column indices of the nonzeros in the given row). Ashcraft observed that this function can be evaluated quickly for all vertices, while at the same time, it is quite rare for two vertices to have the identical hashing value and to not have the identical set of the directly connected vertices. But it could happen, in general, and therefore all vertices with identical hashing value are then checked for the exact structure. The worst-case analysis shows that this might be quite demanding computationwise provided a lot of vertices has the same hashing value but different structure. But this is usually not the case in practice (see the numerical experiments in the Ashcraft's paper) and this algorithm has produced a blocking with quite low costs in many cases. The process is sketched below in the Figure 2.9 and 2.10.

$$A = \begin{bmatrix}
* & * & * & & * & & * & & * & \\
* & * & * & & * & * & * & * & * & * \\
* & * & * & & * & * & * & * & * & * \\
 & & & * & & & * & & * & * \\
* & * & * & & * & * & * & * & * & * \\
 & * & * & & * & * & & * & & \\
* & * & * & * & * & & * & & * & * \\
 & * & * & & * & * & & * & & \\
* & * & * & * & * & & * & & * & * \\
 & * & * & * & * & & * & & * & *
\end{bmatrix}, \qquad G(A) =$$

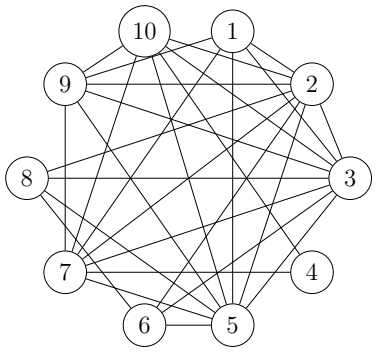

Figure 2.9: On the left is a given sparse matrix A that we would like to block. For that purpose we have to consider its graph model $G(A)$ (on the right).

[6]Two vertices are directly connected provided there is an edge connecting them.

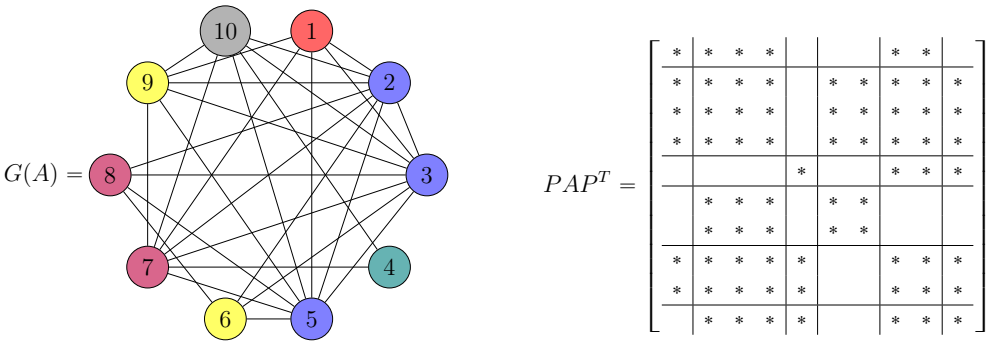

Figure 2.10: The graph compression algorithm of Ashcraft determines the grouping of the vertices of the graph $G(A)$ (the groups are here illustrated by colours) and then symmetrically permutes the rows corresponding to the vertices in one group next to each other. This results in block structure of the original matrix (on the right).

As pointed out at the beginning, the problem might arise when the approach gives blocks of *very small dimensions*, i.e., in the case that *the matrix itself does not allow for exact blocking*, i.e., no two rows have identical structure. The problem can be solved by allowing *some small amount of zeros inside the blocks* and hence allowing for *more* and possibly *larger* blocks. This has been proposed within the field of iterative methods by by, e.g., Saad in [69]. The idea behind the Saad's approach is to first use the Ashcraft's procedure to determine the dense blocks and then use another procedure on the top of the Ashcraft's to "block the blocks".

To describe this with more details, one can define *equivalence relation* in the graph $G(A)$ based on the Ashcraft's approach, i.e., two vertices are *equivalent* if and only if their sets of directly connected vertices coincide[7]. This relation naturally induces the *classes of equivalence* in the graph (rows corresponding to vertices in one class will be blocked together). If one *factor* the graph $G(A)$ with respect to this equivalence, then the result is the *quotient graph*, let us denote it $Q(A)$, see Figure 2.10 below.

Figure 2.11: Considering the same setting as in Figure 2.10, the quotient graph $Q(A)$ and its adjacency matrix are illustrated in this figure. The correspondence between the classes of equivalence and the vertices is illustrated in colours.

[7]It is easy to see that this relation si indeed equivalence, i.e., it is *reflexive, symmetric* and *transitive*.

The quotient graph represents the *block structure*, i.e., the block structure of PAP^T with the suitable permutation matrix P. To see this more clearly, one can assemble the *adjacency matrix C of the quotient graph $Q(A)$*, see Figure 2.12. The key idea of Saad is to *block the matrix C again*, but this time not requiring the nonzero blocks to be fully populated, i.e., allowing some small number of zeros inside the "nonzero"blocks. The blocking of the matrix C is based on a geometrical point of view. Row vectors u_i, u_j of C are grouped together, provided that one has

$$\frac{\langle u_i, u_j \rangle^2}{\|u_i\|^2 \|u_j\|^2} = \cos(\text{angle}(u_i, u_j)) \leqslant \tau, \tag{2.4}$$

where $\tau \in [0, 1]$ is a user-defined quantity, which controls (indirectly) the amount of the zeros in the nonzero blocks. Taking $\tau = 1$, one has the Ashcraft's algorithm only. Saad gave only a specific case analysis of the amount of the "zero fill-in", i.e., of the number of zero entries that will be treated as nonzeros, based on τ, but experimentally shown that this procedure can allow for much bigger blocks and hence can make the computation much more efficient. The results of the Saad's blocking algorithm for the matrix A from Figure 2.11 are given below in Figure 2.12.

One can see that finding a balance determined by τ might be a difficult task in general - too large τ are likely to result in a large number of blocks of size one, i.e., scalars on diagonal and vectors in the given block-row and block-column, whereas too small values of τ are inefficient from the point of view of the structural sparsity, i.e., forces one to compute with large amount of zero entries inside the nonzero blocks. Based on the numerical experiments in [69], values $\tau \in [0.5, 0.7]$ are suggested[8].

[8] In the paper the role of τ is slightly different - Saad compares in (2.4) with τ^2, i.e., one can write $\tau = \tau^2_{\text{Saad}}$.

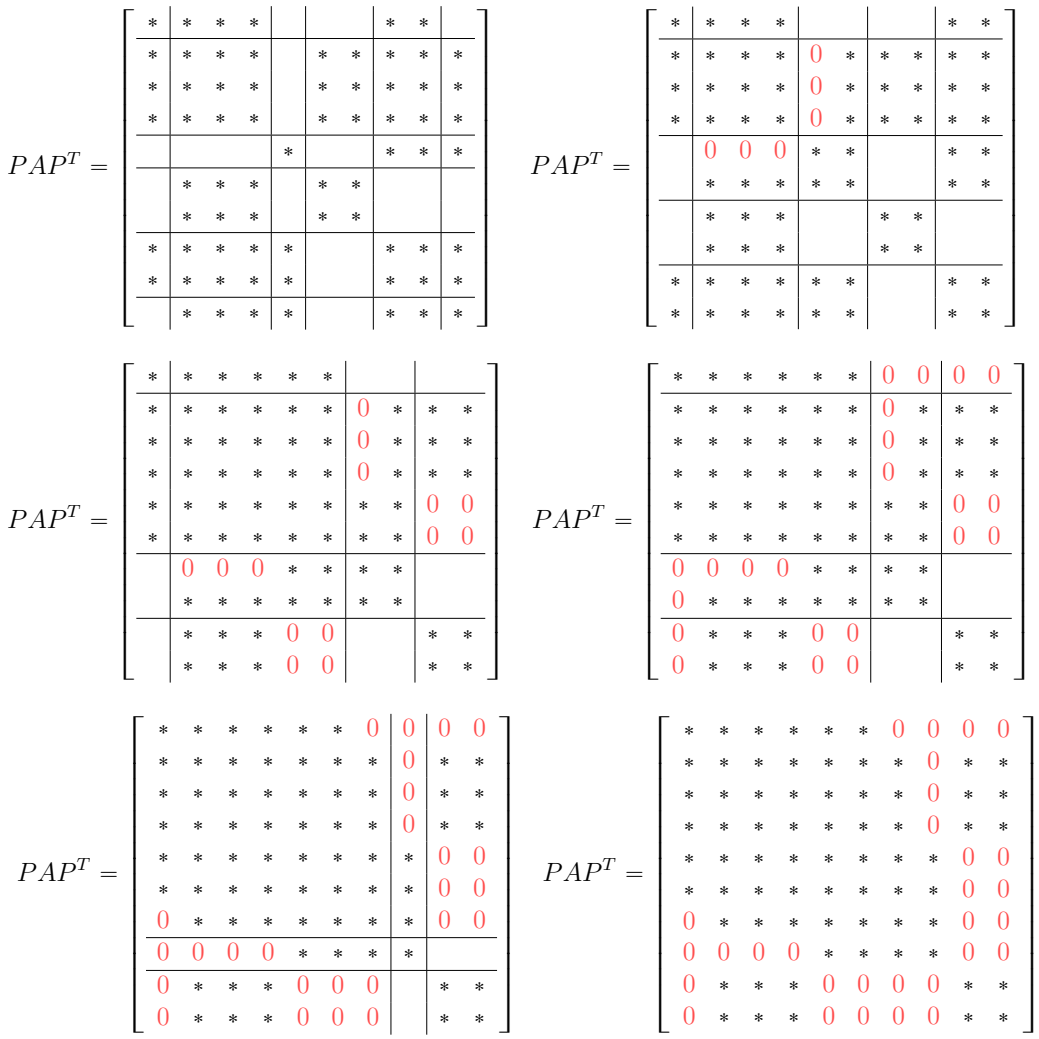

Figure 2.12: Considering the same matrix as in Figure 2.10, the above six matrices show blockings produced by the Saad's procedure for decreasing values of τ. From left to right and top to bottom the matrix blockings corresponds to τ values of $\{1, 0.9, 0.8\}$, $\{0.7\}$, $\{0.6\}$, $\{0.5, 0.4\}$, $\{0.3, 0.2\}$, $\{0.1, 0\}$. The zero elements that will be due to the blocking included into the nonzero blocks (and hence treated as nonzeros) are highlighted in red.

3. Towards incomplete data-sparse Cholesky factorization

The previous chapter was devoted to an overview of the matrix blocking techniques, the related matrix formats and also low-rank approximation techniques that can be utilized. However, a little attention has been devoted to the use of these techniques together in the context of the preconditioners. This chapter will address this topic in detail, first summarizing some of the currently used data-sparse preconditioning approaches in the Section 3.1 and then focus on the development of a new preconditioning technique.

3.1 Incomplete recursion-based Cholesky factorization

The data-sparse formats mentioned have been based on low-rank approximations of matrix blocks, either hierarchical or not. They differ not only in the construction, but, more importantly, also in the compatibility with the classical formulation of the basic methods, e.g., Cholesky factorization. The same applies also to QR and SVD. While the non-hierarchical formats admit simple reformulation of any of the basic algorithm formulations (column, row or submatrix oriented), the hierarchical formats require particularly specialized approaches. This has been already touched upon in the Section 2.2 and this results in nearly all cases into employing some *recursion* into the formulation of the Cholesky factorization. Consequently, this, in general, denies most of the classical ways to incorporate incompleteness, although for some cases one can still find a suitable analogue of the classical, sequential approach.

Given an n-by-n matrix, the recurrent reformulation first imposes an artificial blocking - usually of the 2-by-2 form

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \text{and hence} \quad L = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \tag{3.1}$$

or the 3-by-3 form, often coupled with *nested dissection reordering*, that results in

$$A = \begin{bmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \quad \text{and hence} \quad L = \begin{bmatrix} L_{11} & & \\ & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix}$$

see, e.g., [56], [42] or [83]. However, generalisations are possible as well, see, e.g., [43] or [27]. The factorization (and not only the reordering) is then performed *recurrently*. Therefore, in order to compute the Cholesky factorization of A, one has to compute the Cholesky factorization of A_{11} first and the recursion continues until the problem is so small that the complete Cholesky factorization can be used. After that, the procedure needs to emerge from the recursion, i.e., the

way to compute other blocks of the L factor needs to be prescribed. There are more options how to manage this and we will present here just the simplest one, using the *Schur complement* formulation. Other possibilities can be found in the previously mentioned work, [56] and [43] in particular.

Assuming the 2-by-2 blocking as in (3.1), then

$$\begin{aligned} \text{Solve } A_{11} = L_{11}L_{11}^T \text{ for } L_{11}; \\ \text{Solve } A_{12} = L_{11}L_{12}^T \text{ for } L_{12}; \\ \text{Solve } A_{22} - A_{12}^T A_{11}^{-1} A_{12} = L_{22}L_{22}^T \text{ for } L_{22}. \end{aligned} \tag{3.2}$$

Here the first and the last row in (3.2) represent similar problems of approximately half the dimension, depending on the particular 2-*by*-2 blocking. This observation is the key to the recursion, which continues until problems of much smaller dimensions that can be easily solved exactly, are reached. This, in theory, leads to the *exact* factorization.

In order to incorporate incompleteness, one has to modify the approach of (3.2). A common way to do so is to *use the $\mathcal{H}$-arithmetic* instead of the classical one. This corresponds to first evaluating the second line of (3.2) exactly and then truncation of the result to have rank at most $p_{\max}$. This setting is not well-suited for classical incomplete factorizations, e.g., using threshold dropping or level-of-fill-in. Here, the incompleteness is introduced mainly through the $\mathcal{H}$-arithmetic. This yields a *data-sparse* output (preconditioner), where not only the preconditioner computation is cheap but its application, typically via a triangular solver, is cheap as well. This is a "must have" feature for an efficient preconditioner in general.

Another remark is that the "imposed blocking"is usually chosen so that it couples well with the hierarchical matrix format, e.g., 2-by-2 blocking can work well together with HODLR or HSS formats due to their nature. In some cases one it may have an additional information about the problem, i.e., may be able to predict a suitable matrix format for the factor (see, e.g., [12, Chapter 3] or [30]) or the matrix itself. In that case, the blocking for the procedure (3.2) has to be *compatible* with the matrix format.

This brings into the play another feature that has been mentioned several times already - reordering of the system. One of the most used is the *nested dissection* reordering. The effect of the reordering in this case is, to the best of our knowledge, yet to be fully explored. As pointed out in the Section 2.4, the computational costs of this class of preconditioners are often much more appealing in contrast to the classical ones, provided $p_{\max}$ is not too large. On the other hand, analysis of efficiency of such preconditioners seems to be quite a difficult task. Although there are some results in this direction, see, e.g., [83], this field is not sufficiently explored yet and many of the proposed preconditioners have been studied only experimentally.

Bearing in mind the above short summary, the goal of the rest of this chapter is to propose an *alternative approach* to the recurrent one, i.e., propose a preconditioning procedure that

- could be used for both hierarchical and non-hierarchical formats;

- could utilize the classical preconditioning tools as well (see Section 1.4.1);
- could exploit the possible rank deficiency of the blocks.

To do this, consider the *structurally sparse incomplete Cholesky factorization* as the starting point in the next section.

3.2 Incomplete sequential column-oriented Cholesky factorization

Recalling the complete structurally sparse Cholesky factorization as in Algorithm 3, a classical way to introduce the incompleteness is by *threshold dropping*. However, it is important *when* the dropping is carried out during the computation. To clarify this, let us sketch the incomplete version of Algorithm 3. For simplicity we omit the preprocessing, i.e., we do not reorder the matrix prior to the procedure.

Algorithm 5 Incomplete structurally sparse column-oriented Cholesky factorization

Input: A square n-by-n SPD matrix A.

1: **function** PROTO_ICHOL
2: **for** $j = 2, \ldots, n$ **do**
3: Determine the row structure $Struct(\bar{L}_{*,j})$;
4: Determine the column structure $Struct(\bar{L}_{j,*})$;
5: $\bar{l}_{j-1,j-1} \leftarrow \sqrt{a_{j-1,j-1}}$;
6: **for** $s \in Struct(\bar{L}_{j:n,j})$ **do**
7: $\bar{l}_{s,j-1} \leftarrow \frac{\bar{l}_{s,j-1}}{\bar{l}_{j-1,j-1}}$;
8: **end for**
} **factorize_col**$(j-1)$
9: **for** $k \in Struct(\bar{L}_{j,1:j-1})$ **do**
10: **for** $i \in Struct(\bar{L}_{j:n,k})$ **do**
11: $a_{ij} \leftarrow a_{ij} - \bar{l}_{jk}\bar{l}_{ik}$;
12: **end for**
} **update_col_by_col**(j,k)
13: **end for**
14: Apply dropping to $A_{*,j}$;
15: Apply dropping to $\bar{L}_{j,*}$;
16: **end for**
17: Return the incomplete factor $\bar{L}$;
18: **end function**

Regarding the dropping, there are two qualitatively different places in the algorithm, where the dropping could be applied. First is in line 14 one can drop some

of the fill-in that occurs during the *update* by previous columns. This certainly *affects the following computation as well as the final factor since the updates by the j-th column will not be complete*. In the same sense one can alter the structures on lines 3 - 4 to *modify the future computation*. On the other hand, dropping on line 15 *does not have any effect on the further computation whatsoever and modifies the resulting factor only directly by omitting certain entries*. In this sense, the line 15 represents a *final polishing* and can be focused purely on the *direct improvement of the final factor*. The rest of the dropping has to account not only for that but also for the fact that it shapes the structure of the updates as well. Its poor choice might result in omitting important information in the updates and ruining the whole factor.

Note that in comparison to Algorithm 3 the structural part of the computation (lines 3 - 4) *has to be done on the fly*, since the algorithm has to account for the droppings and the reduced fill-in. The question of how to approach this also needs to be addressed.

The rest of the section will be devoted to the *structural factorization performed on the fly* - Section 3.3 and 3.4, i.e., lines 3 - 4. The issue of *classical dropping*, i.e., lines 14 - 15 will be considered in Section 3.5.

Notice that this can be easily reformulated also for blocks, as long as those blocks form block-rows and block-columns with diagonal blocks being square, see Section 2.4 and since the goal is to utilize the low-rank techniques, it would be definitely natural to do so. However, the center of this section will be elsewhere, i.e., in the structural part of the factorization. In order to make the exposition easier to follow, we will not work with the blocks (at least not explicitly yet). However, the reformulation for blocks can be done, assuming the blocking conditions (B1) and (B2) from the beginning of Subsection 2.4.2 - one can simply work with the *factor matrix* with respect to the particular blocking (see Figure 2.11) on the structural level.

3.3 Explicit search for the srtucture

Let $1 < j < n$ and consider the j-th step of the structurally sparse sequential column-oriented Cholesky factorization as in Algorithm 5. The key to an *efficient* structural computation (lines 3 - 4) is to use known facts and relations of the *complete* Cholesky factorization. A perfect example is the commonly used formulation of lines 3 - 4 in practice, summarized below. Because we will show multiple proposals of *how to compute the structural patterns*, we will number them, starting with S_1 below.

Algorithm 6 Structural factorization for Algorithm 5

Input: An n-by-n SPD matrix A and the first $j-1$ columns of the incomplete Choelsky factop $\bar{L}$.

Output: Row and column structures $(S_1)_{j,*}$ and $(S_1)_{*,j}$.

1: Set $(S_1)_{j,*} = Struct(\bar{L}_{j,*}) \equiv \{k \mid \bar{L}_{k,j} \neq 0\}$;

2: Set $(S_1)_{*,j} = \left[Struct(A_{*,j}) \quad \bigcup \quad \left(\bigcup_{i \in (S_1)_{j,*}, i \neq j} (S_1)_{*,i} \right) \right] \setminus \{1, \ldots, j-1\}$;

Notice that the above procedure uses the observation about *column structures replication* from the complete Cholesky factorization setting, see 1.4, to determine the column structures of the incomplete factor, i.e., it uses the relationship adopted from the complete Cholesky factorization.

To simplify the notation, let us define the *j-th active part of the (incomplete) Cholesky factor* as the submatrix $\bar{L}_{1:j-1,j:n}$. This is exactly the part of the factor that can contribute to the update of the j-th column. Note that the part "above it", i.e., the submatrix $\bar{L}_{1:j-1,1:j-1}$ has been already finalized and it is part of the final factor and the part "right to it", i.e., the submatrix $\bar{L}_{j+1:n,j+1:n}$ is yet to be processed. The missing part, i.e., the j-th column, is being updated in the j-th step and will be in the $(j+1)$-th active part of the factor. Also, let us stress out that the structure S_1 is not necessarily correlated with the final sparsity structure of the factor since it is determined *before the dropping is carried out*.

Returning to Algorithm 7, note that it requires access the j-th active part of the factor, i.e., one needs to store the active part of the factor *explicitly*. Assuming the factor is *sparse*, it has to be saved in a *compressed format* either by rows or columns (for more details on CSR and CSC formats see [33, Section 5.4]). Without loss of generality we will consider the CSC scheme. The problems that will be met would arise analogously for the CSR scheme. The column structures are easy to obtain but the j-th row structure has to be *computed*, because the factor is stored in CSC format. Naive approach, i.e., searching in the structure of the already computed columns for row index j at each iteration, would be inefficient and could noticeably increase the overall costs. Luckily, this can be circumvented by keeping a track of the first nonzero entries of the columns in the current active part of the factor. This requires one additional vector of increasing size (the number of columns in the active part of the factor is equal to j) for saving these and a search in the vector (and possible update, which is, however, cheap thanks to the CSC format). Those costs (time and memory) are usually considered modest as they are somewhat negligible and, consequently, this approach has been commonly used, see [26].

However, *the assumption of active part of the factor being explicitly available is not feasible in our case*. Dealing with blockwise low-rank matrices (or matrices that have rank deficient Cholesky factor) *one simply has to allow for the active part of the factor to be stored implicitly, i.e., in low-rank form. Otherwise the efficiency is lost*. This is not necessarily related to the hierarchical matrix formats only. Taking matrices, which have some blocks that correspond to function values of a smooth function with small derivatives, the blocks are perfect for low-rank approximation[1] and this would be possibly best handled by some non-hierarchical blocking with possibly only a portion of the blocks being rank deficient.

3.4 Implicit search for the structure

The problem we are facing now is that the classical approach for the incomplete structurally sparse Chlesky factorization is *not compatible with the blockwise low-rank formats*. Since the aim is to *approximate* the Cholesky factorization, we will still use the original relations for determining the structures, i.e., lines 3 - 4

[1] This can be observed from the Taylor expansion, see [12, Chapter 1].

in Algorithm 5 will be still computed by Algorithm 6. However, the way the structures inside are obtained has to be changed. This will be addressed in two phases : first, the *row structure* needs to be found, i.e., the line 1 in Algorithm 6 has to be reformulated in case that the active part is not available explicitly. Then the focus will be on obtaining the structures of the columns needed for the update on line 2 of Algorithm 6.

3.4.1 Row structure

Trying to find the row structures, we will utilize the same approach as previously. The key is to use relations for the desired quantities, row structure in this case, that hold true in the (complete) Cholesky factorization and at the same time are advantageous from the current point of view. The next lemma follows precisely in this direction.

Lemma 3.4.1. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Assuming $j-1$ steps of the Cholesky factorization in Algorithm 3 has been carried out, denote the $(j-1)$-th principal leading submatrix of the final Cholesky factor L by L_{j-1}. Then one has*

$$Struct(L_{j,*}) = \{j\} \ \cup \ Struct\left(L_{j-1}^{-1} A_{1:j-1,j}\right).$$

Proof. After $j-1$ steps of Algorithm 3 one can write the result in the matrix form as

$$L^{(j-1)} \dots L^{(1)} A = M^{(j-1)}, \tag{3.3}$$

as sketched in 1.1 in the first chapter. Note that any matrix $L^{(k)}$ can possibly have nonzeros only on diagonal and in the subdiagonal part of the k-th column. The matrix $M^{(j-1)}$ is guaranteed to have zeros in the subdiagonal part of all the first $j-1$ columns. Hence it is easy to see that the $(j-1)$-th active part of the final factor L has been already assembled, i.e., one can write

$$M_{1:j-1}^{(j-1)} = L_{1:n,1:j-1}^{T}.$$

However, one can also rewrite the left-hand side as follows

$$L_{j-1} = \left(L^{(j-1)} \dots L^{(1)}\right)^{-1} \quad \text{and hence} \quad L_{j-1}^{-1} = L^{(j-1)} \dots L^{(1)}.$$

Altogether, one has

$$L_{1:n,1:j-1}^{T} = \left(L_{j-1}^{-1} A\right)_{1:j-1,1:n}.$$

Focusing only on the j-th column on the right-hand side gives the j-th row on the left-hand side and the result follows. □

Although the above lemma is straightforward, it can be very useful, since it *determines the row structure without using the current active part of the (incomplete) Cholesky factor*. On the other hand, this formulation is still not efficient as it requires a triangular solver run per iteration in order to determine the row structure. However, this can be circumvented by realizing that one needs *only the structure*, i.e., *a symbolic triangular solver is enough.* In order to state the result,

it is useful to introduce a new notion - *directed graph*. It is a graph which is *oriented*, i.e., each its edge has a starting node and ending one. The directed graphs can be used to generalize the graph model of a matrix (see Subsection 1.2.1) also for *structurally unsymmetric cases*. Having an n-by-n matrix A, one can define the directed graph $G(A) = (V(A), E(A))$ on vertices $V(A) = \{1, \ldots, n\}$ by

$$(j, i) \in E(A) \iff A_{i,j} \neq 0.$$

This is indeed a generalization since for structurally symmetric matrices the graph models intuitively coincide[2]. Also note that *the direction in the definition could be swapped*, i.e., the edges could be starting at the row-index node and ending at the column-index node of the graph, but this version will prove convenient later on. Focusing on the results of Lemma 3.4.1, consider the graph model of a general lower-triangular matrix. The direction of edges is *always from a lower numbered vertex to a higher one* (due to the lower-triangular property) and therefore there are *no cycles*, i.e., graph model of a lower triangular matrix is a *directed acyclic graph* sometimes called *dag*. Let us now reformulate the results of Lemma 3.4.1 for sparsity structures only.

Theorem 3.4.1 ([36])**.** *Let $\hat{L}$ be an k-by-k non-singular, lower triangular matrix and b a k-dimensional vector for some k natural. Let us state the* non-cancellation assumption*, i.e., sum, difference and product of two numbers is nonzero if and only if at least one of them is nonzero. Then the structure of the solution of the linear problem $\hat{L}x = b$ can be formulated as*

$$Struct(x) = Struct\left(\hat{L}^{-1}b\right) = Reach_{G(\hat{L})}(\, Struct(b)\,),$$

Moreover, computation of the structure can be realized via the breadth-first search in the graph $G(\hat{L})$ resulting in costs bounded by $\mathcal{O}\left(nnz(b) * (k + nnz(\hat{L}))\right)$.

This theorem coupled with Lemma 3.4.1 encourages us to define the second sparsity structure S_2 as follows.

Algorithm 7 Structural factorization for Algorithm 5

Input: An n-by-n SPD matrix A and the $(j-1)$-th leading submatrix of the incomplete factor $\bar{L}_{j-1}$.
Output: Row and column structures $(S_2)_{j,*}$ and $(S_2)_{*,j}$.

1: Set $(S_2)_{j,*} = Reach_{G(\bar{L}_{j-1})}(\, Struct(A_{1:j-1,j})\,)$;

2: Set $(S_2)_{*,j} = \left[Struct(A_{*,j}) \quad \bigcup \quad \left(\bigcup_{i \in (S_2)_{j,*}, i \neq j} (S_2)_{*,i}\right)\right] \setminus \{1, \ldots, j-1\}$;

Let us emphasize here that the sparsity structure S_2 is *not equivalent with the final structure of the factor*, because the dropping is performed later. The main drawback of the introduced structure S_2 is highlighted below.

[2]To precise this statement, it is necessary to introduce the notion of the *graph isomorphism*. We will not do so here.

Observation 3.4.1. *Although the row structure $(S_2)_{j,*}$ is available based only on the already fully processed part of the computed factor, the column structure $(S_2)_{*,j}$ still requires explicit evaluation of the column structures of the columns that form the current active part of the incomplete factor. In other words, fully implicit storing of the current active part of the factor is not compatible with the structures S_2. Moreover, A has to be available as well, i.e., even A can not be stored implicitly.*

Although the above observation might make the structure S_2 not viable in some cases, it can be still used most of the time. In many instances, one only wants *factors to be data-sparse*, while the matrix itself is stored explicitly in some sparse format, as mentioned in Section 2.4. Or, possibly, one is able to obtain the *graph* $G(A)$ before getting A. Also, the columns in the current active part can be stored implicitly, *provided we have stored their structure separately.* This introduces some extra memory costs, but those are likely to be quite negligible, especially in the case of working with blocks instead of entries.

The natural question is how do the structures S_1 and S_2 compare. Assuming there is no row dropping in Algorithm 5, one can state the following.

Theorem 3.4.2. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with an arbitrary fixed column dropping rule on line 14 and* no row dropping *on line 15. Then, carrying out the computation with sparsity structures S_1 and S_2 consecutively, one can write*

$$(S_1)_{j,*} \subset (S_2)_{j,*} \subset Struct(L_{j,*}) \quad \text{and} \quad (S_1)_{*,j} \subset (S_2)_{*,j} \subset Struct(L_{*,j}),$$

where L is the complete Cholesky factor of A.
Moreover, the equalities will take place under the following conditions.

- $(S_1)_{j,*} = (S_2)_{j,*}$ *if and only if the fixed column dropping on line 14 did not drop any entry with row index j so far.*
- $(S_1)_{*,j} = (S_2)_{*,j}$, *provided* $(S_1)_{j,*} = (S_2)_{j,*}$.
- $(S_2)_{j,*} = Struct(L_{j,*})$ *if and only if*

$$Reach_{G(\bar{L}_{j-1})}(\, Struct(A_{1:j-1,j})\,) = Reach_{G(L_{j-1})}(\, Struct(A_{1:j-1,j})\,).$$

- $(S_2)_{*,j} = Struct(L_{*,j})$, *provided*

$$Reach_{G(\bar{L}_{j-1})}(\, Struct(A_{1:j-1,j})\,) = Reach_{G(L_{j-1})}(\, Struct(A_{1:j-1,j})\,)$$

  and the fixed column dropping on line 14 did not drop any entry with row index j so far.

Proof. Let us first focus on the relations for the row structures. To obtain the first inclusion, it is enough to take into account that nonzeros in $(S_1)_{*,j}$ have been created by the same way as in the complete Cholesky factorization. Hence the first inclusion follows from Lemma 3.4.1 and Theorem 3.4.1. Also, it is clear that the equality holds if and only if the column dropping have not discarded any entry of the j-th row throughout the first $j-1$ steps of the algorithm. The second inclusion follows immediately since no fill-in could have been (in comparison to the complete

Cholesky factorization) added by potential column dropping of the algorithm. The condition for the equality is an immediate consequence of Lemma 3.4.1.

Regarding the column structures, the first inclusion follows immediately since the column structure is *derived* from the row structures (and the original matrix structures) only. Hence inclusion of row structures imply the inclusion of the column structures and the same holds for the equality. The second inclusion admits the same reasoning as for the case of row structures, i.e., no additional fill-in (in comparison to the complete Cholesky) could have been added. □

A natural question is, whether a richer structure is something *desirable* when it comes to incomplete Cholesky factorization. We will see in the following section that it may be *beneficial* in the data-sparse case. But first, let us focus on incorporating the row dropping as well.

First observation is that if the row dropping on the line 15 is retrieved, the structures S_1 and S_2 do not satisfy an inclusion either way in general. The above theorem shows a particular case of $S_1 \subset S_2$, but if the most extreme row dropping is employed, i.e., the resulting leading principal submatrix will be *only diagonal*, then

$$(S_2)_{j,*} = Struct(A_{1:j-1,j})$$

and hence $S_1 \nsubseteq S_2$ in general, since the column dropping could have retained also some fill-in in the j-th row. This counterexample seems to be rather specific as complete row dropping is very rarely a useful approach. Nonetheless, even such special case of row dropping results in a decently rich row structure $(S_2)_{j,*}$ as highlighted in the following lemma.

Lemma 3.4.2. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with arbitrary fixed column dropping rule on line 14 and* full row dropping *on line 15. Then, carrying out the computation with sparsity structures S_2 gives*

$$(S_2)_{j,*} = Struct(A_{1:j-1,j}) = Struct(A_{j,1:j-1}).$$

In other words, the column updates structure is identical to the one of the zero fill-in *Cholesky factorization $IC(0)$.*

Lemma 3.4.2 implies that the structure S_2 might be potentially quite rich even if a drastic row dropping takes place. Also, let us once more emphasize that this doesn't necessarily imply that the factor $\bar{L}_{j-1}$ will be as well. In order to elaborate further on the structure S_2, it is necessary to introduce the notion of *transitive closure* and *transitive reduction* of a given graph.

Consider a graph $G(V, E)$, either directed or undirected. The *transitive closure of* G, denoted by $G(V, E^*)$ or simply G^*, is defined as a graph on the vertex set V with edge connecting x to y (possibly directed) if and only if there exists a path from x to y in $G(V, E)$. The *transitive reduction of* G, denoted by $G(V, E^t)$ or simply G^t, is defined as a graph on the vertex set V with the smallest possible edge set $E^t \subset V \times V$ so that its transitive closure is equal to the one of G, i.e., so that

$$\left(G^t\right)^* = G^*.$$

The following theorem summarizes some known results for the above defined notions, following the work of Aho, Garey and Ullman.

Theorem 3.4.3 ([1]). *Let $G(V, E)$ be a finite acyclic directed graph with $|V| = n$. Then its transitive reduction exists and is unique. Moreover, computation of the transitive reduction and the transitive closure of G can be carried out with identical time costs amounting to multiplication of two n-by-n boolean matrices.*

Having this result, we can now state the main results for the sparsity structure S_2. But before doing so, let us emphasize that those results are intended to give a better insight into the structure S_2, i.e., there is, in general, no reason to modify the column or row dropping patterns in order to attain one of the equalities in Theorem 3.4.2.

Lemma 3.4.3. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with an arbitrary fixed column dropping rule on line 14 and* no row dropping. *Using the sparsity structure S_2, denote the $(j-1)$-th leading principal submatrix of the incomplete factor by $\bar{L}_{j-1}$ and the j-th row structure by $(\bar{S}_2)_{j,*}$. Let us consider the same column dropping and* arbitrary row dropping *in line 15 and carry out Algorithm 5 with sparsity structures S_1 and S_2 respectively. Regarding the sparsity structure S_2, denote the $(j-1)$-th leading principal submatrix of the incomplete factor by $\hat{L}_{j-1}$ and the j-th row structure by $(\hat{S}_2)_{j,*}$.*
Moreover, let us assume that the row dropping has been chosen so that the transitive closures of $G(\bar{L}_{j-1})$ and $G(\hat{L}_{j-1})$ are equal. Then, carrying out the computation with sparsity structures S_1 and S_2 consecutively, one can write

$$(S_1)_{j,*} \subset (\bar{S}_2)_{j,*} = (\hat{S}_2)_{j,*} \subset Struct(L_{j,*})$$
$$\textit{and}$$
$$(S_1)_{*,j} \subset (\bar{S}_2)_{*,j} = (\hat{S}_2)_{*,j} \subset Struct(L_{*,j}),$$

where L is the complete Cholesky factor of A.
Moreover, the sufficient conditions for the equalities from Theorem 3.4.2 are still valid.

Proof. By definition of $(S_2)_{j,*}$ on line 1 in Algorithm 7 one can see that $k \in (S_2)_{j,*}$ if and only if there is an edge $k \to j$ in the transitive closure of the graph of the principal leading submatrix. Hence, assuming that $G(\bar{L}_{j-1})^* = G(\hat{L}_{j-1})^*$ one gets $(\bar{S}_2)_{j,*} = (\hat{S}_2)_{j,*}$. Then, using Theorem 3.4.2, the results follow. □

Notice that the above lemma gives several options for further development. First, in Lemma 3.4.3, the focus is on comparison of $\hat{S}_2$ and $\bar{S}_2$, i.e., of the sparsity structures obtained by Algorithm 5 *with* and *without row dropping* and due to Theorem 3.4.2 there is a relatively good understanding of when $(\hat{S}_2)_{j,*} = (S_1)_{j,*}$. But one can formulate an analogous statement comparing $(\hat{S}_2)_{j,*}$ and $Struct(L_{j,*})$ by employing the transitive closure of $G(L_{j-1})$. Second, one could also take into account that even if the transitive closures of the two graphs are not equal, the sparsity structures still can be equal, as, by the definition, the important thing is *reachability from a certain vertex set* rather than reachability from each vertex. Last but not least, one could try to evaluate, for which row dropping the key property of $G(\bar{L}_{j-1})^* = G(\hat{L}_{j-1})^*$ holds true.

Since we have already stated that the sparsity structures inclusion are meant to give a rough orientation rather than to propose a row dropping according to them, we will not pursue the first two options. The last idea is refined in Theorem 3.4.4 below.

Theorem 3.4.4. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with an arbitrary fixed column dropping rule on line 14 and* no row dropping. *Using the sparsity structure S_2, denote the $(j-1)$-th leading principal submatrix of the incomplete factor by $\bar{L}_{j-1}$ and the j-th row structure by $(\bar{S}_2)_{j,*}$. Let us now consider the same column dropping and* arbitrary row dropping *in line 15 and carry out Algorithm 5 with sparsity structures S_1 and S_2 respectively. Using the sparsity structure S_2, denote the $(j-1)$-th leading principal submatrix of the incomplete factor by $\hat{L}_{j-1}$ and the j-th row structure by $(\hat{S}_2)_{j,*}$.*
Moreover, let us assume that the row dropping was chosen so that the the first nonzero of each column was retained, i.e., so that

$$\mathcal{T}(\bar{L}_{j-1}) \subset G(\hat{L}_{j-1}).$$

Here, $\mathcal{T}(\bar{L}_{j-1})$ denotes the elimination tree of the factor $\hat{L}_{j-1}$ (see (1.3)*). Then one can write*

$$(S_1)_{j,*} \subset (\bar{S}_2)_{j,*} = (\hat{S}_2)_{j,*} \subset Struct(L_{j,*})$$
$$\textit{and}$$
$$(S_1)_{*,j} \subset (\bar{S}_2)_{*,j} = (\hat{S}_2)_{*,j} \subset Struct(L_{*,j}),$$

where L is the complete Cholesky factor of A.
Moreover, the sufficient conditions for the equalities from Theorem 3.4.2 are still valid.

Proof. Recalling the fact that in the Cholesky factorization the elimination tree $\mathcal{T}(L)$ is in fact the transitive reduction of the fill-in graph $G(L)$ (see [60]), the result follows from Lemma 3.4.3 and the definition of the transitive closure and the transitive reduction. □

To conclude this subsection, we add one more reformulation for the row sparsity structure, using the elimination tree of the principal leading submatrix.

Theorem 3.4.5. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with arbitrary fixed column and row dropping rules on lines 14 - 15. Using the sparsity structure S_2, denote the $(j-1)$-th leading principal submatrix of the incomplete factor by $\hat{L}_{j-1}$ and the j-th row structure by $(S_2)_{j,*}$.*
Taking the j-th principal leading submatrix of the incomplete factor $\tilde{L}_j$ as

$$\tilde{L}_j = \begin{bmatrix} \hat{L}_{j-1} & \\ \hat{L}_{j-1}^{-1} A_{1:j-1,j} & a_{jj} \end{bmatrix},$$

the row sparsity structure $(S_2)_{j,*}$ is equal to the structure of the j-th row subtree of the elimination tree $\mathcal{T}(\tilde{L}_j)$.

Proof. The result follows by definition of the row subtree, see Chapter 1. □

3.4.2 Column structure

As observed in the previous subsection, the so far proposed formulation of the structural computation is not fully implicit, i.e., the column structure computation still requires the structures of the previous columns to be stored explicitly.

Therefore, the goal of this subsection is to replace the formula

$$(S_2)_{*,j} = \left[Struct(A_{*,j}) \quad \bigcup \quad \left(\bigcup_{i \in (S_2)_{j,*}, i \neq j} (S_2)_{*,i} \right) \right] \setminus \{1, \ldots, j-1\}$$

by a different one that would not require the active part of the computed incomplete factor. The strategy used so far has been to use the known relations of the complete Cholesky factorization that are favourable for us in the above sense. The following theorem continues in this direction.

Theorem 3.4.6 ([60, Theorem 3.7 and Corollary 3.9])**.** *Let A be an n-by-n SPD matrix and let $1 < j < n$. Denoting L the Cholesky factor of A, one can write*

$$\begin{aligned} Struct(L_{*,j}) &= Adj_{G(A)}\mathcal{T}[j] \setminus \{1, \ldots, j-1\} \\ &= Adj_{G(L+L^T)}\mathcal{T}[j] \setminus \{1, \ldots, j-1\}. \end{aligned}$$

Proof. To check the first equality, consider $i \in Struct(L_{*,j})$. Using Theorem 1.2.2 this is equivalent with the fact that there is a vertex k such that $A_{i,k} \neq 0$ and j is an ancestor of k in the elimination tree $\mathcal{T}(L)$, which can be equivalently written as $i \in Adj_{G(A)}\mathcal{T}[j]$.

Regarding the second equality, the inclusion $Adj_{G(A)}\mathcal{T}[j] \subset Adj_{G(L+L^T)}\mathcal{T}[j]$ is trivially met as $G(A) \subset G(L + L^T)$. To prove the other inclusion, consider some $k, i < j$ so that $k \in \mathcal{T}[j]$ and $i \in Adj_{G(L+L^T)}(k)$. Consequently, either $L_{i,k} \neq 0$ or $L_{k,i} \neq 0$, depending on whether $k < i$ or $k > i$. Let us first consider the case $k < i$, i.e., $L_{i,k} \neq 0$. By Theorem 1.2.2 this implies that there exists a vertex l such that $l \in \mathcal{T}[k] \subset \mathcal{T}[j]$ such that $A_{l,k} \neq 0$, which give the result. Assuming $k > i$, i.e., $L_{k,i} \neq 0$, one can use the reformulation of Theorem 1.2.3, see (1.4), to deduce that either $A_{k,i} \neq 0$ (this itself gives the result already) or there exists k' such that $L_{i,k'}$ (which case was already considered above). Either way the result follows. □

In the same fashion as in the previous subsection one can now define new sparsity structure S_3 based on the above result.

Algorithm 8 Structural factorization for Algorithm 5

Input: An n-by-n SPD matrix A and the $(j-1)$-th leading submatrix of its incomplete factor $\bar{L}_{j-1}$.
Output: Row and column structures $(S_3)_{j,*}$ and $(S_3)_{*,j}$.

1: Set $(S_3)_{j,*} = Reach_{G(\bar{L}_{j-1})}(Struct(A_{1:j-1,j}))$;

2: Set $\bar{L}_j$ as $\tilde{L}_j = \begin{bmatrix} \hat{L}_{j-1} & \\ \hat{L}_{j-1}^{-1} A_{1:j-1,j} & a_{jj} \end{bmatrix}$;

3: Get the j-th rooted subtree in $\mathcal{T}[j]$ of the elimination tree $\mathcal{T}(\bar{L}_j)$;

4: Set $(S_3)_{*,j} = Adj_{G(A)}\mathcal{T}(\tilde{L}_j) \, , \setminus \{1, \ldots, j-1\}$;

On the first sight, this approach might look promising. The current active part of the incomplete factor is not needed any more, as required. And although the structure of the original matrix A is necessary, this is only a minor drawback,

considering the structure of A is vital for determination of the row structure anyway. Also, considering the computation costs, the above algorithm should be fairly cheap *because only the adjacent sets need to be determined*, i.e., no search in the graph of $G(A)$ is needed. However, as we will see, the situation is more complicated. In order to see the issue consider the following lemma. The result is analogous to the one of Theorem 3.4.2, i.e., comparison of the new sparsity structure to the established ones.

Lemma 3.4.4. *Let A be an n-by-n SPD matrix and let $1 < j < n$. Consider the Algorithm 5 with arbitrary fixed column dropping rule on line 14 and* no row dropping. *Then, carrying out the computation with sparsity structures S_2 and S_3 consecutively, one can write.*

$$(S_2)_{j,*} = (S_3)_{j,*} \quad \textit{and} \quad (S_2)_{*,j} \subset (S_3)_{*,j}.$$

Proof. The row structures are by definition identical. Regarding the column structures, let us consider $i \in (S_2)_{*,j}$ (hence $i > j$). By definition the corresponding nonzero in the j-th column of the factor is either a copy of a nonzero entry in the original matrix, i.e., $A_{ij} \neq 0$, or it is a fill-in, i.e., one can take $k < j$ such that $\bar{L}_{ik} \neq 0$ and at the same time $k \in \mathcal{T}[j]$. If the first case takes place then clearly $i \in (S_3)_{*,j}$, since $j \in \mathcal{T}[j]$. Considering the alternative holds, one can employ Theorem 1.2.2 about the column structure replication and conclude that there exists $k' < k$ such that $A_{ik'} \neq 0$ and $k' \in \mathcal{T}[j]$ as well. Note that Theorem 1.2.2 can be used since the column structure replication is still present in Algorithm 8. Altogether $i \in (S_3)_{*,j}$ and the proof is finished. □

The conditions for equality are omitted but they can be easily deduced from the proof. The above result leads to the following important observation.

Observation 3.4.2. *Algorithm 8, i.e., the sparsity structure S_3, does not take into account previous column dropping when determining the structure of the j-th column of the incomplete factor. Consequently*

- *Algorithm 5 with structures computed by Algorithm 8 may carry out a* large number of operations with zeros. *The most extreme case would be the complete column dropping below the j-th row. In this case, the j-th column is numerically updated (possibly many times) by zeros that represent the fill-in. Assuming the factorization has been done blockwise, which is our goal, this is even more alarming. On the other hand, it is almost guaranteed that any* numerical-value-based column dropping would resolve this issue.

- *The above mentioned point is not exclusive to the particular formulation of Algorithm 8.* On the contrary, this issue is inherent to any procedure that does not take into account possible column dropping, i.e., any procedure that does not use the current active part of the incomplete factor. *The only exception would be to consider only the structure of the j-th column of the matrix A.*

Although this observation is straightforward, it is fundamental for our attempts to find a suitable way of determining the column structure. In other words, one cannot obtain an efficient procedure for computation of the column structure fully

implicitly - the current active part of the factor has to contribute to the column structure determination. It should be noted that Lemma 3.4.4 does not include the row dropping, which, in theory, could play similar role. However, it is clear that for a general row dropping Observation 3.4.2 holds true as well.

Summarizing the above development, possible enriching of the column structure in comparison to the natural one (used for both S_1 and S_2) might bring additional difficulties. In particular, it may result in adding, subtracting and multiplying with zero blocks *explicitly*, which should be avoided. Considering *only a suitable approximation based on structural patterns* could be fruitful. However, this direction will not be further developed here, also because it partially overlaps with the column dropping. In this sense, one may view it as an example of the need for suitable dropping criterion. This topic will be considered in the next section, but before that, let us update Algorithm 5.

Algorithm 9 Incomplete structurally sparse column-oriented Cholesky factorization

---

Input: A square n-by-n SPD matrix A.

1: **function** PROTO_ICHOL
2: **for** $j = 2, \ldots, n$ **do**
3: $\bar{l}_{j-1,j-1} \leftarrow \sqrt{a_{j-1,j-1}}$;
4: **for** $s \in Struct(\bar{L}_{j:n,j})$ **do**
5: $\bar{l}_{s,j-1} \leftarrow \frac{\bar{l}_{s,j-1}}{\bar{l}_{j-1,j-1}}$;
6: **end for**
} **factorize_col** $(j-1)$

7: **lr_approx_col** $(j-1)$;

8: Set $(S_2)_{j,*} = Reach_{G(\bar{L}_{j-1})}(\, Struct(A_{1:j-1,j})\,)$;
9: Set $(S_2)_{*,j} = \bigcup_{i \in (S_2)_{j,*}, i \neq j} (S_2)_{*,i}$;
10: Set $(S_2)_{*,j} = (S_2)_{*,j} \bigcup Struct(A_{*,j})$;
11: Set $(S_2)_{*,j} = (S_2)_{*,j} \setminus \{1, \ldots, j-1\}$;
} **symb_struct** (j)

12: **for** $k \in (S_2)_{j,*}$ **do**
13: **for** $i \in (S_2)_{*,j}$ **do**
14: $a_{ij} \leftarrow a_{ij} - \bar{l}_{jk}\bar{l}_{ik}$;
15: **end for**
} **update_col_by_col** (j,k)
16: **end for**

17: Apply dropping to $\bar{L}_{j,*}$ and update the finalized part $\bar{L}_j$ of the incomplete factor;
18: **end for**
19: Return the incomplete factor $\bar{L}$;
20: **end function**

The above updated version already has a fixed way to determine the column and row structures, which enables to use the procedure

$$\textbf{lr\_approx\_col}\,(j-1),$$

which approximates the j-th column (possibly a block-column) and stores the approximation *implicitly* only. For block-columns this might mean a low-rank approximation of the whole block-column as in Figure 1.1 or, in some cases, one could approximate only some nonzero blocks of the block-column. A number of different techniques for obtaining such low-rank approximation was considered in Section 2.3. Although our focus is to work with blocks, the memory costs for storing columns can be reduced even in the scalar case. Namely, if all of the entries of a column of a matrix are small in magnitude, one could approximate the values only by their average and then one needs to store only the indices of the nonzero entries and the mean value. Although this is not strictly speaking a low-rank approximation, it fits well into the context here.

Notice that the column dropping is no longer present in the above algorithm. It is not necessary to use it here since, in principle, it is substituted by the low-rank approximation routine. The main goal of the dropping is to significantly reduce the memory costs, which is, in many cases, already achieved by the low-rank approximation. This being said, there are cases where the blockwise fill-in would be not manageable. In such cases the column dropping has to be applied and the dropping can be retrieved immediately. A precise description of this routine will come at the end of this chapter, but we wanted to highlight the fact that the reformulation now *allows for the implicit storage of the active part of the factor*. The price to pay is that the principal leading submatrix $\bar{L}_{j-1}$ *has to be stored explicitly* - at least its structure.

3.5 Exploiting data-sparsity

Up to now, the focus has been on *enabling the data-sparse approach*, i.e., on enabling an implicit storing of the current active part of the incomplete factor during the incomplete Cholesky factorization. The objective has not been met fully, but only to a larger extent. Thereby, the goal of this section is to complement this development and modify it so that *the procedure is not only applicable to blockwise rank-deficient problems but exploits this feature as much as possible.* Naturally, this is a difficult challenge and we do not claim to present *the final solution.* Many other contributions have been published when it comes to this topic, few of which we have already mentioned. One can view this whole chapter rather as *an addition* to the work we have already referred to that might introduce a *different point of view of* and a *different approach to* the field of blockwise low-rank matrices and their utilization in the context of preconditioning.

3.5.1 Double sparsification

The idea of *double sparsification* is based on a simple observation (highlighted below) that can be tracked back[3] to Tismenetsky [77] and Kaporin [54].

Observation 3.5.1. *Having a Cholesky-based preconditioning technique that is based on* dropping*, either structural or based on numerical values, it is, in most cases, beneficial to first carry out the update and perform dropping only after that than the other way around.*

[3]The explicit formulation is not there, but proposed techniques clearly point in this direction.

In other words, keeping more information inside the factorization, even if this information is considered *only for updating*, is desirable. However simply this observation may sound at first, note that the classical scheme of dropping (as well as the one of Algorithm 5 and 9) does not align smoothly with it. Quite on the contrary, the dropping in the j-th column takes place *immediately after the finalization*, i.e., any update *by* the j-th column is carried out with the already finalized, sparsified structure. In order to profit from Observation 3.5.1, one has to introduce a second dropping, i.e., consider a *double sparsification*.

The general approach is to have *two dropping criteria* - presenting a "coarser sieve and a finer one"for the dropping - entries (or blocks) that pass through both are part of the final factor, whereas entries that pass only through the coarser one are *used only for the future updates but are not put into the final incomplete factor*. This can be written down in the matrix form as the following decomposition of A.

$$A = (\bar{L} + R)(\bar{L} + R)^T + E,$$

where $\bar{L} \approx L$ is the incomplete Cholesky factor on the output, R is the matrix which entries were from the final factor but were used for the updates and E is the error matrix. The idea of double sprasification was indirectly used already by Kaporin [54] as mentioned and other authors have adopted it as well, see, e.g., [73], [55] and [74].

Notice that the additional row dropping in Algorithm 5 and 9 fulfils a similar role with one crucial difference - the row dropping is *postponed as far as possible*, i.e., the second dropping process has as much information as possible. That is not the case if the decision is made immediately after the column finalization.

Naturally, the effectiveness strongly depends on the chosen dropping criteria, which are the focus of the rest of this subsection. Let us stress out here that in general there is no need for the two dropping criteria to be compatible in the sense that any entry that is not dropped by the second "finer"dropping will not be dropped by the "coarser"either. The terms *finer* and *coarser* are chosen rather for illustration of the overall purpose of each of the dropping steps but are not meant literally.

The coarser sieve

Considering the complete block Cholesky factorization (either structurally sparse or dense), the computational bottleneck is the **update_col_by_col** procedure, i.e., the lines 12-16 in Algorithm 9[4]. This procedure can be further dissected into block-by-(column)-block multiplication and summation of two (column)-blocks, where the first one demand w_k-times more operations, where w_k is the *width of the k-th block-column*. However, assuming the current active part of the (incomplete) factor is stored implicitly, e.g., the block-column is stored as a low-rank approximation - either only the nonzero blocks or the entire block-column - this is not necessarily the case any more.

For illustration, consider the block-column treated as a unit, i.e., it is stored implicitly by some $p_{\max}$-rank approximation. The multiplication now requires only $(2w_k - 1)p_{\max}w_j + (2p_{\max} - 1)w_k h_j$ in comparison to $(2w_k - 1)w_j h_j$. Here and

[4]In sense of the actual computation costs, we do not consider the issue of data movement.

later on h_j denotes the *height of the j-th block-column*, i.e., the number of matrix rows. Asymptotically this difference scales as $w_k/p_{\max}$ and the meaning of this ration is how much faster - measured in number of operations - the whole update procedure is carried out. Or, equivalently, one can interpret it as an indicator of *how many additional updates could be incorporated, provided both algorithms spend equal amount of time by column updates*. This formulation points out to the fact that one can exploit the data-sparsity in two different directions that can be summarized as follows. Either one keeps the number of updates and obtain a speed-up in the computation or one can keep the time costs, perform more updates and, hopefully, obtain a better performing preconditioner. Here we have chosen the second option, i.e., aiming for more updates and hopefully getting a better preconditioner, but that does not mean that the other option is not viable.

Another implication of the above observation is that if one *needs a possibility for an additional updates*, i.e., possibly richer row structure it is, surprisingly, not an issue. Also, this somewhat sets the tone for the dropping - more updates corresponds to "utilization of the data-sparsity".

The finer sieve

In comparison to the above dropping - the coarser sieve - the objective for this second dropping is to *finalize the incomplete factor*. Let us to recall Section 1.3 and the last paragraph in particular - solving systems with the preconditioner has to be as fast as possible. In our case, one can enhance the efficiency *mainly by sparsity of the final factor* and thereby the second dropping - so called finer sieve - has to account for that. Second objective - often competing with the one just mentioned - is to retain as good approximation of the true Cholesky factor as possible. This holds for both structural and numerical part as the finalized factor will still be used for the structural computation. Unfortunately, this is where one has to settle for heuristics only as a proper analysis for the general case is not available even in the case of the classical incomplete Cholesky without the modification we have introduced so far.

One commonly used technique is based on *control of the growth of the condition number of the finalized principal leading submatrix of the incomplete factor*. Clearly

$$\kappa(A) = \kappa(LL^T) \quad \text{, i.e.,} \quad \kappa(L) = \sqrt{\kappa(A)}$$

and one can argue that any incomplete factor $\bar{L} \approx L$ such that $\kappa(\bar{L}) >> \sqrt{\kappa(A)}$ is not a suitable approximation. However, the computation of $\kappa(A)$ is computationally out of question and one has to settle for an estimator or, alternatively, focus on the growth of $\kappa(\bar{L})$ only, i.e., requiring only that the condition number is not sky rocketing when the next row is added to the incomplete factor. This idea can be found in many papers, e.g.,[62], [17], [10] and [11] to name just a few[5]. The techniques for the estimation of a condition number of a matrix form a deep and wide field itself and we will not discuss this problematic with more details. Rather than that we refer the interested reader to the survey paper of Higham [50] and the work cited there or, more recently, the mentioned work of Bollhöfer [10].

[5]Note that this approach is not limited to the SPD case.

Our proposal follows the development in [74, Section 4] to some extent, although the paper is focused on symmetric *indefinite* systems. There the authors use an estimate on the quantity called *condest*, originally due to Chow and Saad [17], defined as

$$condest \equiv condest(L) = \|(\,\bar{L}\bar{L}^T\,)^{-1}e\|_\infty$$

that measures the *instability of the triangular solver with* $\bar{L}$. Scott and Tůma estimates *condest* by *instability factor* g_j *at the* j*-th step of the factorization*, defined as the largest entry (in absolute value) in the vector $L_j^{-1}e$[6], where L_j is the current finalized part of the factor (i.e., the leading principal submatrix) and $e = (1, \ldots, 1)^T$. Clearly, g_j can be monitored at each step of the factorization as illustrated in Algorithm 10. For simplicity, we present here the scalar version. One can easily reformulate it for a general blocking. Also, the LDL^T factorization is assumed, i.e., the factor L is unscaled by the the square roots of the diagonal elements. This corresponds to our implementation, but it is only a technical difference and the procedure given below can be easily modified for the other version.

Algorithm 10 Computation of the instability factor g_j, see [74, p.8]

1: **if** $j = 1$ **then**
2: $v_1 = 1$;
3: $g_1 = 1$;
4: **else**
5: Set $e = (1, \ldots, 1)^T \in \mathbb{R}^j$;
6: $v_j = L_j^{-1}e = \begin{bmatrix} L_{j-1} & \\ l_j & 1 \end{bmatrix}^{-1} e = \begin{bmatrix} L_{j-1}^{-1} & \\ -l_j L_{j-1}^{-1} & 1 \end{bmatrix} e = \begin{bmatrix} v_{j-1} \\ -l_j v_{j-1} + 1 \end{bmatrix}$;
7: $g_j = \max(g_{j-1},\ |v_{j_{\text{end}}}|)$;
8: **end if**

The dropping strategy utilizing the instability factor g_j is based on a simple *greedy approach*. A row entry (scalar in the above example or block in general) is tried out in the j-th row, i.e., l_j is enriched, and provided g_j does not increase (or does not increase too much) the entry is accepted into the final factor. The algorithm stops once an entry is rejected, i.e., once a growth (or too large growth) in the instability factor would occur by adding another entry. The entries are fetched in reasonable order, i.e., one first tries to add the entries that increase the estimate the least.

However, Scott and Tůma use the procedure only to *enhance* some precomputed factor, i.e., on input is also a set of entries of the j-th block-row that is already incorporated in the final factor and their procedure attempts to improve it by adding additional blocks. In our case, we do not have any a-priori given structure of the final factor. This is resolved later in the following section. Assuming the initialization fixed blocks with block-column indices $k_1, \ldots k_l$, let us denote them by λ_j, i.e., $\lambda_j = \{k_1, \ldots k_l\}$ and let us denote the rest of the block-column indices of the nonzero blocks in the j-th row by γ_j (following the notation

[6] Originally, they propose to control the largest entry of the vector $|L_j^{-1}e|$, but then they argue for the vector $L_j^{-1}e$ being more suitable.

of Scott and Tůma). Then one can write

$$v_j = 1 + \sum_{k \in \lambda_j} (l_j)_k (v_{j-1})_k + \sum_{k \in \gamma_j} (l_j)_k (v_{j-1})_k. \tag{3.4}$$

The greedy strategy will consecutively try to add indices from γ_j to λ_j, i.e., incorporate more blocks into the j-th block-row of the final factor. Note that from (3.4) one can see that evaluation of g_j after addition of a particular entry is very cheap. The procedure is therefore computationally quite appealing, provided that the number of accepted entries for each row is not too large.

Notice that the focus above has no apparent connection to control of the growth of the condition number as suggested previously. However, using a similar greedy strategy, only coupled with a condition number estimator could lead in that direction. However, the computational costs of the condition number estimator could be unmanageable, making the routine infeasible for practical usage.

3.6 The proposed preconditioner

The previous section was devoted to the presentation of the core concepts and building blocks of the preconditioning technique we are proposing. However, for the sake of simplicity and in comprehensibility many of the technical details were skipped. Here we will provide more details related to the implementation and we will continue in the next chapters, where the focus will be on costs analysis and presentation of the results. The implementation was done in MATLAB code and *it was not optimized towards either memory or computational costs*. The goal has been not to provide final stage code but rather *to test viability of the ideas and of the complementary approach to the usually used one*. This required the key features to be implemented reasonably efficiently, while some commonly used subroutines were adopted from the MATLAB library. This approach is coupled with a *careful analysis* to distinguish the conceptual bottleneck and the particular implementation challenges that could be overcome. The first section of the next chapter focuses on this. The rest of this section is split into paragraphs that will discuss each of the procedures of the following schematic abbreviation of Algorithm 9. To avoid confusion, let us denote by $\tilde{S}_2$ the structure *after the first dropping is carried out*, i.e., the structure accepted by the coarser sieve dropping that determines the column update structure. The structure of the final factor, i.e., after both of the droppings have been carried out will be denoted by $Struct(\bar{L}_j)$.

Algorithm 11 Incomplete structurally sparse column-oriented Cholesky factorization

Input: A square n-by-n SPD matrix A.

```
1: function PROTO_ICHOL
2:     blocking (A);
3:     for j = 2, ..., ν do
4:         factorize_col (j − 1);
5:         lr_approx_col (j − 1);
6:         finer_sieve (j − 1);
7:         symb_struct (j);
8:         coarser_sieve (j);
9:         for k ∈ (S̃_2)_{j,*} do
10:            update_col_by_col (j, k);
11:        end for
12:    end for

13:    Return the incomplete factor L̄;
14: end function
```

Notice that the main **for**-cycle runs only through $2, \ldots, \nu$ and no longer through $2, \ldots, n$. The ν is a parameter retrieved from the **blocking**$(\cdot)$ routine as the number of block-columns (and block-rows).

The code is meant for sparse or almost sparse matrices foremost and thereby all matrices are stored in a format similar to CSR[7]. That is why the whole code considers the *upper triangular* factor, i.e., with L^T instead of L. The entire exposition above and also below is done with the lower triangular case, although it is not identical with the implemented format. *However, the reformulation does not change anything important and definitely does not affect the costs of the algorithm, either computational- or memory-based.*

blocking (A) For the case of structurally sparse or structurally block sparse matrices, the blocking techniques were described in Subsection 2.4.2 - the basic one adopted from the paper of Ashcraft ([4]) and then a more general one by Saad ([69]). In the paper of Saad one can find the implementation for both of the blocking procedures (Algorithm 2.1, 2.2 and 2.3). Since this is simply a preprocessing for the preconditioning routine, we omit the algorithms here.

For the case of structurally non-sparse matrices, e.g., for matrices that are somewhere on the edge of being dense, or for those which are dense, those blocking algorithms are likely to give very large blocks[8]. A scheme similar to the hierarchical ones presented in Subsection 2.4.1, but the sequential Cholesky cannot process these blocks (see Subsection 2.4.2, Asssumption B1). To overcome this issue, we propose *a hierarchical matrix splitting*. Consider some hierarchical format induced by an admissibility condition, it is easy to see that adopting the blocking based on the dimension of the diagonal blocks is acceptable for the sequential Cholesky factorization. This means symmetrical blocking with the

[7]The abbreviation stands for *compressed sparse rows*, see [33].

[8]For dense matrices both Ashcraft's and Saad's algorithms return no blocking, i.e., the result is a "1-by-1 blocking"with the matrix itself being the entire block.

i-th block-column (and block-row respectively) having the width (or the height respectively) equal to the dimension of the i-th diagonal block. Naturally, this requires the diagonal blocks to be square or, in terms of the hierarchical format, it requires the block-cluster tree to be derived from one labelled cluster tree only, i.e., one cannot use different row and column indices partitioning. This condition is met for all of the commonly used formats. Notice that one *does not need to change the matrix data structure in any way* - the algorithm works with subblocks of the implicitly stored blocks and *this corresponds to considering only part of the column and row matrices* as shown in Figure 3.1 below, which does not introduce new costs.

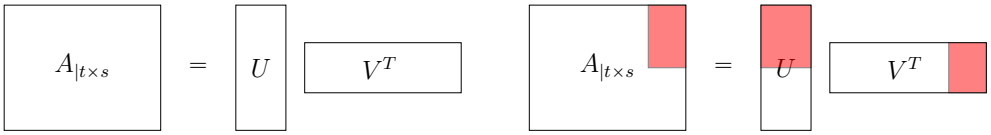

Figure 3.1: Having the low-rank block of A, taking a subblock can be easily formulated in terms of taking only certain part of the row and column matrices U and V^T.

This blocking has been coded in MATLAB as well[9], but *assuming only the strong admissibility condition*, which consequently makes the diagonal blocks of identical dimension $MinDim$. In other words, the strong admissibility condition is used and the splitting is stopped if the resulting blocks have smaller dimension than $MinDim$. However, once another admissibility condition is provided, the code can be easily reformulated - in the sense of applicability, not necessarily the coding itself.

In the rest of the thesis we will consider two different cases. First, the *structurally block-sparse* case, where the block-columns are considered to be block-sparse, i.e., considerable portion of the blocks in each of the block-columns are *zero blocks*. The matrix is in this case treated as a block-sparse one, i.e., operations with those zero blocks are avoided and the matrix is stored in an block-CSC-like format. Second, the *block-column unit* case, where the block-columns are treated as a unit throughout the factorization and are not partitioned any further. This means that the procedure **lr_approx_col** stores implicitly the entire subdiagonal part of the block-column, including the possible zero blocks, which might be a more suitable approach for structurally block-non-sparse matrices.

factorize_col (j) In the scalar case this procedure carries out the square root operation and division by a scalar - see Algorithm 5, lines 5 - 8. For the block scheme, the division is replaced by the multiplication by an inverse - naturally implemented through a dense Gaussian elimination process, i.e., no inverse is in fact assembled during the procedure. Regarding square roots, one can encounter two different versions - either square root free or the classical one. The first one computes the so-called *square-root-free* version, i.e., $A = LDL^T$ instead of the $A = LL^T$. From the computational point of view, the second approach substitutes routine $chol(\cdot)$ for the scalar square root function and then solves the

[9]We would like to thank to Yingzhou Li from the Duke University, who provided us with a basic code skeleton in MATLAB, which we modified to fit our requirements.

systems with the triangular matrix instead of the scalar division. In comparison to that, the square-root-free approach avoids the operation $chol(\cdot)$ completely and simply keeps the diagonal block as it is and then solves the systems with the diagonal block in place of the scalar division. In our implementation the first approach is taken. The main advantage is that the code does not break down when a diagonal block is not SPD but the price to pay is that the systems are solved by general Gaussian elimination, i.e., LU factorization of the diagonal block is computed and forward and backward substitution are carried out. In the second case, one only needs to compute the Cholesky factorization (which is already cheaper than the LU factorization - for both memory and flops costs) and then apply only the backward substitution.

Algorithm 12 The **factorize _ col** procedure - structurally block-sparse

```
1: function FACTORIZE_COL(j)
2:     D = assemble_diag_block(j);
3:     Set col_nnzs = |(S̃_2)_{*,j}|;
4:     for k = 2, ..., col_nnzs do
5:         M = assemble_blck_ordr(k);
6:         prod = MD^{-1};
7:         Incorporate block prod into the CSR storage scheme output;
8:     end for
9:     Return output;
10: end function
```

Algorithm 13 The **factorize _ col** procedure - block-column unit

```
1: function FACTORIZE_COL(j)
2:     D = assemble_diag_block(j);
3:     M = assemble_blck-col(j);
4:     prod = MD^{-1};
5:     Return prod;
6: end function
```

The output, i.e., the nonzero blocks of the column, is saved in the block-CSR format and fetched to the routine **lr _ approx _ col** (j). On lines 6 and 4 in Algorithm 6 and 13, respectively, the MATLAB routine **mldivide** is used. It is, the inverse is not constructed and the result is obtained by reformulation to linear systems.

lr _ approx _ col (j) The low-rank approximation is done either blockwise, provided the structurally block-sparse blocking was done, or for the entire block-column in the case that the hierarchical blocking was applied. The method we have chosen is the randomized SVD (see [47, Section 1.5, p. 226] and the Section 2.3). The oversampling parameter was chosen ad hoc, based on the dimension of the blocks one works with. However, providing any other favourite low-rank method code, one can simply rewrite the name of the routine in the code and everything runs the same.

finer _ sieve (j) This procedure performs the second row dropping that finalizes the incomplete factor. There were mentioned several possibilities of this dropping, but we will consider only three particular implementations.

- Based on threshold dropping as in (1.5), denoted by **finer _ sieve _ norm**;
- Based on control of the condition number, denoted by **finer _ sieve _ cond**;
- Based on the approach of Scott and Tůma, denoted by **finer _ sieve _ ST**.

We will consider each option separately below, but let us emphasize that the output from this routine is simply a structure of the next row to be incorporated into the final factor. The process of doing so will not be described here. The same holds for the assembling of the input, i.e., we will not differentiate between the structurally block-sparse case and the block-column unit case.

First, let us consider the most straightforward of the three methods - the threshold-based dropping. All of the blocks in the block-row are consecutively compared with the diagonal one in the sense of Frobenius norm as in (1.5). Provided the ratio is sufficiently large, the block is accepted. The implementation is given below in Algorithm14. The threshold $\tau \in [0, 1]$ has to be user-specified.

Algorithm 14 The **finer _ sieve _ norm** procedure

```
1: function FINER_SIEVE_NORM(j, τ)
2:     D = assemble_diag_block(j);
3:     Set row_nnzs = |(S̃_2)_{j,*}|;
4:     for k = 1, ..., row_nnzs − 1 do
5:         M = assemble_blck_ordr(k);
6:         if ‖M‖_F ⩾ τ‖D‖_F then
7:             Accept the k-th block in the block-row j;
8:         end if
9:     end for
10: end function
```

The second approach attempts to control the growth of the condition number of the already finalized factor. Since the actual computation of the condition number is not feasible, one has to settle for estimators only. We utilized the built-in MATLAB routine **condest** that is based on the work of Higham and Tisseur [51]. The algorithm uses a simple greedy strategy, i.e., as long as we do not increase the estimator too much, we keep adding the blocks to the final factor. Once a block is not accepted, the algorithm ends. The blocks are considered in order of descending magnitude, i.e., the first attempt is with the (off-diagonal) block with the largest Frobenius norm and so on. The final sparsity is heavily affected by the allowed increase of the **condest** estimate. We compare only their decadic logarithms. It is, the decision is based on the growth of the *order of magnitude* of the quantity rather than the value.

Algorithm 15 The **finer _ sieve _ cond** procedure

1: **function** FINER _ SIEVE _ COND(j, τ)
2: $D =$ assemble _ diag _ block(j);
3: Assemble the factor $\bar{L}_j$ defined as $\begin{bmatrix} \bar{L}_{j-1} & \\ 0 & D \end{bmatrix}$;
4: Estimate its condition number $C =$ **condest**$(\tilde{L}_j)$;
5: Find the block M_k from the j-th block-row that
has the largest Frobenius norm and that has not been added yet;
6: Set $\tilde{L}_j$ by adding the block M_k onto its position
in the j-th block-row of $\bar{L}_j$;
7: Compute $\tilde{C} =$ **condest**$(\tilde{L}_j)$;
8: **if** $\log_{10}(\tilde{C}) \geqslant \tau \log_{10}(C)$ **then**
9: Accept the k-th block in the block-row j;
10: GOTO line 5;
11: **end if**
12: **end function**

The third approach is based on the quantity *condest* of the already finalized factor, mentioned in Section 3.5. Scott and Tůma use a quantity named identically to the condition number estimator routine in MATLAB. We do not change the notation but we will make as clear distinction as possible. The MATLAB routine **condest** will be always emphasized in bold, while the quantity used by Scott and Tůma *condest* will be highlighted in italics. The notation conflict is to the best of our knowledge purely coincidental as both of these measure something different. The routine **condest** is based on estimating the 1-norm of the inverse of the given sparse square matrix, whereas the quantity *condest* measures the instability of a triangular solver of the given block-lower-triangular matrix.

The approach based on the quantity *condest* has been already mentioned above and the algorithm below follows the one presented in [74] as closely as possible. But there are some important differences. We will not describe the computation of the estimator g_j in any more details and settle for only a pseudo-code as the routine was already sketched in Algorithm 10 and then in (3.4). The interested reader is welcomed to look either into [74, Section 4, p.9-10] or into the submitted implementation.

Algorithm 16 The **finer _ sieve _ ST** procedure

```
 1: function FINER_SIEVE_ST(j, τ)
 2:     Initialize the structure of the j-th block-row of L̄_j;
 3:     Assemble the factor L̄_j;
 4:     Assemble the vector v_{j−1};
 5:     Find the block M_k from the j-th block-row, which increases g_j the least;
 6:     Add the block M_k onto its position in the j-th block-row
        of L̄_j and compute the updated instability factor g̃_j;
 7:     if g̃_j ⩾ τ g_j then
 8:         Accept the k-th block in the block-row j;
 9:         Update the estimate g_j = g̃_j;
10:         GOTO line 5;
11:     end if
12: end function
```

symb _ struct (*j*) The row structure is obtained by working with the structure of the quotient matrix of $\bar{L}_j$, i.e., the quotient matrix composed of zeros and ones, instead of the matrix $\bar{L}_j$ itself. The row structure is computed by the breadth-first search in the quotient graph of the quotient matrix. Although the MATLAB Graph toolbox contains a built-in routine for the search, we coded a simple search ourselves. Once the search is finished, the structure of the column of the already blocked matrix A is incorporated. Since this might be troublesome for dense or almost dense cases (since the structure would get dense as well), an alternative way to search for the row structure is proposed - only the leaves of the j-th row subtree are taken as the structure of the j-th row. This is motivated by the fact that from the structural point of view, the leaves are the vertices that *fully determine* the structure of the updates carried out in the complete factorization.

The column structure is obtained by adding the structure of all of the updating columns in one long vector (with possibly and probably repeating entries) and running the built-in MATLAB routine **unique**(queue, 'sorted'), which returns a vector of all the entries sorted from the lowest to the highest, without repetitions.

coarser _ sieve (*j*) This procedure performs the first row dropping that determines the structure of the upcoming column update. The purpose of this routine is to allow a rich enough row structure through to exploit the fact that the current active part of the factor is stored implicitly, which reduces the costs of the **update _ col _ by _ col** routine later on. Notice that both variants of the procedure (Algorithm 17, 18) *scale* the block column by the appropriate block from the j-th block-row. If the block is *very small* in some sense (either elementwise or normwise) relatively to the rest of the block-row, the update by the corresponding block-column may be much smaller than the rest of the updates in the same sense. In other words, heuristically that update is not as important and can be omitted, i.e., the block can dropped. This corresponds to the intuitive reasoning behind the threshold-based dropping in general. Here in order to obtain a really *coarse* sieve, the threshold has to be set quite low. The implementation corresponds to the one in Algorithm 14 (although the use is different).

update_col_by_col (j, k) In the scalar case this procedure carries out multiplication of two scalars and one subtraction - see Algorithm 5, lines 10 - 12. For the block scheme, this is formulated analogously, only the first block is transposed and both of the blocks are either stored in low-rank decomposition themselves or have to be retrieved from the low-rank approximation of the entire block-column, as in Figure 3.1 above. The output, i.e., the nonzero blocks of the column, is saved into CSR format and fetched to the routine **factorize_col** (j). The procedures are summarized in Algorithms 17 and 18 below.

Algorithm 17 The **update_col_by_col** procedure - structurally block-sparse

1: **function** UPDATE_COL_BY_COL(j, k)
2: X = assemble_subdiag_blckcol_A(j);
3: Set $col\_nnzs = |(\tilde{S}_2)_{*,k}|$;
4: **for** $ii = 2, \ldots, col\_nnzs$ **do**
5: i = determine_rowindex(ii);
6: Li = assemble_factorized_blck_pos(i, k);
7: Aij = assemble_unfactorized_blck_pos(j, k);
8: $X = X - Lik * Ajk^T$;
9: **end for**
10: Compress the block-vector X to CSR format and return it as output;
11: **end function**

Algorithm 18 The **update_col_by_col** procedure - block-column unit

1: **function** UPDATE_COL_BY_COL(j, k)
2: X = assemble_subdiag_blckcol_A(j);
3: Lk = assemble_factorized_blckcol(k);
4: Ajk = assemble_unfactorized_blck_pos(j, k);
5: $X = X - Lk * Ajk^T$;
6: Return X on the output;
7: **end function**

It is important to notice that the above implementation *requires storing of two block-triangular matrices* - the factorized one and the unfactorized one. This may result, in many instances, into large memory costs. There are two simple possibilities how to resolve this problem.

- One can go back to the procedure **factorize_col** in Algorithm 11 and force the computation of the Cholesky factorization of the diagonal block and scale the block-column by the inverse of this factor, i.e., replace MD^{-1} by $M * chol(D)^{-1}$ on lines 6, 4 in Algorithm 12 and 13 respectively. Consequently, the memory costs are halved. On the other hand, the procedure **factorize_col** can have a breakdown due to D not being SPD.

- One can save only the factorized block-triangular matrix and modify the update formula in the procedure **update_col_by_col** in Algorithm 17 and 18 respectively by

$$X = X - Lik * D * Ljk^T,$$
$$X = X - Lk * D * Ljk^T,$$

where one has $Ljk = AjkD^{-1}$. This can bring further instabilities into the algorithm as the diagonal blocks might be ill-conditioned.

Since our implementation is intended mainly for small, test problems, we simply use an additional memory.

We have summarized the main challenges we have met during the implementation as well as the building blocks. The analysis of the proposed preconditioning technique is in the following chapter.

4. Analysis of the proposed preconditioner

4.1 Costs analysis

As already mentioned, reasonable computational and memory costs are a must for a preconditioner. This section is devoted to this issue - we will discuss analysis of computational costs. But, it is important to distinguish between the possible final implementation and the current version of the code. The analysis here is done in regard to the code that computed the numerical examples (which will be presented in the next chapter), i.e., Algorithm 11. Let us emphasize the fact that this code can be further improved or modified and that these modifications can bring considerable speed-up in the sense of Section 3.6 (such instances will be highlighted).

The analysis will be done for two separate cases - first, focusing on each nonzero block separately (*structurally block-sparse*) and second, treating the block columns as a unit (*block-column unit*). First, each of the procedures in Algorithm 11 is analysed separately (for both of the instances) and in the end the final computational costs will be estimated. The cost of the k-th procedure will be denoted by η_k for the structurally block-sparse case and by ϑ_k for the structurally non-sparse case, e.g., costs of *factorize_col*(j) will be denoted $\eta_1(j)$ and $\vartheta_1(j)$, respectively, costs of *lr_approx_col*(j) will be denoted by $\eta_2(j)$ and $\vartheta_2(j)$, respectively and so forth. The notation is adopted from above, i.e., w_j and h_j stand for the *width* and *height* of the subdiagonal part of the block-column, i.e., the number of scalar columns and rows the subdiagonal part of the block-column contains.

factorize_col(j) This procedure has to multiply the block-column by the inverse of the diagonal block. Naturally, the inverse is not assembled and the code solves a sequence of linear systems with the diagonal block. This corresponds to the Gaussian elimination, i.e., LU factorization of the diagonal block followed by the forward and backward run instead. These costs are as follows.

$$\eta_1(j) = \overbrace{2/3w_j^3 + \mathcal{O}(w_j^2)}^{\text{GE of the diag block}} + \overbrace{\sum_{i\in(\tilde{S}_2)_{*,j}} 2w_j^2 \cdot w_i}^{\text{forward and backward run}}$$

$$\vartheta_1(j) = \underbrace{2/3w_j^3 + \mathcal{O}(w_j^2)}_{\text{GE of the diag block}} + \underbrace{2w_j^2 \cdot (h_j - w_j)}_{\text{forward and backward run}}$$

lr_approx_col(j) This procedure uses the randomized SVD algorithm at this moment, but, in general, one can plug in any desired low-rank procedure. Notice that for the structurally block-sparse case the costs are quite acceptable, even if some of the more costly procedures are employed, since *neither j nor h_j are present*. However, treating the block-column as a unit, the procedure has to be chosen more carefully. Here we also use the assumption *of the uniformly bounded numerical rank of the blocks*, i.e., we assume that the numerical rank of the block

that is being approximated is, at most, $p_{\max}$. For the computational costs, we refer to the work of Halko, Martinsson and Tropp [47, Section 1.4.1] obtaining the following.

$$\eta_2(j) = \sum_{i \in (\tilde{S}_2)_{*,j}} \mathcal{O}(w_i w_j \log(p_{\max}) + (w_j + w_i) p_{\max}^2$$
$$\vartheta_2(j) = \mathcal{O}\left((h_j - w_j) w_j \log(p_{\max}) + h_j p_{\max}^2\right)$$

finer _ sieve (j**)** This procedure performs the second row dropping that finalizes the incomplete factor. We have mentioned three different approaches. First, if **finer _ sieve _ norm** is used, the computational costs amount to computing the Frobenius norm of each of the nonzero blocks in the j-th row, i.e.,

$$\eta_3^{\text{norm}}(j) = \vartheta_3^{\text{norm}}(j) = \sum_{k \in (\tilde{S}_2)_{j,*}} 2 w_k w_j - 1$$

Since these blocks are kept in the low-rank format, it is possible here to relax the complexity by bounding $\|UV^T\|_F \leqslant \|U\|\|V\|_F$.

Second, if **finer _ sieve _ cond** is used, the computational costs amount to computing the Frobenius norm of each of the nonzero blocks, finding the largest of those and then evaluating **condest**$(\cdot)$, the latter two possibly several times. The bottleneck in this case is the **condest** routine. This routine complexity is, however, quite difficult to estimate and in many cases it is not even $\mathcal{O}(j)$. Hence, this could possibly be a significant bottleneck of the overall costs of the algorithm. Therefore, we consider it here rather as an illustration of a similar approach as the one of Scott and Tůma, which is, unfortunately, probably not applicable in practice for large problems.

At last, if **finer _ sieve _ ST** is used, the one has to carry out initialization, computing matrix-vector products with the blocks in the j-th block-row in order to evaluate the block version of (3.4), computing 1-norm of a vector of length w_j and search for the largest entry in a scalar vector of length at most j. Assuming either no initialization or initialization by **norm**, these costs amount to the following.

$$\eta_3^{\text{ST}}(j) = \vartheta_3^{\text{ST}}(j) = \sum_{k \in (\tilde{S}_2)_{j,*}} (2w_k - 1) p_{\max} + (2p_{\max} - 1) w_j - 1 + \mathcal{O}(1)$$

symb _ struct (j**)** This procedure first performs a breadth-first search in the graph of the currently completed part of the incomplete factor. For a breadth-first search in a general graph $G(V, E)$ one has the generally known worst case bound of $\mathcal{O}(|V| + |E|)$. Notice that the column structure computation does not need to be considered here as its cost is certainly inferior with respect to the costs of the actual update, since the pattern is the same. Also, note that there is no difference between the two considered cases as the quotient graph of the finalized factor is the same in both of them.

We will impose the structural sparsity condition at this point, fixing the number of blocks per block-row (and block-column) to be constant in the final incomplete factor. This can restriction is based on the paper of Lin and Moré [59] and

is a common one when a structurally (block) sparse incomplete Cholesky factor is desired.

Denoting $G(\bar{L}_{j-1}) = (V_{\bar{L}_{j-1}}, E_{\bar{L}_{j-1}})$ the graph of interest, one can write $|V_{\bar{L}_{j-1}}| = j$ and $|E_{\bar{L}_{j-1}}| \leqslant \text{const} \cdot j$ and the estimate follows.

$$\eta_4(j) = \vartheta_4(j) = \mathcal{O}(j)$$

coarser _ sieve (j) This procedure performs the first row dropping that determines the structure of the upcoming column update by threshold-based dropping based on comparison with the Frobenius norm of the diagonal block. To perform this dropping, one only needs to compute the Frobenius norm of all of the nonzero blocks in the j-th block-row, which (in both cases) amounts to the same costs.

$$\begin{aligned}\eta_5(j) &= \sum_{k \in (S_2)_{j,*}} 2w_k w_j - 1\\ \vartheta_5(j) &= \sum_{k \in (S_2)_{j,*}} 2w_k w_j - 1\end{aligned}$$

update _ col _ by _ col (j, k) The process of the update consists of multiplication of the correct part of the k-th block-column (from block-index j below) by a single block, i.e., multiplication of matrices of dimensions $w_k \times h_j$ and $w_k \times w_j$, and consequent summation of two matrices of dimension $w_j \times h_j$. However, all of the matrices are stored in low-rank format, with the rank bounded by $p_{\max}$ from above. In the case of the structurally block-sparse format, naturally only the nonzero multiplications are performed. Consequently, number of flops per each run of this procedure can be estimated as follows.

$$\begin{aligned}\eta_6(j) &= \overbrace{\sum_{i \in (\tilde{S}_2)_{*,k}} (2w_i - 1)w_k p_{\max} + (2p_{\max} - 1)w_i w_j}^{\text{multiplication}} + \overbrace{\sum_{i \in (\tilde{S}_2)_{*,k}} w_k w_i}^{\text{summation}}\\ \vartheta_6(j) &= \underbrace{(2w_k - 1)p_{\max} w_j (2p_{\max} - 1)h_j w_j +}_{\text{multiplication}} + \underbrace{h_j w_j}_{\text{summation}}\end{aligned}$$

Overall computational costs Here, we will sum up the computational costs for each of the cases. But prior to that let us consider two *additional assumptions.*

(A1) The sizes of the block-columns (and block-rows) are uniformly bounded, i.e., there is a natural number W such that for all $1 \leqslant j \leqslant \nu$ one has $w_j \leqslant W$;

(A2s) The number of the nonzero blocks for each column after it has been updated is uniformly bounded as well, i.e., after any block-column update, the block-colum has at most Knonzero blocks (only for the structurally block-sparse case).

In order to evaluate the overall computational costs let us first evaluate only lines 9-11, i.e., the entire updating process, denoted by $\bar{\eta}_6$ and $\bar{\vartheta}_6$ for structurally

block-sparse and block-column unit respectively. Starting with $\bar{\eta_6}$, one can use (A1) to obtain

$$\bar{\eta_6}(j) = \sum_{k \in (\tilde{S}_2)_{j,*}} p_{\max} w_j \sum_{i \in (\tilde{S}_2)_{*,k}} 2(w_k + w_i) - 1 \leqslant \sum_{k \in (\tilde{S}_2)_{j,*}} 4W^3 p_{\max} |(\tilde{S}_2)_{*,k}|.$$

Employing the second assumption (A2s) one can bound the number of nonzero blocks in the j-th column by jK, obtaining

$$\bar{\eta_6}(j) \leqslant 4W^3 p_{\max} jK.$$

However, note that this bound is rather pessimistic as it assumes *no overlap in the column structure replication.* If (A2s) is strengthened, e.g., assuming that *after the updates each of the column contains at most K nonzero blocks*, the above bound becomes *independent* on j and therefore further asymptotically decrease the computational costs.

Considering the block-column unit alternative, one obtains the following bound on $\bar{\eta_6}(j)$.

$$\begin{aligned} \bar{\vartheta_6}(j) &= \sum_{k \in (\tilde{S}_2)_{j,*}} (2w_k - 1) p_{\max} w_j + (2p_{\max} - 1) h_j w_j + h_j w_j \\ &\leqslant W \cdot \sum_{k \in (\tilde{S}_2)_{j,*}} 2W p_{\max} + 2p_{\max} h_j W + h_j. \end{aligned}$$

In order to proceed any further, one needs to bound h_j. i.e., the number of the scalar rows in the j-th block-column (subdiagonal part). Since $h_j = \sum_{l<j} w_l$, the worst case scenario gives $h_j = n - j + 1$. However, in the mentioned case when all of the blocks have identical dimension, the quantity h_j scales linearly with j. This is more or less unavoidable if one wants to treat the columns as units. Analogously to the bound on $\bar{\eta_6}$, one can write the following.

$$\bar{\vartheta_6}(j) \leqslant W \sum_{k \in (\tilde{S}_2)_{j,*}} 2W p_{\max} + 2p_{\max} h_j W + h_j \leqslant 2W^3 p_{\max} (n - j + 2) |(\tilde{S}_2)_{j,*}|.$$

Therefore, in order to avoid the $\mathcal{O}(n^3)$ complexity of the complete factorization, one has to bound $|(S_2)_{j,*}|$ independently of j, i.e., assume that already the *coarser sieve will considerably sparsify the row structure.*

Having all of the subroutines analysed, the overall computational costs are summarized in the following theorem.

Theorem 4.1.1. *Let A be an n-by-n SPD matrix and assume that the blocking procedure in Algorothm 11 resulted in a block structure with each of the diagonal blocks having dimension bounded by W. Then the following holds true.*

(i) Let the number of nonzero blocks in each of the block-column of blocked A is bounded by a constant K independently of the block-column index. Assume that the structurally block-sparse *version of Algorithm 11 has been carried out without a breakdown and that the final factor is block sparse, i.e., number of nonzero blocks per block-row is constant and independent on the block-row index. Then the computational costs asymptotically amounts to $\mathcal{O}(n^2)$.*

(ii) Consider the block-column unit *version of Algorithm 11 and assume it has been carried out without a breakdown. Assume that* $|(S_2)_{j,*}|$ *can be bounded by some constant* independent on j *for all block-column indices* $j = 2, \ldots, \nu$*, the computational costs asymptotically amounts to* $\mathcal{O}(n^2)$*.*

Proof. The costs of each of the subroutines of Algorithm 11 have been bounded linearly in j for both of the considered cases. Therefore, the overall costs amount to

$$\sum_{j=1}^{\nu} \mathcal{O}(j) = \mathcal{O}(\nu^2) = \mathcal{O}(n^2),$$

where the last equality follows from the definition of ν. □

4.2 Accuracy, stability and convergence

The importance of analysis have been already highlighted throughout this thesis. There are several options how to approach this task. Having the incomplete Cholesky factor $\bar{L}$ of A, one might consider its *accuracy*, i.e., the quantity $\|\bar{L}\bar{L}^T - A\|$, or its *stability*, i.e., the quantity $\|\bar{L}^{-1}A\bar{L}^{-T} - I\|$. However, it is a well-known fact that reasonable bounds on these quantities do not necessarily imply good convergence of CG and vice versa. In most of our experiments, the inaccuracy was quite large even though the results were sometimes positive. The correct way to approach the analysis of the efficiency of a given preconditioner has to involve the iterative method, i.e., it has to take into account the number of iterations needed for the convergence as well as costs per iteration. Analysis of those quantities *even for the unpreconditioned CG* in finite precision is very challenging and a great deal of work have gone into it. Nice overview of the literature covering this topic is in the PhD thesis of Carson [14, Section 2.5]. Naturally, incorporating the preconditioning adds other challenges. The preconditioned CG algorithm in finite precision was discussed by, e.g., Strakoš and Tichý in [75], [76].

The main challenge we face during the analysis that needs to be highlighted here is that both the droppings in our preconditioner affect each other. The second one is even *very hard to analyse*, because of the unpredictability of the estimators. Although the analysis *is much needed*, the task remains as a challenge for future work.

4.3 IFCM vs. CFIM

The purpose of this section is to point out to an interesting concept proposed by Gilbert et al. [37]. In this project, the authors suggest a conceptually new framework for structurally sparse preconditioners, which they abbreviate by *complete factorization of incomplete matrices* (CFIM) and put it into the contrast with the classical way of preconditioning, which they summarize by *incomplete factorization of complete matrix* (IFCM). The difference is obvious from the title - while the usual preconditioning techniques tend to preserve the given matrix and modify the factorization in order to build an efficient preconditioner, Gilbert et al. proposed to keep the factorization intact and to perform it on a modified matrix.

The main advantages they highlighted were the possibility to change and prescribe the matrix structure *so that the complete factorization can run in parallel* and *avoid massive fill-in*. One can immediately see that in this way the memory allocation can be done prior to the factorization itself and the matrix structure can be determined *so that the memory costs will perfectly fit the given conditions*. Moreover, *one could use the effective libraries for parallel direct methods* and hence the work on implementation could be drastically simplified. The efficiency should be kept by *suitable interplay between the structural and algebraic properties of the modified matrix*. Although this idea sounds very promising, we have not found any further information related to this project, nor any published results in this direction.

We mention it here because the core idea is connected to our approach, despite the reasoning being different. In our case, the preconditioning technique uses the classical tools of the incomplete factorization, e.g., threshold-based dropping, but the *structural part of the factorization can be viewed as a complete factorization of an incomplete matrix*, since the governing mechanisms correspond to the complete Cholesky factorization, only used on an incomplete factor. The motivation is, however, completely different. Rather than focusing on parallelism and prescription of the sparsity pattern in advance, we employ these techniques to enable implicit storing of the current active part of the factor.

5. Numerical experiments

The goal of the last chapter is to present several examples, showing that the ideas and approaches given above are applicable and may result in decent preconditioners. This is, in our opinion, even more important due to the lack of analysis. In order to validate further research, the numerical examples should be promising, at least to some extent.

All of the experiments were performed on the MATLAB R2015a software. To give a comparison, the built-in routine **ichol** was used to produce *modified Cholesky factorization with zero fill-in* and *incomplete Cholesky factorization with threshold dropping* - setting "ichol(sparse(A), struct('michol','on','type','nofill'))" and "ichol(sparse(A), struct('type','ict','droptol',1e-3))" , respectively. The right-hand side in all of the examples is set so that the exact solution is the vector of ones. Initial guess is fixed to be the zero vector and the stopping criterion is set to

$$\frac{\|r_k\|}{\|r_0\|} \leqslant 10^{-10}. \tag{5.1}$$

The maximal number of CG iterations is set to $maxit = 300$.

In total, five different preconditioning techniques have been proposed here, all of them following the Algorithm 11, but employing different strategy for the routine **finer_sieve**. The options for setting the **finer_sieve** procedure are

- the **threshold row dropping**, i.e., Algorithm 14;
- the **condition number estimate**, i.e., Algorithm 15;
- the **instability factor-based**, i.e., Algorithm 16.

Moreover, the last one, following the proposal of Scott and Tůma, has three different possible initializations - by threshold dropping, condition number estimate, or without any initialization. Fixing the **finer_sieve** procedure to one of the listed options, the resulting preconditioners will be denoted by "norm" "cond" "STn" "STc" "STp" [1]. In general, we refer to the group of preconditioners as *data-sparse incomplete Cholesky* preconditioners (DSIC).

Overall, our experience is that if no breakdown occurs in the built-in routines, then they tend to outperform our preconditioner in terms of number of iterations of preconditioned CG, in some cases significantly. On the other hand, in most of the cases the number of iterations of preconditioned CG is comparable. The used test matrices are freely available at the SuiteSparse Matrix Collection [20].

The rest of this chapter is divided into two sections, one for each of the considered blocking variants, and in each is presented the performance of the preconditioned CG method as well as the structural properties of the preconditioner for several matrices. Naturally, the additional user-defined parameters will be specified as well.

[1] The abbreviation ST stands for Scott and Tůma, "p" stands for *pure*, i.e., without initialization.

5.1 Structurally block-sparse matrices

We will consider one problem *in full detail* and summarize the rest in a table below. The parameters in our preconditioner were set as follows.

- **blocking** - the parameter τ_{block} for the blocking algorithm of Saad (see (2.4)) was taken as $\tau_{\text{block}} = 0.75^2 \approx 0.56$;
- **coarser dropping** - the parameter τ_{coarse} for the coarser sieve based on threshold dropping has been taken as $\tau_{\text{coarse}} = 10^{-4}$;
- **finer dropping** - the parameter $\tau_{\text{fine\_norm}}$ for the finer sieve based on threshold dropping has been taken as $\tau_{\text{fine\_norm}} = 10^{-3}$;
- **finer dropping** - the parameter $\tau_{\text{fine\_cond}}$ for the finer sieve based on the MATLAB routine **condest** has been taken as $\tau_{\text{fine\_cond}} = 1.1$;
- **finer dropping** - the parameter $\tau_{\text{fine\_ST}}$ for the finer sieve based on instability factor g_j has been taken as $\tau_{\text{fine\_ST}} = 1$;
- **initialization of ST** - the parameter $\tau_{\text{init\_ST}}$ for initialization of the finer sieve based on instability factor g_j has been taken as $\tau_{\text{init\_ST}} = 1$ for both threshold dropping and **condest**-based initialization;

The results for the matrix `Trefethen_150` are illustrated in three separate figures. First, Figure 5.1 shows the nonzero structure of the matrix A, its Cholesky factor and then the incomplete Cholesky factor with threshold dropping. In the modified incomplete Cholesky factor with zero fill-in occurred a breakdown. Second, Figure 5.2 shows the nonzero structures of the computed preconditioners by our code and the structure used for updates, i.e., the structure after the coarser sieve dropping. Let us emphasize that the update structure should be understood *row-wise*, i.e., the nonzero pattern of the j-th row here shows, which columns were used to update the j-th column. However, the structure of the updating columns *is not presented*! Finally, Figure 5.3 shows the number of iterations of the preconditioners.

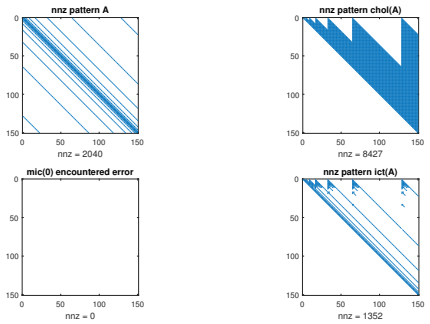

Figure 5.1: The nonzero structure of the original matrix A as well as its complete Cholesky factor and its incomplete counterparts - modified with no fill-in and with threshold dropping.

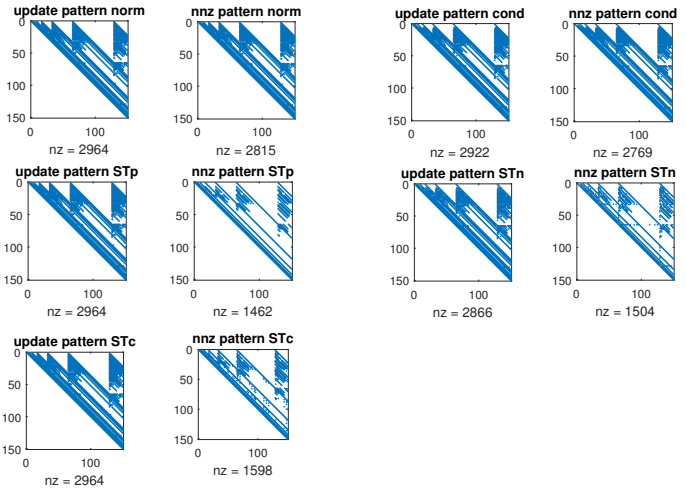

Figure 5.2: The nonzero structure of the coarser (on the left) and finer (on the right) sieve dropping result. In comparison to the complete Cholesky factor, there is a substantial dropping in all of these and even among them, there is a substantial difference between the one with least nonzero entries and its opposite.

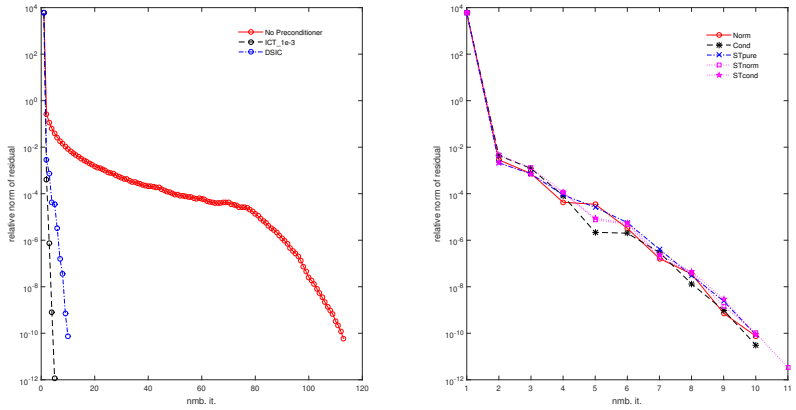

Figure 5.3: The convergence behaviour of the preconditioned CG method with the five options described above is on the left. On the right are convergence curves of unpreconditioned CG and preconditioned CG using the best performing preconditioner from the left (in terms of number of iterations) and the incomplete Cholesky with threshold dropping preconditioner.

The results can be summarized as follows. The purpose of the double sparsification was fulfilled as the update pattern contains almost twice the number of nonzero entries in comparison to the final incomplete factor. Also, the convergence was obtained in reasonable number of iterations that is both comparable to the number of iterations needed with the incomplete Cholesky factorization with threshold dropping and is significantly lower than the number of iterations the unpreconditioned CG method needed to satisfy the stopping criterion (5.1).

It is important to emphasize that *no blocking occurred in this particular case*, i.e., all of the blocks are of size 1-by-1. From one point of view, this does not represent the full picture, as only a part of the above proposed approach have

been exploited. On the other hand, it shows that the structural computation proposal *is not limited to the block variant*. Although we see the final aim *in block variant*, this suggests that the structural proposal could be useful on its own.

We run the code also for other matrices from classes "Trefethen" and "mesh" with identical setting as above. Since there is several versions and in most of the cases their performance among themselves was comparable, we restrict ourselves to present the results in Table 5.1 below. Each of the matrices is referred to by name from the SuiteSparse Matrix collection and the number of nonzero entries of the complete Cholesky factor is given as well. The column "DSIC type" clarifies, which of the preconditioners resulted in the fastest convergence and is followed by the number of nonzeros contained the incomplete factor and by the number of iterations of preconditioned CG needed. Last column gives the number of iterations needed for the incomplete Cholesky factorization with threshold dropping MATLAB routine with threshold dropping 10^{-3}. The modified incomplete Cholesky factorization had a breakdown in most cases and therefore we do not include it into the comparison.

Let us stress out that the purpose of the following table is to show the promises of the proposed preconditioners. In other words, the focus is on showing that the approaches *are viable*, even though they may not be best suited for the particular problems. Therefore, the table does not give a full comparison and number of important metrics is omitted.

Name	Dimension	nnzs chol(A)	DSIC type	nnzs DSIC(A)	nmb. it. DSIC pCG	nmb. it. ICT pCG
Trefethen_20	20	169	STp	76	11	5
Trefethen_150	150	8427	cond	1598	10	6
Trefethen_200	200	14877	STc	2217	10	5
Trefethen_300	300	33 409	STp	3266	10	4
Trefethen_500	500	84809	STn	5180	10	5
Trefethen_700	700	184 337	norm	14564	9	6
mesh1e1	48	541	STn	163	15	5
mesh2e1	306	12 464	norm	1998	25	10
mesh2em5	306	12 464	norm	1604	19	6
mesh3e1	289	11 049	STp	545	21	6
mesh3em5	289	11 049	cond type	290	20	5

Table 5.1: A summary of performance of the DISC-type of preconditioners on some test matrices form the SuiteSparse Matrix collection.

5.2 Block-column unit matrices

As we have already mentioned, for this case we have implemented the $\mathcal{H}$-matrix format with the strong admissibility condition (see (2.2)), i.e., the HODLR format, and adopted the uniform blocking[2] given by the block of the smallest dimension, i.e., the diagonal blocks of the format, see Figure 5.4 below.

[2]The matrix is blocked into square blocks of equal dimension.

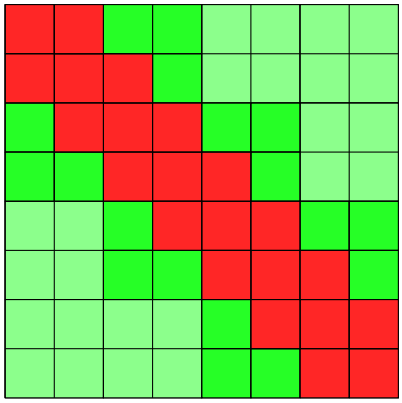

Figure 5.4: The figure shows the uniform matrix partitioning corresponding to the HODLR format from Figure 2.8.

Notice that in this form, one has to assume that the dimension n of the original matrix is written as a power of the dimension of the diagonal block, i.e., it should be $n = MinDim^k$ for some k. This could be relaxed, see [12], to allow for any dimension n in general. However, we have not managed to implement that relaxation in our code. Therefore, we will *consider only the leading principal submatrix of size $\tilde{n}$ of the original matrix*, where $\tilde{n} \leqslant n$ is chosen as large as possible, i.e., so that

$$\tilde{n} = MinDim^k \quad \text{for as large } k \text{ as possible.}$$

Considering the *reduced matrix* $A(1:\tilde{n}, 1:\tilde{n})$ and the uniform blocking, we will now treat the block-columns as individual items[3]. One can see that for *structurally sparse matrices with limited fill-in this will result in very inefficient approach*, because the block-column unit approach stores the whole blocks in a low-rank format, i.e., it does not distinguish zero and nonzero blocks during the computation. Therefore we decided to test *on modified matrices with added nonzeros of small magnitudes in comparison to the rest of the matrix row*. To be more specific, a modification matrix M is defined row by row as

$$M(i,:) = 0.5 * (1e-3) * \|A(i,:)\|_1 * randn(1,n)$$

and then we have considered the matrix $\tilde{A} = A + M + M^T$, provided it was SPD.

Also, in order to simulate the blockwise rank-deficiency, we first projected the given matrix to the set of $\mathcal{H}$-matrices, i.e., each block with respect to the original HODLR format was truncated. Those two modifications should, in our opinion, create a well-suited problem for the proposed block-column unit routines, although in many cases almost unrelated to the original one. However, let us emphasize that the modifications are not part of our proposal, i.e., we do not suggest to always modify the problem as we did prior to the computation. The goal was simply to create a problem for which our approach would be well-suited. In reality, the small-magnitude error is often already present in the matrix as well as the rank-deficiency.

The parameters in our preconditioner were set as follows.

[3]Naturally, one can easily modify this scheme in many ways, e.g., consider the uniform blocking with respect to a *coarser blocking*, i.e., defining the size of the block columns to be twice the size of the diagonal block from the HODLR format. We referred to this block-column unit approach as *$\mathcal{H}$-matrix slicing*.

- **blocking** - the dimension of the diagonal blocks $MinDim$ for the blocking algorithm was taken as $MinDim = 4$. The $p_{\max}$ bound is set to $p_{\max} = MinDim - 1$;
- **coarser dropping** - the parameter τ_{coarse} for the coarser sieve based on threshold dropping has been taken as $\tau_{\text{coarse}} = 10^{-4}$;
- **finer dropping** - the parameter $\tau_{\text{fine\_norm}}$ for the finer sieve based on threshold dropping has been taken as $\tau_{\text{fine\_norm}} = 10^{-3}$;
- **finer dropping** - the parameter $\tau_{\text{fine\_cond}}$ for the finer sieve based on the MATLAB routine **condest** has been taken as $\tau_{\text{fine\_cond}} = 1.1$;
- **finer dropping** - the parameter $\tau_{\text{fine\_ST}}$ for the finer sieve based on instability factor g_j has been taken as $\tau_{\text{fine\_ST}} = 1$;
- **initialization of ST** - the parameter $\tau_{\text{init\_ST}}$ for initialization of the finer sieve based on instability factor g_j has been taken as $\tau_{\text{init\_ST}} = 1$ for both threshold dropping and **condest**-based initialization;

The results for the matrix `Trefethen_150` are illustrated in three separate figures, analogously to the previous section. First, Figure 5.5 shows the nonzero structure of the matrix A, its Cholesky factor and then of the modified incomplete Cholesky factor with zero fill-in and incomplete Cholesky factor with threshold dropping. Second, Figure 5.6 shows the nonzero structures of the computed preconditioners by our code and the structure used for updates. Let us again stress out that this is to be understood *row-wise*. It is, the nonzero update pattern of the j-th row shows, which of the columns were used to update the j-th one. However, the structure of the updating columns *is not presented*! At last, Figure 5.7 shows the number of iterations of the preconditioners.

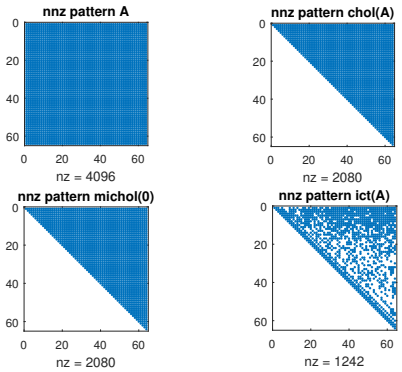

Figure 5.5: The nonzero structure of the original matrix A as well as its complete Cholesky factor and its incomplete counterparts - modified with no fill-in and with threshold dropping.

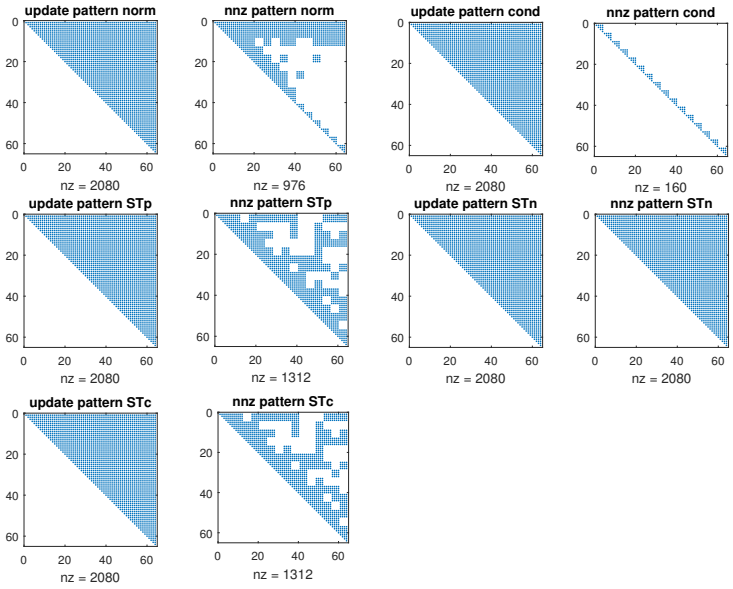

Figure 5.6: The nonzero structure of the coarser (on the left) and finer (on the right) sieve dropping result. Recall that the *update pattern* matrices have to be understood *row-wise*. That is, the j-th block-row shows *which of the block-columns updated the j-th block column*. This does not imply that those updates are dense.

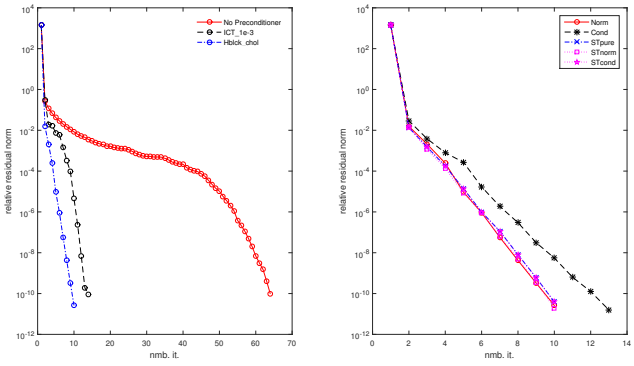

Figure 5.7: The convergence behaviour of the preconditioned CG method with the five options described above is on the left. On the right are convergence curves of unpreconditioned CG and precondtioned CG using the best performing preconditioner from the left (in terms of number of iterations) and the incomplete Cholesky with threshold dropping preconditioner.

The convergence was obtained in reasonable number of iterations that is both comparable to the number of iterations needed with the incomplete Cholesky factorization with threshold dropping and is significantly lower than the number of iterations the unpreconditioned CG method needed to reach the stopping criterion (5.1). Moreover, as one can see in Table 5.2 below, the performance of the incomplete Cholesky factor with threshold dropping as a preconditioner is in some cases *considerably worse* than the one of our preconditioners.

Notice that the update pattern is *dense*, since the structure of the matrix has been made *dense* by the preprocessing. Due to Lemma 3.4.2 the structure of the update *always contains the structure of the original matrix*. This might be, in practice, rather inconvenient for matrices that are not structurally sparse. In order to further sparsify, one needs *only to approximate the row structures*. This will not be pursued any further here, but it is an interesting direction for future work. From our point of view, this only supports Observation 3.5.1 and shows *how powerful the additional updates are, even if they are eventually discarded from the final factor.*

We run the code out also for other matrices from classes `Trefethen` and `mesh` with identical setting as above. Because for the matrices `Trefethen` the principal matrices are identical, e.g., `Trefethen_150` = `Trefethen_200`$(1:150, 1:150)$, we considered additional test matrices in Table 5.2.

Since there is several versions and in most of the cases their performance among themselves was comparable, we present the result in Table 5.2 below. Each of the matrices is referred to by name from the SuiteSparse Matrix collection and the number of nonzero entries of the complete Cholesky factor is given as well. The column "DSIC type" clarifies, which of the preconditioners resulted in the fastest convergence and it is followed by the column with nonzero counts of the incomplete factor and by the number of iterations of preconditioned CG needed. Last column gives the number of iterations needed for the incomplete Cholesky factorization with threshold dropping MATLAB routine with threshold dropping 10^{-3}. The modified incomplete Cholesky factorization had a breakdown in most cases and therefore we do not include it into the comparison.

Name	trunc_dim	nnzs chol(A)	DSIC type	nnzs DSIC(A)	nmb. it. DSIC pCG	nmb. it. ICT pCG
`Trefethen_150`	64	2080	norm	976	10	14
`Trefethen_300`	256	32 896	norm	4320	11	31
`mesh2e1`	256	32 896	norm	20 048	26	48
`mesh2em5`	256	32 896	STp	17 920	14	16
`mesh3e1`	256	32 896	STp	18 144	18	16
`mesh3em5`	256	32 896	STc	17 776	14	12
`bcsstk04`	64	2080	STp	1168	28	no convergence
`Journals`	64	2080	STp	1600	22	18

Table 5.2: A summary of performance of the DISC-type of preconditioners on some test matrices form the SuiteSparse Matrix collection. The test matrix has been taken as the principal leading submatrix of order *trunc_dim*.

Conclusion

The *data-sparsity* of matrices has started to attract a significantly more attention in the last decade. This notion is recalled in the first chapter, together with the notion of the classical structural sparsity and brief summary of the structurally sparse complete Cholesky factorization. The Krylov subspace methods and the method of conjugate gradients in particular, are introduced and the incomplete Cholesky factorization is recalled as a particularly important representative of preconditioning techniques for the CG method.

Focusing on the term of data-sparsity, we have given a compressed overview of methods (and literature) currently connected to the data-sparse matrices. Chapter 2 summarizes the commonly used low-rank techniques and their possible alternatives and also the commonly used matrix blocking techniques and derived matrix formats. This is complemented by the beginning of chapter 3, where the classical scheme of the incomplete Cholesky factorization methods for data-sparse matrices is pointed out.

Based on those, a new and complementary way of data-sparsity utilization in the context of preconditioning has been proposed, focusing on the CG method. Unlike the commonly used, recurrently-based methods, we retrieve the classical Cholesky factorization scheme and modify the symbolic part of the factorization. This is done in such a way that the implicit storing of the current active part of the factor is possible. This approach also allows to be combined with the classical incomplete Cholesky factorization, e.g., structural dropping to reduce the memory and computational costs of the entire process. This requires the facts from both the graph theory of the structurally sparse Cholesky factorization and the data-sparsity techniques.

Together with theoretical results, a preconditioning technique has been proposed and implemented in MATLAB R2015a. It has been tested on several test matrices from the SuiteSparse Matrix collection. The results are, in our opinion, promising enough to validate a further interest in this direction, especially in the theoretical analysis of the proposed preconditioner coupled with the CG method. Such analysis together with further development of the code remains a challenge for future work.

Bibliography

[1] A. V. Aho, M. R. Garey, and Ullman J. D. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1:131–137, 1972.

[2] S. Ambikasaran. *Fast Algorithms for Dense Numerical Linear Algebra and Applications*. phdthesis, Stanford University, August 2013.

[3] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving multifrontal methods by means of block low-rank representations. Technical Report RR-8199, Inria, January 2013.

[4] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing*, 16:1404–1411, 1995.

[5] J. Ballani and D. Kressner. Matrices with hierarchical low-rank structures. In *Exploiting Hidden Structure in Matrix Computations: Algorithms and Applications*, pages 161–209. Springer, 2016.

[6] M. Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86:565–589, 2000.

[7] M. Bebendorf and S. Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70:1–24, 2003.

[8] M. Benzi. Preconditioning techniques for large linear systems: A Survey. *Journal of Computational Physics*, 182:418–477, 2002.

[9] M. Benzi. Preconditioning techniques for large linear systems: Survey. *Journal of Computational Physics*, 182:418–477, 2002.

[10] M. Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing*, 25:86–103, 2001.

[11] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing*, 27:1627–1650, 2006.

[12] S. Börm, L. Grasedyck, and W. Hackbusch. Hierarchical Matrices. *Max-Planck-Institut für Mathematik in den Naturwissenschaften, Leipzig*, Lecture notes 21, June, 2006.

[13] W. L. Briggs, S. F. McCormick, and V. E. Henson. *A Multigrid Tutorial*. SIAM, second edition edition, 2000.

[14] E. C. Carson. *Communication-Avoiding Krylov Subspace Methods in Theory and Practice*. phdthesis, University of California at Berkeley, August 2015.

[15] E. C. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018.

[16] T. F. Chan. Rank-revealing QR-factorizations. *Linear Algebra and Its Application*, 88-89:67–82, 1987.

[17] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19:995–1023, 1998.

[18] A. Civril and M. Magdon-Ismail. Exponential inapproximability of selecting a maximum volume sub-matrix. *Algorithmica*, 65:159–176, 2013.

[19] T. A. Davis et al. SuiteSparse: A suite of sparse matrix software. http://www. suitesparse. com, 2015.

[20] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1, 2011.

[21] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

[22] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, First edition, 1997.

[23] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, third edition, 2017.

[24] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT Numerical Mathematics*, 29:635–657, 1989.

[25] I. S. Duff and J. K Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[26] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale Sparse Matrix Package I: The Symmetric Codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[27] B. Engquist and L. Ying. Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation. *Communications on Pure and Applied Mathematics*, 64:697–735, 2011.

[28] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, Second edition, 2010.

[29] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester, Second edition, 1999.

[30] M. J. Gander and F. Nataf. AILU: A preconditioner based on the analytic factorization of the elliptic operator. *Numerical linear algebra with applications*, 7:543–567, 2000.

[31] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[32] A. George. On finding and analyzing the structure of the Cholesky factor. *Algorithms for Large Scale Linear Algebraic Systems*, pages 73–106, 1998.

[33] A. George and W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Inc., New Jersey, 1981.

[34] P. Ghysels, X. S. Li, and G. Chávez. STRUMPACK library. http://portal.nersc.gov/project/sparse/strumpack/.

[35] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38:S358–S384, 2016.

[36] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 15:62–79, 1994.

[37] J. R. Gilbert, E. Ng, B. W. Peyton, and P. Raghavan. Portable Parallel Preconditioning: Project Proposal.

`http://www2.mta.ac.il/~hillel/sites+papers/iterative_solvers.htm`

.

[38] G. Golub. Numerical methods for solving linear least squares problems. *Numerische Mathematik*, 7:206–216, 1965.

[39] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14:403–420, 1970.

[40] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Third edition, 1996.

[41] S. A. Goreinov, E. E. Tyrtyshnikov, and N. L. Zamarashkin. A theory of pseudoskeleton approximations. *Linear Algebra and Its Applications*, 261:1–21, 1997.

[42] L. Grasedyck, R. Kriemann, and S. Le Borne. Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11:273–291, 2008.

[43] L. Grasedyck, R. Kriemann, and S. Le Borne. Domain decomposition based $\mathcal{H}$-LU preconditioning. *Numerische Mathematik*, 112:565–600, 2009.

[44] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[45] M. Gu and S. C. Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17:848–869, 1996.

[46] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[47] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53:217–288, 2011.

[48] P. C. Hansen. *Discrete Inverse Problems: Insight and Algorithms*. Fundamentals of Algorithms. SIAM, 2010.

[49] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[50] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29:575–596, 1987.

[51] N. J. Higham and F. Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM Journal on Matrix Analysis and Applications*, 21:1185–1201, 2000.

[52] Y. P. Hong and C.-T. Pan. Rank-revealing QR factorizations and the singular value decomposition. *Mathematics of Computations*, 58:213–232, 1992.

[53] R. A. Horn and Ch. R. Johnson. *Matrix Analysis*. Second Edition. Cambridge University Press, Cambridge, 2012.

[54] I. E. Kaporin. High quality preconditioning of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$-decomposition. *Numerical Linear Algebra with Applications*, 5:483–509, 1998.

[55] I. Konshin, M. Olshanskii, and Y. Vassilevski. LU factorizations and ILU preconditioning for stabilized discretizations of incompressible Navier-Stokes equations. *Numerical Linear Algebra with Applications*, 24:e2085, 15, 2017.

[56] R. Kriemann and S. Le Borne. $\mathcal{H}$-FAINV: Hierarchically factored approximate inverse preconditioners. *Computing and Visualization in Science*, 17:135–150, 2015.

[57] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53, 1952.

[58] J. Liesen and Z. Strakoš. *Krylov Subspace Methods: Principles and Analysis*. Oxford University Press, Oxford, first edition edition, 2013.

[59] C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. *SIAM Journal on Scientific Computing*, 21:24–45, 1999.

[60] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Aanalysis and Applications*, 11:134–172, 1990.

[61] J. Málek and Z. Strakoš. *Preconditioning and the Conjugate Gradient Method in the Context of Solving PDEs*. SIAM Spotlight series. SIAM, first edition edition, 2014.

[62] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34:473–497, 1980.

[63] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Springer Science & Business Media, 2013.

[64] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[65] C.-T. Pan. On the existence and computation of rank-revealing LU factorizations. *Linear Algebra and Its Application*, 316:199–222, 2000.

[66] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press, 1984.

[67] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations. In J. K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, 1971.

[68] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software*, 42:27, 2016.

[69] Y. Saad. Finding exact and approximate block structures for ILU preconditioning. *SIAM Journal on Scientific Computing*, 24:1107–1123, 2003.

[70] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. SIAM, second Edition edition, 2003.

[71] Y. Saad and Schultz M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[72] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9:359–378, 2002.

[73] J. Scott and M. Tůma. On positive semidefinite modification schemes for incomplete Cholesky factorization. *SIAM Journal on Scientific Computing*, 36:A609–A633, 2014.

[74] J. Scott and M. Tůma. Improving the stability and robustness of incomplete symmetric indefinite factorization preconditioners. *Numerical Linear Algebra with Applications*, 24(5), 2017.

[75] Z. Strakoš and P. Tichý. On error estimation in the conjugate gradient method and why it works in finite precision computations. *Electronic Transactions on Numerical Analysis*, 13(56-80):8, 2002.

[76] Z. Strakoš and P. Tichỳ. Error estimation in preconditioned conjugate gradients. *BIT Numerical Mathematics*, 45:789–817, 2005.

[77] M. Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and Its Application*, 154-156:331–353, 1991.

[78] U. Trottenberg, C. W. Oosterlee, and A. Schuller. *Multigrid*. Academic Press, first edition edition, 2001.

[79] A. D. Tuff and A. Jennings. An iterative method for large systems of linear structural equations. *International Journal for Numerical Methods in Engineering*, 7:175–183, 1973.

[80] E. Tyrtyshnikov. Incomplete cross approximation in the mosaic-skeleton method. *Computing*, 64:367–380, 2000.

[81] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 12:631–644, 1992.

[82] J. H. Wilkinson. A priori error analysis of algebraic processes. In *Proceedings of International Congress of Mathematics*, pages 629–639, Moscow, 1968.

[83] J. Xia and Z. Xin. Effective and robust preconditioning of general SPD matrices via structured incomplete factorization. *Journal on Matrix Analysis and Applications*, 38(4):1298–1322, 2017.