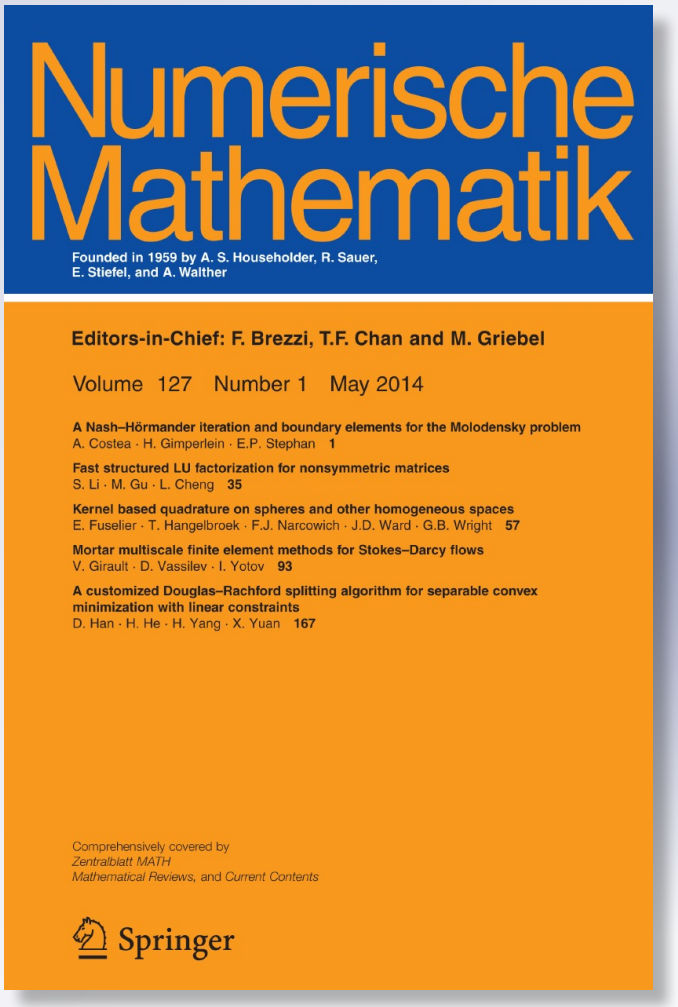
Fast structured LU factorization for nonsymmetric matrices

Shengguo Li, Ming Gu & Lizhi Cheng

Numerische Mathematik

ISSN 0029-599X
Volume 127
Number 1

Numer. Math. (2014) 127:35-55
DOI 10.1007/s00211-013-0582-0



Springer

Your article is protected by copyright and all rights are held exclusively by Springer-Verlag Berlin Heidelberg. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".



Author's personal copy

Numer. Math. (2014) 127:35–55
DOI 10.1007/s00211-013-0582-0

Numerische Mathematik

Fast structured LU factorization for nonsymmetric matrices

Shengguo Li · Ming Gu · Lizhi Cheng

Received: 10 January 2012 / Revised: 7 March 2013 / Published online: 1 October 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract In this paper, an approximate LU factorization algorithm is developed for nonsymmetric matrices based on the hierarchically semiseparable matrix techniques. It utilizes a technique involving orthogonal transformations and approximations to avoid the explicit computation of the Schur complement in each factorization step. A modified compression method is further developed for the case when some diagonal blocks have small singular values. The complexity of the methods proposed in this paper is analyzed and shown to be $O(N^2k)$, where N is the dimension of matrix and k is the maximum off-diagonal (numerical) rank. Depending on the accuracy and efficiency requirements in the approximation, this factorization can be used either as a direct solver or a preconditioner. Numerical results from applications are included to show the efficiency of our method.

This paper is dedicated to Professor Lothar Reichel on the occasion of his 60th birthday.

The work of Gu was supported in part by NSF Award CCF-0830764 and by the DOE Office of Advanced Scientific Computing Research under contact number DE-AC02-05CH11231. The work of Cheng was supported by Natural Science Project of NUDT (No. JC120201) and NSF of Hunan Province in China (No. 13JJ2001).

S. Li (✉) · L. Cheng
College of Science, National University of Defense Technology,
Changsha 410073, China
e-mail: nudtlsg@gmail.com

L. Cheng
e-mail: clzcheng@nudt.edu.cn

M. Gu
Department of Mathematics, University of California,
Berkeley, CA 47920, USA
e-mail: mgu@math.berkeley.edu

Author's personal copy

Mathematics Subject Classification 65F05 · 65F30 · 65R20 · 68P05 · 68W40

1 Introduction

Rank-structured matrices have been an area of intense research recently. We call a matrix A *rank structured* if the ranks of all its submatrices not containing any diagonal entries are small compared to the size of matrix A. Examples include $\mathcal{H}$-matrices [2,14,16], $\mathcal{H}^2$-matrices [15,17], sequentially semiseparable (SSS) matrices [5], and hierarchically semiseparable (HSS) matrices [5]. They all share the same feature that all their off-diagonal blocks are numerically low rank. More recently, they have been shown to play central roles in the superfast direct factorization and preconditioning of certain classes of large sparse matrices [29,31].

Our main objective of this work is to develop a fast and numerically stable algorithm to compute an approximate factorization of matrix A such that

$$A = S + O(\tau), \tag{1}$$

where S is a generalized LU factorization with $S = \widehat{L}\widehat{U}$, $\widehat{L}$ ($\widehat{U}$) an HSS matrix (see Sect. 3 for more details), and τ is a user-prescribed tolerance. The factorization (1) can be used as a direct solver for a tiny value of τ or a preconditioner for a large value of τ.

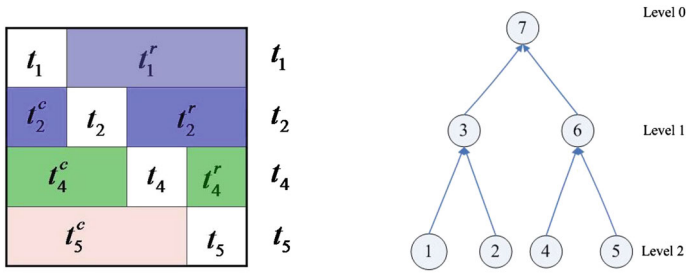
A similar algorithm has been proposed in [20] to compute an approximate Cholesky factorization of a symmetric positive definite (SPD) matrix. These two algorithms have the same complexity $O(N^2k)$, where N is the dimension of A and k is the maximum numerical rank of off-diagonal blocks. The complexity for LU factorization (1) is analyzed in Sect. 3.4.

The main idea of these two algorithms is to find two orthonormal matrices, $\mathscr{U}$ and $\mathscr{V}$, such that the LU or Cholesky factorization of $\mathscr{U}A\mathscr{V}^T$ is approximately computed without calculating the classical Schur complements. Note that $\mathscr{U}$ and $\mathscr{V}$ would be the same matrix in the SPD case. Since only a small piece of the Schur complement in each factorization step is computed, our algorithm saves many floating point operations and has better data locality than the methods which calculate the classical Schur complements (for example the method [30] in the SPD case).

Note that an LU factorization algorithm has been proposed for HSS matrices originally in [22] and its proof is given later in [25]. Comparing with that, our algorithm works directly on a (dense) matrix, not requiring the HSS construction step. More importantly, our factorizations are mainly computed through sequences of orthonormal transformations such as Householder or QR, and thus the generators for off-diagonal blocks of $\widehat{L}$ or $\widehat{U}$ are orthogonal. While, the generators computed by method in [22] can be ill-conditioned, see Sect. 4 for an example. To distinguish these two LU factorization algorithms, we call the factorization in [22] *HLU* and the factorization (1) *generalized LU factorization* (GLU) in this paper.

Recent work has suggested the efficiency and effectiveness of utilizing randomized algorithms [18,21,24] for low-rank matrix compression, which is faster than truncated SVD or rank-revealing QR (RRQR). We incorporate the randomized technique into our factorization algorithm in Sect. 3.4. To show the efficiency of our algorithm, some

Author's personal copy



(a) Matrix A **(b)** HSS tree

Fig. 1 A 4×4 HSS matrix

numerical results are included in Sect. 4, which show that our algorithm can be 2–4 times faster than *backslash* in Matlab, with even greater speedups for larger matrices or implemented in other high-level programming language such as Fortran or C. Being a preconditioner, GLU could make the iterative methods such as biconjugate gradient stabilized iterative [BICGSTAB(ℓ)] method converge very quickly even when τ is relatively big.

The structure of this paper is as follows. In Sect. 2 we introduce some basic concepts of HSS matrices and some standard low-rank matrix approximation methods including the randomized sampling method. In Sect. 3 we present the generalized HSS LU factorization algorithm in detail, and analyze its complexity. Section 4 illustrates the efficiency of our algorithm via two standard boundary integral equations and two matrices constructed from large sparse matrices. We close this paper with some conclusions in Sect. 5.

2 Preliminary

In this section, we introduce some notation and some key concepts of HSS matrices. Some standard low-rank matrix approximation methods are also included, especially the randomized sampling method that is chosen to implement our algorithm.

2.1 Notation

Let $\|\cdot\|$ denote the l_2 norm. Let $\mathcal{I} = \{1, 2, \ldots, N\}$ be the index set of columns of matrix A and $\mathcal{T}$ be a binary tree. For a nonleaf node i, denote its left child as i_1 and right child as i_2 respectively. We further assume $\mathcal{T}$ is *postordered*. That is, the ordering of nonleaf node i satisfies $i_1 < i_2 < i$. Figure 1b shows a postordering tree, where node 1 and node 2 are siblings, node 3 is a parent node.

Each node i is associated with a contiguous subset of $\mathcal{I}$, t_i, which satisfies the following conditions:

- $\cup_{i \in \text{level } \ell} t_i = \mathcal{I}$, and $t_i \cap t_j = \emptyset$, where i and j are different nodes at level ℓ;
- $t_i \cup t_{\text{sib}(i)} = t_{\text{par}(i)}$, where sib(i) and par(i) denote the sibling and parent of a node i, respectively.
- $t_{\text{root}}(\mathcal{T}) = \mathcal{I}$, root($\mathcal{T}$) denotes the root of $\mathcal{T}$.

Author's personal copy

Furthermore, for each node i, let t_i^c be the set of all indices less than those in t_i and t_i^r be the set of all indices greater than those in t_i. Thus, $\mathcal{I} = t_i^c \cup t_i \cup t_i^r$ for each node i. Just as introduced in [30], we call a block row (column) of A excluding the diagonal block an *HSS block row (column)*, or just *HSS block*. We use

$$H_i^{row} = \begin{bmatrix} A\left(t_i, t_i^c\right) & A\left(t_i, t_i^r\right) \end{bmatrix} \quad \text{and} \quad H_i^{col} = \begin{bmatrix} A\left(t_i^c, t_i\right) \\ A\left(t_i^r, t_i\right) \end{bmatrix}$$

to represent the ith HSS block row and block column respectively.

2.2 Introduction of HSS

The HSS matrix was first discussed in [7,8], based on the work of Starr [26]. It has been widely used to solve many problems such as integral equations and certain classes of large sparse matrices. It is interesting to note that HSS matrices also have connections with the more traditional structured matrices such as Toeplitz matrices and Cauchy matrices (see [6]).

The HSS is an efficient hierarchical matrix representation that systematically explores the low-rank property of off-diagonal blocks. We follow the notation in [27,30] and use postordering to order the nodes of HSS tree. Let A be a 4×4 block matrix. The partition of A is shown in Fig. 1a, b shows its corresponding HSS tree. Then, A can be written as in HSS form

$$A = \begin{bmatrix} D_1 & U_1 B_1 V_2^T & U_1 R_1 B_3 W_4^T V_4^T & U_1 R_1 B_3 W_5^T V_5^T \\ U_2 B_2 V_1^T & D_2 & U_2 R_2 B_3 W_4^T V_4^T & U_2 R_2 B_3 W_5^T V_5^T \\ U_4 R_4 B_6 W_1^T V_1^T & U_4 R_4 B_6 W_2^T V_2^T & D_4 & U_4 B_4 V_5^T \\ U_5 R_5 B_6 W_1^T V_1^T & U_5 R_5 B_6 W_2^T V_2^T & U_5 B_5 V_4^T & D_5 \end{bmatrix}. \quad (2)$$

These D_i, U_i, R_i, B_i, W_i, and V_i are called *generators*. If A is symmetric, $D_i = D_i^T, U_i = V_i, R_i = W_i$, and $B_i = B_j^T$, where i and j are siblings. The structured LU factorization proposed in this paper works directly on matrix A without constructing these generators.

To introduce our algorithm more precisely, we need one more definition, the *visited set* for a node of the HSS tree [30].

Definition 1 The visited set associated with node i of the postordered binary tree $\mathcal{T}$ is

$$\mathcal{V}_i = \{j \,|\, j \text{ is a left node and } \mathrm{sib}(j) \in \mathrm{pred}(i)\}, \quad (3)$$

where $\mathrm{pred}(i)$ is the set of predecessors associated with node i:

$$\mathrm{pred}(i) = \begin{cases} \{i\}, & \text{if } i = \mathrm{root}(\mathcal{T}), \\ \{i\} \cup \mathrm{pred}(\mathrm{par}(i)), & \text{otherwise.} \end{cases}$$

Author's personal copy

The *visited set* can be recorded by using a stack $\mathcal{S}_v$: if node i is a left node, push i onto the stack $\mathcal{S}_v$; otherwise, pop an element from the stack. Just before we dealing with node i, the elements in the stack $\mathcal{S}_v$ form $\mathcal{V}_i$.

2.3 Low rank approximation

In this subsection we introduce some low-rank matrix approximation methods. Some well-known low-rank approximation methods include RRQR factorization [3,4], SVD [12], interpolative decomposition [9,21], etc.

In our algorithm, we use randomized sampling methods to do compressions, which are very efficient and stable with quite high probability [18,24]. This kind of methods can approximately compute the SVD or QR decomposition of a large matrix quickly. In general, it has two stages:

- compute an approximate low-rank basis $Q \in \mathbb{R}^{m\times\ell}$ of the range of A such that $A \approx QQ^*A$, $\ell < \min\{m, n\}$;
- compute the desired decomposition of the matrix $B = Q^*A$.

The basic idea of randomized algorithm is to multiply A by a random matrix Ω from right (or left) to get a much smaller matrix Y, which captures most of the interested properties of A. For example, the SVD of matrix A can be computed approximately by the following steps. We assume $A \in R^{m\times n}$ and its rank is k ($\ll \min(m, n)$). Firstly, choose a random matrix $\Omega \in R^{n\times(k+p)}$ whose elements are from a normalized Gaussian distribution, and compute $Y = A\Omega$. The parameter p is called *oversampling parameter*, a modest constant around 5 or 10, [18,24]. Then, Q can be an orthonormal basis for the column space of Y, which can be computed by QR or SVD. Finally, $A \approx Q(Q^*A) = Q(U\Sigma V^T) = (QU)\Sigma V^T$.

In our algorithm, we are only interested in finding an approximate orthogonal basis of the range of A, i.e., a low-rank approximation $A \approx QT$, $T = Q^*A$ where Q has orthonormal columns. The randomized algorithm that we used has the following steps:

- generate an $n \times (k + p)$ matrix Ω whose elements are Gaussian random variables with zero mean and unit variance,
- compute the product $Y = A\Omega$,
- compute an approximate orthonormal basis Q of the range of Y.

Low rank matrix approximation problems can be divided into two kinds: *fixed-rank approximation* and *fixed-precision approximation*. Fixed-rank approximation finds a tall matrix Q and a wide matrix T, such that $A \approx QT$ and Q has exactly rank k, a prefixed number. While, fixed-precision approximation finds a low-rank approximation such that the error between A and its approximation is less than τ, i.e., satisfying $\|A - QQ^*A\| = O(\tau)$, Q has orthogonal columns. We can implement our algorithm in these two ways. Fixed-precision approximation methods are usually found to be more efficient. We describe in detail how to use the randomized algorithm to compress HSS block rows and columns in Sect. 3.4.

Author's personal copy

3 Generalized LU factorization for nonsymmetric matrices

In this section, we develop our algorithm to compute an approximate LU factorization (1). This algorithm is hierarchical in nature, which follows a postordering HSS tree. On each node of this tree, a compression scheme is used to approximate the associated off-diagonal submatrices; and a merging scheme is used from one level of the tree to the level above.

As an illustration, consider a given matrix $A \in \mathbb{R}^{N\times N}$ in 2×2 partition:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \tag{4}$$

where $A_{11} \in \mathbb{R}^{m\times m}$, $A_{22} \in \mathbb{R}^{(N-m)\times(N-m)}$, and A_{12}, A_{21} are matrices with appropriate dimensions. We further assume that matrices A_{12} and A_{21} have rapidly decaying singular values and hence can be well-approximated by low-rank submatrices.

Several approaches can be used to replace A_{12} and A_{21} by low-rank submatrices. Our goal, however, is to approximately factorize A so that the inverse of matrix S in (1) is a good approximation A^{-1}. To this end, we first assume that

$$A_{11} = X_1 Y_1^T \tag{5}$$

is a given factorization of A_{11}. Later on in Sect. 3.1, we will discuss issues related to the factorization (5). Similar to the Cholesky factorization in [20,30], we would compress the HSS block row $X_1^{-1}A_{12}$ and HSS block column $A_{21}Y_1^{-T}$. In the SPD case, an important benefit of doing so is that some positive semidefinite matrices would be implicitly added to the Schur complements, which guarantees the existence of Cholesky factorization. This compression method uses the information both from diagonal and off-diagonal submatrices, which is different from the HSS construction algorithms proposed in [8,23].

Furthermore, the compressed HSS block row and column should have the same numerical rank (which will be more clear later). To achieve this, compute an orthogonal decomposition of the form

$$\begin{bmatrix} X_1^{-1}A_{12} & Y_1^{-1}A_{21}^T \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} G_1 & Z_1 \\ G_2 & Z_2 \end{bmatrix}, \tag{6}$$

where $Q = [Q_1 \quad Q_2]$ is an orthogonal matrix, $Q_2 \in R^{m\times k_1}$. The decomposition is partitioned such that $\| [G_1 \quad Z_1] \|_2 = O(\tau)$. In other words, this single decomposition can reveal the numerical low-rank structures in both $X_1^{-1}A_{12}$ and $Y_1^{-1}A_{21}^T$.

Equation (6) is equivalent to

$$A_{12} = X_1 Q \begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \quad \text{and} \quad A_{21}^T = Y_1 Q \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix},$$

and the (1, 1) block of A can be rewritten as

$$A_{11} = (X_1 Q)\,(Y_1 Q)^T .$$

Author's personal copy

We now compute two QL factorizations of the forms,

$$X_1 Q = U^{(1)} L^{(1)}, \quad \text{and} \quad Y_1 Q = V^{(1)} \left(R^{(1)}\right)^T,$$

where $L^{(1)}$ is lower triangular and $R^{(1)}$ is upper triangular. This allows us to write

$$A_{11} = U^{(1)} L^{(1)} R^{(1)} \left(V^{(1)}\right)^T,$$

which leads to

$$A = \mathscr{U}_1 \begin{bmatrix} L^{(1)} R^{(1)} & L^{(1)} \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \\ \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix}^T R^{(1)} & A_{22} \end{bmatrix} \mathscr{V}_1^T, \tag{7}$$

where

$$\mathscr{U}_1 = \begin{bmatrix} U^{(1)} & \\ & I \end{bmatrix}, \quad \mathscr{V}_1 = \begin{bmatrix} V^{(1)} & \\ & I \end{bmatrix}, \quad I \in \mathbb{R}^{(N-m)\times(N-m)}.$$

Note that the lower (upper) triangular matrix $L^{(1)}$ ($R^{(1)}$) is computed via orthogonal transformations, and $\mathscr{U}_1$ and $\mathscr{V}_1$ are orthogonal matrices which can be ignored in the following process. To perform a Gaussian elimination step, we partition

$$L^{(1)} = \begin{bmatrix} L_{11}^{(1)} & \\ L_{21}^{(1)} & L_{22}^{(1)} \end{bmatrix}, \quad R^{(1)} = \begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} \\ & R_{22}^{(1)} \end{bmatrix}, \tag{8}$$

where $L_{22}^{(1)}$ and $R_{22}^{(1)}$ have the same row dimension as submatrix G_2. Now the middle matrix on the right hand side of Eq. (7) can be factorized as

$$\begin{aligned} \begin{bmatrix} L^{(1)} R^{(1)} & L^{(1)} \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \\ \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix}^T R^{(1)} & A_{22} \end{bmatrix} &= \begin{bmatrix} L_{11}^{(1)} & & \\ L_{21}^{(1)} & I & \\ Z_1^T & & I \end{bmatrix} \begin{bmatrix} I & & \\ & L_{22}^{(1)} R_{22}^{(1)} & L_{22}^{(1)} G_2 \\ & Z_2^T R_{22}^{(1)} & A_{22} - Z_1^T G_1 \end{bmatrix} \begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} & G_1 \\ & I & \\ & & I \end{bmatrix} \\ &\approx \begin{bmatrix} L_{11}^{(1)} & & \\ L_{21}^{(1)} & I & \\ & & I \end{bmatrix} \begin{bmatrix} I & & \\ & L_{22}^{(1)} R_{22}^{(1)} & L_{22}^{(1)} G_2 \\ & Z_2^T R_{22}^{(1)} & A_{22} \end{bmatrix} \begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} & \\ & I & \\ & & I \end{bmatrix}. \end{aligned} \tag{9}$$

In the above equation we have ignored matrices G_1, Z_1 [causing $O(\tau)$ errors in the factors], and $Z_1^T G_1$ [causing $O(\tau^2)$ error in the Schur complement]. Ignoring $Z_1^T G_1$ makes our algorithm free of computing the classical Schur complement.

Author's personal copy

The benefits are reducing floating point operations and data movements, see also [20]. In the SPD case, ignoring $Z_1^T G_1$ further makes the Cholesky factorization more stable, and this technique is called *Schur compensation* in [30] and *Schur-monotonic* in [13].

Now, we have compressed the first HSS block and only need to store

$$\widetilde{L}^{(1)} = \begin{bmatrix} L_{11}^{(1)} \\ L_{21}^{(1)} \end{bmatrix}, \quad \widetilde{R}^{(1)} = \begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} \end{bmatrix}, U^{(1)}, V^{(1)}, \text{ and } k_1, \tag{10}$$

which will be called *generators* for the generalized LU factorization. They will be used in the solvers (Algorithms 4, 5).

The process can be continued similarly by following the postordering HSS tree. We denote

$$\begin{bmatrix} I & & \\ & L_{22}^{(1)} R_{22}^{(1)} & L_{22}^{(1)} G_2 \\ & Z_2^T R_{22}^{(1)} & A_{22} \end{bmatrix} := \begin{bmatrix} I & & \\ & D_1 & A_{\hat{t}_1 t_2}^{(1)} \\ & A_{t_2 \hat{t}_1}^{(1)} & A_{22} \end{bmatrix}.$$

It should be noted that after dealing with node 1 we only need to work on the following smaller matrix

$$\begin{bmatrix} D_1 & A_{\hat{t}_1 t_2}^{(1)} \\ A_{t_2 \hat{t}_1}^{(1)} & A_{22} \end{bmatrix}. \tag{11}$$

To keep matrix (11) square, the row dimension of $A_{\hat{t}_1 t_2}^{(1)}$ should be the same as the column dimension of $A_{t_2 \hat{t}_1}^{(1)}$. That is why we compress the HSS block rows and columns simultaneously, see (6).

For node i in the general case, the process is similar. We need to determine the HSS block row H_i^{row} and HSS block column H_i^{col} which are defined as $H_i^{row} = [A_{t_i \hat{t}_{j_1}}^{(j_1)}, \ldots, A_{t_i \hat{t}_{j_{s_i}}}^{(j_{s_i})} \mid A_{t_i t_i^r}]$ and $H_i^{col} = [A_{\hat{t}_{j_1} t_i}^{(j_1)}, \ldots, A_{\hat{t}_{j_{s_i}} t_i}^{(j_{s_i})} \mid A_{t_i^r t_i}]^T$, where $j_1, j_2, \ldots, j_{s_i}$ are the elements of visited set $\mathcal{V}_i$. Figure 2 shows the HSS block row (column) of node 2. It will be easier to use a stack to find the HSS block rows and columns.

The process of compressing HSS block rows (columns) and factorizing main diagonal blocks is summarized in the following algorithm.

Algorithm 1 (*compressing off-diagonal blocks*) Suppose each off-diagonal block of A corresponding to node i has m_i rows and columns and its rank is k_i.

- Factorize the main diagonal block $A_{t_i t_i} = X_i Y_i^T$.
- Determine the HSS block row H_i^{row} and HSS block column H_i^{col}, and compress $H = [X_i^{-1} H_i^{row} \quad Y_i^{-1} (H_i^{col})^T] \approx Q_2 [G_2 \quad Z_2^T]$, where $Q = [Q_1 \quad Q_2]$ is an orthonormal matrix and $Q_2 \in \mathbb{R}^{m_i \times k_i}$.

Author's personal copy

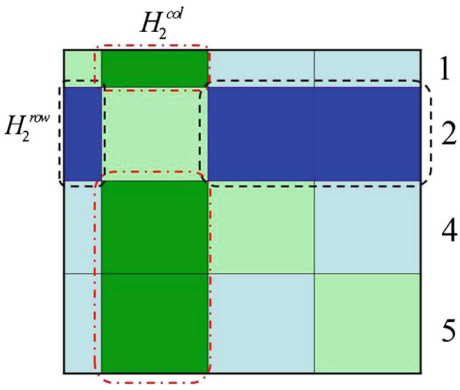


Fig. 2 The HSS block row (column) of node 2

– Compute $U^{(i)}$ and $V^{(i)}$ from $Y_i^T V^{(i)} = QR^{(i)}$ and $X_i Q = U^{(i)} L^{(i)}$, where

$$L^{(i)} = \begin{bmatrix} L_{11}^{(i)} & \\ L_{21}^{(i)} & L_{22}^{(i)} \end{bmatrix}, \quad R^{(i)} = \begin{bmatrix} R_{11}^{(i)} & R_{12}^{(i)} \\ & R_{22}^{(i)} \end{bmatrix}, \text{ and let } \widetilde{\mathrm{L}}^{(\mathrm{i})} = \begin{bmatrix} L_{11}^{(i)} \\ L_{21}^{(i)} \end{bmatrix},$$
$$\widetilde{R}^{(i)} = \begin{bmatrix} R_{11}^{(i)} & R_{12}^{(i)} \end{bmatrix}, L_{22}^{(i)}, R_{22}^{(i)} \in \mathbb{R}^{m_i \times k_i}.$$

– Compute $D_i = L_{22}^{(i)} R_{22}^{(i)}$ and the keeping nonzero off-diagonal rows $A_{\hat{t}_i,:}^{(i)} = L_{22}^{(i)} G_2$, and off-diagonal columns $A_{:,\hat{t}_i}^{(i)} = Z_2^T R_{22}^{(i)}$ of node i.

Remark 1 The small block D_i is the only Schur complement that needs to be computed during the compression step for node i. Note that the size of D_i is much smaller than the size of A.

3.1 Modified compression method

In this subsection, we discuss the computation of factorization (5). As motivated by work in [20] for SPD matrices, we choose X_1 and Y_1 to be similarly conditioned. If the matrix A_{11} itself does not contain any tiny singular values as compared with $\|A\|_2$, we can computed its factorization $A_{11} = X_1 Y_1^T$ directly by LU factorization or SVD. When A_{11} does contain some tiny singular values, we should be more careful. Otherwise, the orthogonal decomposition (6) can potentially become numerically unstable. We provide a strategy for this case.

In short, we perform compressions only after separating the well-conditioned part of A_{11} from the ill-conditioned part. We choose two (orthogonal) matrices, $\widehat{U}$ and $\widehat{V}$, to transform A_{11} into

$$\widehat{U}^T A_{11} \widehat{V} = \begin{bmatrix} \Delta_1 & * \\ * & \Delta_2 \end{bmatrix}, \tag{12}$$

where Δ_1 is the well-conditioned part and Δ_2 has some small singular values.

Author's personal copy

One way to achieve that is to let $\widehat{U}$ and $\widehat{V}$ be the left and right singular vectors of A_{11} repectively. Then, $\widehat{U}^T A_{11}\widehat{V}$ is a diagonal matrix and its diagonal entries are the singular values of A_{11}. There may be some other (more efficient) ways to choose $\widehat{U}$ and $\widehat{V}$. We multiply the first block row and column of A by $\widehat{U}$ and $\widehat{V}$ respectively, and get matrix $\begin{bmatrix} \Sigma & \widehat{U}^T A_{12} \\ A_{21}\widehat{V} & A_{22} \end{bmatrix}$, where $\Sigma = \begin{bmatrix} \Sigma_1 & \\ & \Sigma_2 \end{bmatrix}$ is a diagonal matrix with singular values of A_{11} as its diagonals, Σ_1 is an $m_1 \times m_1$ matrix and Σ_2 has small singular values.

Because of the singularity of Σ_2, we only compress the first $m_1 (< m)$ off-diagonal rows and columns by using Algorithm 1 similarly (now the main diagonal block is a diagonal matrix Σ_1), and get the following factorization,

$$\mathscr{U}_1^T \begin{bmatrix} \Sigma & \widehat{U}^T A_{12} \\ A_{21}\widehat{V} & A_{22} \end{bmatrix} \mathscr{V}_1 = \begin{bmatrix} L^{(1)}R^{(1)} & & L^{(1)}\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \\ & \Sigma_2 & \widehat{A}_{12} \\ \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix}^T R^{(1)} & \widehat{A}_{21} & A_{22} \end{bmatrix}, \tag{13}$$

where G_i, Z_i are obtained as in (6).

If we partition $L^{(1)}$ and $R^{(1)}$ as in (8), we can get a similar factorization to (9), and for node 2 we only need to consider

$$\begin{bmatrix} D_1 & A^{(1)}_{\hat{t}_1 t_2} \\ A^{(1)}_{t_2 \hat{t}_1} & A_{22} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} L^{(1)}_{22}R^{(1)}_{22} & \\ & \Sigma_2 \end{bmatrix} & \begin{bmatrix} L^{(1)}_{22}G_2 \\ \widehat{A}_{12} \end{bmatrix} \\ \begin{bmatrix} Z_2^T R^{(1)}_{22} & \widehat{A}_{21} \end{bmatrix} & A_{22} \end{bmatrix}. \tag{14}$$

To be more clear, we summarize the process in the following algorithm.

Algorithm 2 (*modified compression method*) Suppose each HSS block of A corresponding to node i has m_i rows and columns and its rank is k_i.

– Find the well-conditioned part of the main diagonal block $A_{t_i t_i}$,

$$\widehat{U}^T A_{t_i t_i}\widehat{V} = \begin{bmatrix} \Sigma_1 & \\ & \Sigma_2 \end{bmatrix}, \text{ and compute } \Sigma_1 = X_i Y_i^T \in \mathbb{R}^{d_i \times d_i};$$

– Determine the HSS block row H_i^{row} and HSS block column H_i^{col}, and compress $H = [X_i^{-1}(\widehat{U}^T H_i^{row})(1:d_i,:) \;\; Y_i^{-1}(H_i^{col}\widehat{V})(:,1:d_i))^T] \approx Q_2[G_2 \;\; Z_2^T]$, where $Q = [Q_1 \;\; Q_2]$ is an orthonormal matrix and $Q_2 \in \mathbb{R}^{d_i \times k_i}$.

– Compute $U^{(i)}$ and $V^{(i)}$ from $Y_i^T V^{(i)} = QR^{(i)}$ and $X_i Q = U^{(i)}L^{(i)}$, where

$$L^{(i)} = \begin{bmatrix} L^{(i)}_{11} & \\ L^{(i)}_{21} & L^{(i)}_{22} \end{bmatrix}, \quad R^{(i)} = \begin{bmatrix} R^{(i)}_{11} & R^{(i)}_{12} \\ & R^{(i)}_{22} \end{bmatrix}, R^{(i)}_{22} \in \mathbb{R}^{k_i \times k_i};$$

Author's personal copy

– Compute $D_i = \begin{bmatrix} L_{22}^{(i)} R_{22}^{(i)} & \\ & \Sigma_2 \end{bmatrix}$ and the keeping nonzero off-diagonal rows

$$A_{\hat{t}_i,:}^{(i)} = \begin{bmatrix} L_{22}^{(i)} G_2 \\ (\widehat{U}^T H_i^{row})(d_i + 1 : end, :) \end{bmatrix},$$

and off-diagonal columns

$$A_{:,\hat{t}_i}^{(i)} = \begin{bmatrix} Z_2^T R_{22}^{(i)} & (H_i^{col} \widehat{V})(:, d_i + 1 : end) \end{bmatrix};$$

– Store the generators,

$$\widetilde{L}^{(i)} = \begin{bmatrix} L_{11}^{(i)} \\ L_{21}^{(i)} \\ 0 \end{bmatrix}, \widetilde{R}^{(i)T} = \begin{bmatrix} R_{11}^{(i)} \\ R_{12}^{(i)} \\ 0 \end{bmatrix}, U^{(i)} = \widehat{U} \begin{bmatrix} U^{(i)} & \\ & I \end{bmatrix}, V^{(i)} = \widehat{V} \begin{bmatrix} V^{(i)} & \\ & I \end{bmatrix},$$

$k_i := m_i - d_i + k_i$, where I is an $(m_i - d_i) \times (m_i - d_i)$ identity matrix.

Remark 2 – Algorithm 2 first computes two (orthogonal) matrices to make the top-left part of the main diagonal block well-conditioned, whereas Algorithm 1 factorizes the diagonal block directly.
– In Algorithm 2, we can compute $\Sigma_1 = X_1 Y_1^T$ by $X_1 = Y_1 = \Sigma_1^{1/2}$.
– Note that we do not need to store the zeros in $\widetilde{L}^{(i)}$ and $\widetilde{R}^{(i)}$ in practice. It is written this way just to be consistent with Algorithm 1.

3.2 Merging

Like other HSS algorithms, our HSS LU factorization method also has two phrases: "compression" and "merging". For each leaf node, we need to compress its HSS block row (column) and factorize its main diagonal block. For each parent node, we need to first merge the data of its children and then perform the compression and factorization in a similar fashion.

We use node 3 as an example to show how to deal with parent nodes. When we arrive at node 3, we merge the nonzero parts of node 1 and node 2 and get the following matrix

$$\mathcal{A}_3 = \begin{bmatrix} D_1 & B_1 & A_{\hat{t}_1 t_2^r}^{(1)} \\ B_2 & D_2 & A_{\hat{t}_2 t_2^r}^{(2)} \\ A_{t_2^r \hat{t}_1}^{(1)} & A_{t_2^r \hat{t}_2}^{(2)} & A_{t_2^r t_2^r} \end{bmatrix}, \tag{15}$$

where B_1 is the part belonging to both the HSS block column of node 2 and the HSS block row of node 1, B_2 is the part belonging to both the HSS block column of node

Author's personal copy

1 and the HSS block row of node 2. Here we are assuming the original matrix A has been partitioned into $n \times n$ blocks ($n \geq 3$).

As the leaf nodes, we first determine the diagonal matrix $A_{33} = \begin{bmatrix} D_1 & B_1 \\ B_2 & D_2 \end{bmatrix}$, HSS block row $H_3^{row} = \begin{bmatrix} A^{(1)}_{\hat{t}_1 t_2^r} \\ A^{(2)}_{\hat{t}_2 t_2^r} \end{bmatrix}$ and HSS block column $H_3^{col} = \begin{bmatrix} A^{(1)}_{t_2^r \hat{t}_1} & A^{(2)}_{t_2^r \hat{t}_2} \end{bmatrix}$, then we use Algorithm 1 to compress the HSS block row (column) and get the generators for node 3. Writing it explicitly, we get the following factorization

$$\begin{bmatrix} D_1 & B_1 & A^{(1)}_{\hat{t}_1 t_2^r} \\ B_2 & D_2 & A^{(2)}_{\hat{t}_2 t_2^r} \\ A^{(1)}_{t_2^r \hat{t}_1} & A^{(2)}_{t_2^r \hat{t}_2} & A_{t_2^r t_2^r} \end{bmatrix} \approx \begin{bmatrix} L^{(3)}_{11} & & \\ L^{(3)}_{21} & I & \\ & & I \end{bmatrix} \begin{bmatrix} I & & \\ & D_3 & A^{(3)}_{\hat{t}_3 t_2^r} \\ & A^{(3)}_{t_2^r \hat{t}_3} & A_{t_2^r t_2^r} \end{bmatrix} \begin{bmatrix} R^{(3)}_{11} & R^{(3)}_{12} & \\ & I & \\ & & I \end{bmatrix}, \quad (16)$$

where $D_3 = L^{(3)}_{22} R^{(3)}_{22}$, $D_3 \in \mathbb{R}^{k_3 \times k_3}$, and the generators of node 3 are

$$\widetilde{L}^{(3)} = \begin{bmatrix} L^{(3)}_{11} \\ L^{(3)}_{21} \end{bmatrix}, \widetilde{R}^{(3)} = \begin{bmatrix} R^{(3)}_{11} & R^{(3)}_{12} \end{bmatrix}, U^{(3)}, \ V^{(3)}, \text{ and } k_3. \quad (17)$$

We continue this process until we arrive at the *root* node, where we only need to compute an LU factorization. We summarize the whole process in the following algorithm. Assume there are N_t nodes totally, and node N_t is the root node.

Algorithm 3 (*generalized LU factorization for HSS matrices*) Inputs:

1. A general matrix A which is to be factorized;
2. A binary tree $\mathcal{T}$, corresponding to A.
3. The block size of each leaf node $\{m_i\}$.
4. Approximation parameter: r: the preset rank or τ: the precision parameter.

Outputs:

1. $U^{(i)}$: a orthonormal matrices for each node;
2. $V^{(i)}$: a orthonormal matrices for each node;
3. $\widetilde{L}^{(i)}$: a low-triangular matrix for each node;
4. $\widetilde{R}^{(i)}$: an up-triangular matrix for each node;
5. k_i: the number of keeping nonzero off-diagonal rows (columns) for each node;

– For $i = 1, \ldots, N_t - 1$
  1. if node i is a leaf node
     – use Algorithm 1 to get the generators of node i;
  2. if node i is a non-leaf node,
     – Determine the main diagonal block A_i from its children i_1, i_2,
     $A_{ii} = \begin{bmatrix} D_{i_1} & B_{i_1} \\ B_{i_2} & D_{i_2} \end{bmatrix}$;

Author's personal copy

  - Determine the HSS block row H_i^{row} and column H_i^{col} from its children;
  - Just like leaf nodes, use Algorithm 1 to compute $U^{(i)}$, $V^{(i)}$, $\widetilde{L}^{(i)}$ and $\widetilde{R}^{(i)}$, and the keeping nonzero parts.
- End for
- For the root N_t, determine the main diagonal block $A_{N_t} = \begin{bmatrix} D_{N_{t1}} & B_{N_{t1}} \\ B_{N_{t2}} & D_{N_{t2}} \end{bmatrix}$ from its children, then compute $L_{N_t} R_{N_t} = lu(A_{N_t})$.

3.3 HSS LU solver

This solver is similar to those proposed in [20,28], which we include for completeness. This solver includes two steps: "forward substitution" and "backward substitution". Assume each node i has generators $U^{(i)}$, $V^{(i)}$, $\widetilde{L}^{(i)}$, $\widetilde{R}^{(i)}$, and k_i, where k_i is the number of keeping nonzero rows. We follow the HSS postordering tree to implement the forward and backward substitution.

Algorithm 4 (*forward substitution*) Partition b according to the leaf nodes and associate each leaf node i with b_i, and assume there are N_t nodes.

- For node $i = 1, \ldots, N_t - 1$,
  1. if node i is a leaf node, compute

$$\hat{b}_i = U^{(i)T} \cdot b_i = \begin{bmatrix} \hat{b}_{i;1} \\ \hat{b}_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix}, \text{ then } b_i = \begin{bmatrix} L_{11}^{(i)} & \\ L_{21}^{(i)} & I \end{bmatrix}^{-1} \cdot \hat{b}_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix};$$

  2. if node i is a nonleaf node, b_i is calculated by collecting the second parts of its children, $b_i = \begin{bmatrix} b_{i_1;2} \\ b_{i_2;2} \end{bmatrix}$, where i_1, i_2 are the left and right child of node i, respectively. Compute

$$b_i = \begin{bmatrix} L_{11}^{(i)} & \\ L_{21}^{(i)} & I \end{bmatrix}^{-1} \cdot U^{(i)T} \cdot b_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix}.$$

- End for
- For root node N_t, $b_{N_t} = \begin{bmatrix} b_{N_{t1};2} \\ b_{N_{t2};2} \end{bmatrix}$, where N_{t1}, N_{t2} are the left and right child of node N_t respectively, compute $b_{N_t} = L_{N_t}^{-1} \cdot b_{N_t}$.

The backward substitution is similar to the conventional one except that we need to multiply some orthogonal matrices $V^{(i)}$ in our algorithm.

Algorithm 5 (*backward substitution*) For root N_t, compute $b_{N_t} = R_{N_t}^{-1} \cdot b_{N_t}$ and partition it according to its children N_{t1} and N_{t2}, i.e., $b_{N_t} = \begin{bmatrix} b_{N_t;1} \\ b_{N_t;2} \end{bmatrix} \begin{matrix} k_{N_{t1}} \\ k_{N_{t2}} \end{matrix}$, and update the second part values of its children, $b_{N_{t1};2} = b_{N_t;1}$ and $b_{N_{t2};2} = b_{N_t;2}$.

Author's personal copy

– For node $i = N_t - 1, \ldots, 1$,
  1. if node i is a non-leaf node, compute

$$b_{i;1} = (R_{11}^{(i)})^{-1} \cdot (b_{i;1} - R_{12}^{(i)} \cdot b_{i;2}), \quad b_i = V^{(i)} \cdot b_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix}$$

     then update the values of its children: $b_{i_1;2} = b_{i;1}, \quad b_{i_2;2} = b_{i;2}$;
  2. if node i is a leaf node, compute

$$b_{i;1} = (R_{11}^{(i)})^{-1} \cdot (b_{i;1} - R_{12}^{(i)} \cdot b_{i;2}), \quad \text{then get the solution at node } i, b_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix}.$$

– End for
– Collect all the solutions $\{b_{i_j}\}$ of the leaf nodes, the final solution will be $x = [b_{i_1}^T, \ldots, b_{i_s}^T]^T$, if there are s leaf nodes.

3.4 Low rank approximation and complexity

Compression of HSS block rows and columns is a standard low rank matrix approximation problem. In our implementation, we used the randomized SVD method to do these compressions. Recall that we want to compress the HSS block row and column simultaneously, i.e., compress $H = [X_i^{-1} H_i^{row} \quad Y_i^{-1} (H_i^{col})^T]$.

In this subsection, we will analyze the complexity of the whole algorithm. Before we do that, we need a theorem from [27], which makes our analysis much easier and more precise.

Theorem 1 ([27]) Let $\mathcal{T}$ be a perfect binary tree. Assume its root is at level 0 and its leaves are at the largest (bottom) level. Let s_i be the cardinality of $\mathcal{V}_i$ in (3). Then at level l, s_i satisfies $1 \le s_i \le l$ for any node i, and the number of nodes i with the same value $s_i \equiv j$ $(1 \le j \le l)$ is $f_j^l = \binom{l}{j}$.

The algorithm can be divided into two parts: *leaf* nodes and *parent* nodes. In the following complexity analysis, we assume A is an $N \times N$ matrix; the HSS tree has L levels, where $l = 0$ is the root level; and all the HSS blocks have the same rank k. Moreover, we assume there are $2n - 1$ nodes on this tree where n is the number of leaves, $n = 2^{L-1}$, and each leaf node has m rows. Then, $N = mn$. Denote $LN = \{i |$ node i is a leaf node$\}$. Let N_i be the number of columns in $A_{t_i t_i^r}$.

For each leaf node i, we have the following steps.

– Factorize the main diagonal block $A_{ii} = U \Sigma V^T = X_i Y_i^T$, which needs $O(m^3)$ flops. Let $X_i = U \Sigma^{\frac{1}{2}}$ and $Y_i = V \Sigma^{\frac{1}{2}}$ (There may be some other factorizations).
– Compute $H = [\Sigma^{-\frac{1}{2}} U^T H_i^{row} \quad \Sigma^{-\frac{1}{2}} V^T (H_i^{col})^T]$, which needs $O(2m^2 \times (ks_i + N_i)) + O(2m \times (ks_i + N_i))$ flops, where H_i^{row} $((H_i^{col})^T)$ is a $m \times (ks_i + N_i)$ matrix, and s_i is the cardinality of $\mathcal{V}_i$.
– Compress H and compute Q by the following steps:

Author's personal copy

  - multiply H by a random matrix Ω from right which has $k+p$ columns, $Y = H\Omega \in \mathbb{R}^{m\times(k+p)}$, which needs $O(2m(k+p)(ks_i + N_i))$ flops.
  - compute $Y \approx QQ^*Y$ using SVD or (RR)QR, needs $O(4m^2(k+p))$ flops;
- Compute $U^{(i)}$ and $V^{(i)}$ from $Y_i^T V^{(i)} = QR^{(i)}$ and $X_i Q = U^{(i)} L^{(i)}$, which needs $O(m^3)$ flops;
- Compute $D_i = L_{22}^{(i)} R_{22}^{(i)}$, the keeping nonzero off-diagonal rows $L_{22}^{(i)} G_2$ and off-diagonal columns $A_{:,\hat{t}_i}^{(i)} = Z_2^T R_{22}^{(i)}$, which needs $\frac{2k^3}{3} + 2N_i k^2 + 2k^3 s_i$ flops.

Therefore, the total cost of all the leaf nodes is approximately

$$\begin{aligned} C_f &= \sum_{i\in LN} \{2(m^2 + m(k+p) + k^2)N_i + 2(m^2 k + mk(k+p))s_i \\ &\quad + m^3 + 4m^2(k+p) + \frac{2k^3}{3}\} \\ &\approx 8m^2 \sum_{i\in LN} N_i + 8k^3 \sum_{i\in LN} s_i + 10m^3 \sum_{i\in LN} 1 \\ &= 4m^2 n(n-1)m + 4k^3 nL + 10m^3 \frac{N}{m} \\ &\approx O(N^2 k) + O(Nk^2), \end{aligned}$$

where we have used $N = mn$ and $m = O(k)$ and the results of Theorem 3.3 [27] or Theorem 1, i.e.,

$$\sum_{i\in \text{levl}\, l} s_i = \sum_{j=1}^{l} j \binom{l}{j} = \frac{1}{2} l 2^l, \tag{18}$$

$$\sum_{i\in \text{level}\, l} N_i = \sum_{j=1}^{2^l} \left(n - j\frac{n}{2^l}\right) m = \frac{1}{2} mn(2^l - 1). \tag{19}$$

For the complexity of parent nodes, the analysis is similar. There are totally $n-1$ parent nodes and the main diagonal block of each parent node is a $2k \times 2k$ matrix, i.e, $m = 2k$. At level l, there are 2^l parent nodes. Adding up all the operations level by level,

$$\begin{aligned} C_p &\approx \sum_{l=1}^{L-2} \sum_{i\in \text{level}\, 1} \{18k^2 N_i + 16k^3 s_i + 41k^3\} \\ &= 18k^2 \sum_{l=1}^{L-2} \frac{1}{2} N(2^l - 1) + 8k^3 \sum_{l=1}^{L-2} l 2^l + 41k^3(n-1) \\ &= 9k^2 N 2^{L-1} + 8k^3(n(L-3)+2) + 41k^3(n-1) \\ &\approx O(N^2 k), \end{aligned} \tag{20}$$

Author's personal copy

where we have used $n = 2^{L-1}$, $N = mn$ and $m = O(k)$. The complexity for the root node is $C_r = \frac{2(2k)^3}{3} < N^2 k$. So the total complexity $C = C_f + C_p + C_r = O(N^2 k)$.

4 Numerical examples

In this section we illustrate some applications of our algorithm. All the numerical experiments are implemented in Matlab, and performed on a laptop with 4G memory and Intel(R) Core(TM) i7-2640M CPU.

Example 1 We use double layer and single layer potential method to solve an interior Dirichlet problem for Laplace equation on a smooth planar boundary. We compute an HSS LU factorization of the matrix getting from Nyström discretization of one of the following integral equations:

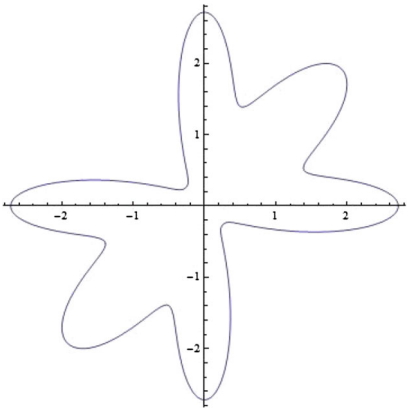
$$\int_{\Gamma} \log|P - Q|\phi(Q)ds_Q = f(P), \quad P \in \Gamma, \tag{21}$$

$$-\frac{1}{2}\phi(P) + \frac{1}{2\pi}\int_{\Gamma} \frac{\partial}{\partial n_Q}\log|P - Q|\phi(Q)ds_Q = f(P), \tag{22}$$

where n_Q is the outward pointing unit normal of Γ, illustrated in Fig. 3 and Ω is the domain enclosed by Γ, and $P = (x, y)$.

The kernel in (22) is smooth and we use the trapezoidal rule to discretize this integral equation. The equation (21) has a log-singular kernel and we use a quadrature method proposed in [10] to solve this problem. This quadrature rule is easy to implement, but has lower order of accuracy compared with the methods proposed, for example, in [1,19]. In the implementation, we assume the exact solution is $u(P) = e^x \cos(y) \in \Omega \cup \Gamma$ and we could choose some random points in Ω and compare the numerical solution to the exact one. We first use our algorithm as a direct solver and compare it with the Gaussian elimination (GE) method ('backslash' in Matlab). Then we use

Fig. 3 A hexagram boundary



Author's personal copy

Table 1 *Example 1* Comparison with Gaussian elimination when our algorithm is used as a direct solver

Problems		512	1,024	2,048	4,096	8,192
Single layer	Gaussian	7.62e−03	5.10e−02	3.02e−01	2.10e+00	1.42e+01
	GLU	6.97e−02	1.59e−01	4.20e−01	1.57e+00	6.50e+00
Double layer	Gaussian	7.23e−03	5.85e−02	4.09e−01	3.30e−00	2.12e+01
	GLU	7.70e−02	1.45e−01	3.90e−01	1.29e−00	4.82e+00

The execution times are in second

Table 2 *Example 1* Comparison with other methods when our algorithm is used as a preconditioner

n	Gauss	w. hss	w.o. pre.	w. Jacobi	Flops ratio
5,12	7.20e−03	7.35e−02(1.5)	1.44e−02(9.25)	2.85e−02(9.25)	5.32e−01
1,024	7.94e−02	1.73e−01(1.25)	5.30e−02(9.25)	1.07e−01(9.5)	1.79e−01
2,048	4.32e−01	4.16e−01(1.5)	1.66e−01(9.25)	3.57e−01(9.5)	6.62e−02
4,096	2.96e−00	1.39e−00(1.5)	6.41e−01(9.5)	1.34e−00(9.5)	2.59e−02
8,192	2.10e+01	4.67e−00(1.25)	2.45e−00(9.5)	5.19e−00(9.5)	1.09e−02

The execution times are in second, and the numbers in the parentheses are the iteration numbers of BICGSTAB(ℓ)

our factorization as a preconditioner of BICGSTAB(ℓ) to solve these problems. In this example we let the bottom block size $m = 60$ and oversampling parameter $p = 5$.

Table 1 illustrates the results when using our algorithm as a direct solver to solve (21) and (22). The results of our algorithm are obtained by letting $\tau = 1.0\text{e}{-06}$ during the low-rank approximations. When the matrix becomes larger and memory becomes a issue, our algorithm can be more than four times faster than Gaussian elimination. It should be noted that our algorithm would be much faster if implemented in C or Fortran. This is supported by results in the last column of Table 2, which show the ratio of flops needed by our algorithm over that needed by GE ($2N^3/3$). In the sense of flops, our algorithm can be about 100 times faster than GE for large matrices.

We also compare our algorithm as a preconditioner of BICGSTAB(ℓ) against the Gaussian elimination method and BICGSTAB(ℓ) with/without Jacobi preconditioner. Being a preconditioner, we let $\tau = 1.0\text{e}{-03}$ and solve the linear equations resulting from discretizing equation (22). We illustrate the execution times (in second) in Table 2, and use the following notation:

Gauss:	the 'backslach' in Matlab,
w. hss:	BICGSTAB(ℓ) with our factorization as a preconditioner,
w.o. pre:	BICGSTAB(ℓ) without preconditioner,
w. Jacobi:	BICGSTAB(ℓ) with Jacobi preconditioner,
flops ratio:	the number of flops of our algorithm divided by $2N^3/3$, where N is the dimension of matrix

For BICGSTAB(ℓ), we let the relative residual tolerance be $1.0\text{e}{-13}$. The numbers in the parentheses of Table 2 are the iteration steps of BICGSTAB(ℓ). Table 2 shows

Author's personal copy

Table 3 *Example 2* Solve $Hx = b$, where b is a random vector

	Transient			Rajat29		
	w. hss	w.w. pre.	w. Jacobi	w. hss	w.o. pre.	w. Jacobi
Time	5.69e−01	2.03e+01	4.38e+01	1.65e−00	8.03e+01	1.77e+02
Iteration	1.25	–	1430.25	4	–	–
Rel. res.	8.25e−10	1.97e−02	6.05e−13	3.32e−10	1.17e−02	6.71e−02

time denotes the execution times in second, *iteration* denotes the iteration steps of BICGSTAB(ℓ), and *rel. res.* denotes the relative residual of the computed x

that our algorithm can be four times faster than the Gaussian elimination method when the matrix is large and that BICGSTAB(ℓ) needs much fewer iterations when using our factorization as a preconditioner.

Example 2 Another and more important application of our algorithm is to develop superfast solvers for sparse matrices, and this is also the main driving force of computing the factorization (1). Superfast algorithms with structured factorization can be found, for example, in [29] where a superfast method is proposed for large sparse SPD matrices. Our algorithm is also suitable for nonsymmetric or symmetric not definite sparse matrices. We take the matrices 'transient' and 'rajat29' from sparse matrix collections [11] to show this point. We use the following Matlab commands to get a dense matrix

- P = colamd(A);
- Q = colamd(A');
- [L, U, PP] = lu(A(Q, P), 'vector');
- $\mathrm{H} = \mathrm{L}(\mathrm{end} - \mathrm{N} : \mathrm{end},\ \mathrm{end} - \mathrm{N} : \mathrm{end}) * \mathrm{U}(\mathrm{end} - \mathrm{N} : \mathrm{end}, \mathrm{end} - \mathrm{N} : \mathrm{end})$,

where N is the dimension of the dense block at the right-bottom part of L and U. Matrix H has off-diagonal low-rank property. For 'transient' matrix we let $N = 1,948$ and 'rajat29' matrix $N = 3,947$. Both these two dense matrices are nonsymmetric and ill-conditioned with condition numbers 3.04e+07 and 8.10e+06 respectively.

In this experiment, we use different methods to solve linear equations $Hx = b$, where b is a random vector. The factorization (1) would be used as a preconditioner. For BICGSTAB(ℓ), we let the relative residual tolerance be 1.0e−09 and the maximum number of iterations be 1,500. The results are shown in Table 3.

For these two matrices in Table 3, BICGSTAB(ℓ) does not converge without preconditioner, and it converges with the Jacobi preconditioner for the first matrix, while not for the second one. Table 3 shows that BICGSTAB(ℓ) preconditioned by GLU is the fastest among these three methods. For the second dense matrix obtained from 'rajat29' matrix, Gaussian elimination needs 2.66 s and the relative residual is 1.23e−12. It further supports our conclusion that for large matrices our algorithm can be faster than Gaussian elimination.

Example 3 In this example, we show that the generators computed by HLU proposed in [22] can be ill-conditioned, and thus HLU factorization can potentially loss accuracy.

Author's personal copy

Table 4 *Example 3* The results after preconditioned by HLU and GLU

	HLU			GLU		
τ	1.0e−01	1.0e−2	1.0e−03	1.0e−01	1.0e−02	1.0e−03
Pre. cond	5.86e+03	5.60e+00	5.60e+00	5.41e+00	2.75e+00	2.37e+00
Iteration	3.5	2.5	2.5	2.0	1.5	1.25
Generators	7.30e+26	7.79e+26	7.79e+26	5.12e+05	6.09e+05	7.78e+05

We use the dense matrix constructed from 'transient' matrix to show this point, and the results are included in Table 4, where

τ denotes the accuracy parameter for the compressions of HLU and GLU algorithm,
pre. cond denotes the condition number of the preconditioned matrix,
iteration denotes the iteration steps of BICGSTAB(ℓ),
generators denotes the largest condition number of all generators.

We recall the process of HLU factorization briefly. It first constructs an HSS approximation $\widetilde{H}$, with generators U, V, R, W and B, to the dense matrix H obtained from 'transient' matrix. Then the generators of factors $\widetilde{L}$ and $\widetilde{R}$ ($\widetilde{H} = \widetilde{L}\widetilde{R}$), $\hat{D}$, $\hat{U}$, $\hat{V}$, $\hat{R}$, $\hat{W}$ and $\hat{B}$, can be computed by traversing the HSS tree in two directions, upwards and downwards. The computation involves LU factorizations and solving linear equations, see [22,25] for details.

Some intermediate matrices during the computation of $\widetilde{L}$ and $\widetilde{R}$ may be ill-conditioned. Therefore, some generators of $\widetilde{L}$ and $\widetilde{R}$ may have very large condition numbers. The largest condition numbers of generators of $\widetilde{L}$, $\widetilde{R}$ and $\widehat{L}$, $\widehat{U}$ are shown in the last row of Table 4. From it, we can see that generators for GLU have much smaller condition numbers than generators for HLU. This is because GLU is mainly computed through sequences of orthogonal transformations. The results in the third row of Table 4 also show that being a preconditioner GLU is usually better than HLU.

Note that the ULV decomposition [8] can also be used to solve an HSS linear system. ULV is computed through orthogonal factorizations, and is more numerically stable than HLU. Our algorithm (GLU) usually competes with the ULV solver. For the matrices shown in Table 3, GLU would be a better preconditioner than ULV. While, for some matrices whose diagonal blocks are ill conditioned, ULV can be better than GLU since the computations of $X_i^{-1} H_i^{row}$ and $Y_i^{-1} (H_i^{col})^T$ in GLU may introduce many rounding errors.

5 Conclusions

We propose a structured LU factorization algorithm for nonsymmetric matrices, which generalizes the Cholesky factorization algorithm in [20,28]. These factorizations are both based on the HSS structure. In this paper, we further propose a strategy for the case when some diagonal blocks are ill-conditioned. In summary, there are two important features of our factorization.

Author's personal copy

– Our HSS factorization algorithm computes low-rank approximations based on information from both diagonal and off-diagonal submatrices of the given matrix.
– This factorization is designed to avoid the explicit computation of Schur complements throughout the whole computation process.

The first feature allows us to construct very effective preconditioners, whereas the second significantly reduces the amount of required memory references.

Future work includes incorporating our factorization into superfast sparse direct solvers and preconditioners for general large and sparse matrices.

Acknowledgments The authors are very grateful to the referees for their valuable suggestions.

References

1. Atkinson, K., Shampine, L.: Algorithm 876: Solving Fredholm integral equations of the second kind in MATLAB. ACM Trans. Math. Softw. **34**(4), 21:1–21:20 (2008)
2. Bebendorf, M.: Hierarchical Matrices. Lecture Note in Computational Science and Engineering. Spring-Verlag, Berlin (2008)
3. Chan, T.F.: Rank revealing QR factorizations. Linear Algebr. Appl. **88/89**, 67–82 (1987)
4. Chan, T.F., Hansen, P.C.: Some applications of the rank-revealing QR factorization. SIAM J. Sci. Stat. Comput. **13**(3), 727–741 (1992)
5. Chandrasekaran, S., Dewilde, P., Gu, M., Pals, T., Sun, X., van der Veen, A.J., White, D.: Some fast algorithms for sequentially semiseparable representation. SIAM J. Matrix Anal. Appl. **27**, 341–364 (2005)
6. Chandrasekaran, S., Gu, M., Sun, X., Xia, J.: A superfast algorithm for Toeplitz systems of linear equations. SIAM J. Matrix Anal. Appl. **29**, 1247–1266 (2007)
7. Chandrasekaran, S., Gu, M., Pals, T.: Fast and stable algorithms for hierarchically semi-separable representations. Department of Mathematics, University of California, Berkeley, Technical report(2004)
8. Chandrasekaran, S., Gu, M., Pals, T.: A fast ULV decomposition solver for hierarchically semiseparable representations. SIAM J. Matrix Anal. Appl. **28**, 603–622 (2006)
9. Cheng, H., Gimbutas, Z., Martinsson, P.G., Rokhlin, V.: On the compression of low rank matrices. SIMA J. Sci. Comput. **26**, 1389–1404 (2005)
10. Christiansen, S.: Numerical solution of an integral equation with a logarithmic kernel. BIT **11**, 276–287 (1971)
11. Davis, T., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
12. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edn. The Johns Hopkins University Press, Baltimore (1989)
13. Gu, M., Li, X.S., Vassilevski, P.S.: Direction-preserving and Schur-monotonic semiseparable approximations of symmetric positive definite matrices. SIAM J. Matrix Anal. Appl. **31**, 2650–2664 (2010)
14. Hackbusch, W.: A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$. Computing **62**, 89–108 (1999)
15. Hackbusch, W., Börm, S.: Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices. Computing **69**, 1–35 (2002)
16. Hackbusch, W., Khoromskij, B.: A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part II: application to multi-dimensional problems. Computing **64**, 21–47 (2000)
17. Hackbusch, W., Khoromskij, B., Sauter, S.: On $\mathcal{H}^2$-matrices. In: Bungartz, H., Hoppe, R.H.W., Zenger, C. (eds.) Lecture on Applied Mathematics, pp. 9–29. Springer, Berlin (2000)
18. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev. **53**(2), 217–288 (2011)
19. Kapur, S., Rokhlin, V.: High-order corrected trapezoidal quadrature rules for singular functions. SIAM J. Numer. Anal. **34**(4), 1331–1356 (1997)
20. Li, S.G., Gu, M., Wu, C.J., Xia, J.: New efficient and robust HSS Cholesky factorization of symmetric positive definite matrices. SIAM J. Matrix Anal. Appl. **33**, 886–904 (2011)

Author's personal copy

21. Liberty, E., Woolfe, F., Martinsson, P.G., Rokhlin, V., Tygert, M.: Randomized algorithms for the low-rank approximation of matrices. PNAS **104**(51), 20167–20172 (2007)
22. Lyons, W.: Fast Algorithms with Applications to PDEs. PhD Thesis, University of California, Santa Barbara (2005)
23. Martinsson, P.G.: A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. SIAM. J. Matrix Anal. Appl. **32**, 1251–1274 (2011)
24. Martinsson, P.G., Rokhlin, V., Tygert, M.: A randomized algorithm for the approximation of matrices. Appl. Comput. Harmon. Anal. **30**, 47–68 (2011)
25. Sheng, Z., Dewilde, P., Chandrasekaran, S.: Algorithms to solve hierarchically semi-separable systems. Oper. Theory Adv. Appl. **176**, 255–294 (2007)
26. Starr, P.: On the numerical solution of one-dimensional integral and differential equations. Thesis advisor: Rokhlin, V., Research Report YALEU/DCS/RR-888. Department of Computer Science, Yale University, New Haven (1991)
27. Xia, J.: On the complexity of some hierarchical structured matrices. SIAM J. Matrix Anal. Appl. **33**, 388–410 (2011)
28. Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Fast algorithm for hierarchically semiseparable matrices. Numer. Linear Algebr. Appl. **17**, 953–976 (2010)
29. Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Superfast multifrontal method for large structured linear systems of equations. SIAM J. Matrix. Anal. Appl. **31**, 1382–1411 (2009)
30. Xia, J., Gu, M.: Robust approximate Choleksy factorization of rank-structured symmetric positive definite matrices. SIAM J. Matrix Anal. Appl. **31**, 2899–2920 (2010)
31. Xia, J.: Efficient structured multifrontal factorization for general sparse matrices. SIAM J. Sci. Comput. **35**(2), A832–A860 (2013)