RC25499 (WAT1410-069) October 16, 2014
Mathematics

IBM Research Report

Computing Square Root Factorization for Recursively Low-Rank Compressed Matrices

Jie Chen
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

COMPUTING SQUARE ROOT FACTORIZATION FOR RECURSIVELY LOW-RANK COMPRESSED MATRICES

JIE CHEN*

Abstract. We present an algorithm for computing a factorization $A = GG^*$ for an $n \times n$ Hermitian positive definite matrix A, where both A and G have the same recursively low-rank compressed structure. This factorization is a Cholesky factorization in a general sense, because the factor G is not (block) triangular. Both time and storage costs are $O(n)$. The factorization can be used for sampling a multivariate normal distribution or a Gaussian process, where A is the (compressed) covariance matrix. In this case, a Gaussian sample is the mean vector plus the matrix-vector product of G with a random vector from the standard Gaussian. The matrix-vector product can be formed by using an $O(n)$ algorithm discussed in [6].

Key words. Compressed matrix, matrix square root, Cholesky factorization, Gaussian sampling

AMS subject classifications. 65F30, 65C60

1. Introduction. We consider factorizing an $n \times n$ Hermitian positive definite matrix A in the form

$$A = GG^*. \tag{1.1}$$

Such a factorization has several uses. Apart from solving a linear system of equations $Ax = b$, where the factor G is in some special form (e.g., triangular) such that the action of G^{-1} on a vector can be efficiently computed, an important application concerned in this paper is Gaussian sampling. It is well known that a sample of a multivariate Gaussian distribution or a Gaussian process with a mean vector μ and a covariance matrix A can be computed as

$$Gy + \mu, \tag{1.2}$$

where the entries of the random vector y are independent standard Gaussian (mean 0 and variance 1).

Two methods for computing (1.1) are extensively used. One is the dense Cholesky factorization, where G is lower triangular [11]. This method is limited by the $O(n^2)$ storage and $O(n^3)$ time cost. A size n on the order 10^7 will quickly hit the memory barrier on nowadays supercomputers. Another method is a sparse Cholesky factorization with some use of reordering [9]. This method is $O(n)$, loosely speaking, but it applies to only a sparse matrix. Its efficiency heavily depends on the sparsity pattern and the reordering scheme that controls fill in.

In this paper, we consider a class of dense matrices that can be compressedly stored by using only $O(n)$ memory. One example often seen in practice is a kernel matrix generated by using a positive definite kernel function (e.g., the covariance matrix of a Gaussian process, generated from a covariance function). Such a matrix can be compressed by using several structures; we focus on the *recursively low-rank compressed* structure studied in [6]. We propose a method for computing (1.1) in $O(n)$ time, where the resulting factor G has the same compressed structure as A. Hence, a sample of the Gaussian process in the form (1.2) can be computed in $O(n)$ time by using the matrix-vector multiplication algorithm considered in [6]. The time

*IBM T. J. Watson Research Center, Yorktown Heights, NY 10598. Email: chenjie@us.ibm.com

for factorization in general dominates that for computing one matrix-vector product. Hence, the proposed method is particularly favorable when a large number of samples are needed, because the time for factorization is amortized in the repeating matrix-vector multiplications.

Kernel matrices can be compressed in several structures. For some of them, the factorization (1.1) has been studied: the HSS structure is considered in [17, 14] and the HODLR structure is considered in [1]. The intricate relationship between different structures is discussed in [6]. In particular, our recursively low-rank compressed structure is almost equivalent to the HSS structure, except that we consider a slightly more general compression tree. Both structures have an $O(n)$ storage cost. However, the factorization methods proposed in [17, 14] have an $O(n^2)$ time cost, whereas that for ours is $O(n)$. The HODLR structure differs from ours in that the bases are not nested; hence, the storage and time costs are $O(n \log^\tau n)$, where τ are small integers.

For any compressed structure, one typical challenge is the loss of positive definiteness. This issue is known when compressing the matrix [3] and when computing the factorization [17]. Hence, some forms of compensation are used therein to ensure robustness. Here, we consider the two problems, compression and factorization, separately. For compression, we assume that the matrix A has already been compressed and is positive definite. Often, covariance matrices generated from kernels have tiny eigenvalues when the matrix is large. These eigenvalues may appear negative even when they are computed by using the most robust eigen solver, and they may also cause a breakdown for the Cholesky factorization. Thus, adding an appropriate positive constant to the diagonal becomes imperative. Whereas such a "regularization" approach may seem artificial, the constant is meaningful in the context of Gaussian processes—it models the variance of measurement error [15, 16]. The constant compensates the loss of positive definiteness caused by compression.

Once we have constructed a positive definite A (in a numerical sense), our task is to compute the factorization (1.1) in a stable manner. The method proposed here does not need any form of compensation. We demonstrate in the experiments that the factorization error is comparable with that of the standard Cholesky factorization. Traditionally, the error in the linear system solution is another metric for quantifying the accuracy of a Cholesky factorization. In our case, however, the factorization is not used for solving linear systems; thus, we do not consider such an error metric here. One may refer to the $O(n)$ matrix inversion algorithm proposed in [6] for solving linear systems.

For the application of Gaussian sampling, other numerical methods exist. A class of methods treats the problem as computing $A^{1/2}y + \mu$. Several iterative methods for computing $A^{1/2}y$ take a "matrix-free" form [12, 7, 8]. That is, the matrix A may not be explicitly stored, as long as matrix-vector products with A can be efficiently computed. The methods in [12, 7, 8] all approximate $A^{1/2}y$ by $p(A)y$, where p is some function that converges to the square root. In [12], p is a rational polynomial resulting from a quadrature evaluation of the contour integral that expresses the square root on the complex plane. In [7], p is the least squares polynomial given a specially designed L^2 inner product. In [8], p is the Krylov polynomial that interpolates the Ritz values; these values converge to the eigenvalues of A in a Lanczos procedure. Each method has several pros and cons and we do not expand the discussion of these methods in the current paper.

We will first review the recursively low-rank compressed structure [6] in Section 2. To be general, we lay all the discussions in the context of complex matrices,

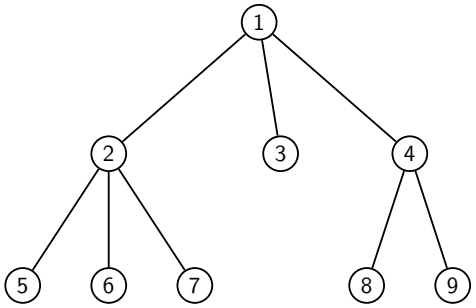
even though kernel matrices are often real. Then, we sketch the algorithm for computing (1.1) in Section 3. Similar to the algorithms consdiered in [6], the algorithm proposed here exploits the recurrence relation between two consecutive levels of the tree. Some technical complication exists, however, in the computation that satisfies the recurrence relation. Hence, we devote separately Sections 4 and 5 to discuss further details. Then, the full algorithm is presented in Section 6, with cost analysis in Section 7. We briefly discuss the generation of a positive definite matrix in Section 8. In Section 9, we show comprehensive numerical results, including the application of Gaussian sampling, to demonstrate the usefulness of the proposed algorithm. We conclude in Section 10.

2. Recursively low-rank compressed matrix. The compression structure begins with a *partitioning tree*.

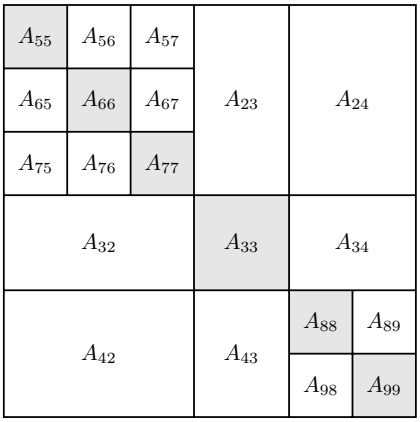
DEFINITION 2.1. *A rooted tree T is called a* partitioning tree *of a set I of indices if*

1. *no nodes have exactly one child;*
2. *the root contains I;*
3. *the children of a nonleaf node i constitutes a partitioning of the set of indices i contains.*

Figure 2.1(a) shows an example. A partitioning tree is almost as general as an arbitrary rooted tree, except for the first requirement that every nonleaf node must have more than one child. Based on this structure, a *recursively low-rank compressed matrix* is defined.



(a) Partitioning tree T



(b) Recursively low-rank compressed matrix A

FIG. 2.1. *A tree and the matrix it represents.*

DEFINITION 2.2. *For every partitioning tree T of a set of indices $\{1, \ldots, n\}$ and for a constant positive integer r, there defines the structure of a* recursively low-rank compressed matrix $A \in \mathbb{C}^{n \times n}$ *such that*

1. *for every node i, A_{ii} is defined as $A(I_i, I_i)$, where I_i denotes the collection of indices i contains;*
2. *for every pair of siblings i and j in the tree, A_{ij} is defined as $A(I_i, I_j)$;*
3. *every such matrix block A_{ij} admits a factorization*

$$A_{ij} = U_i \Sigma_{ij} V_j^*, \tag{2.1}$$

where $\Sigma_{ij} \in \mathbb{C}^{r\times r}$*;*

4. *for every* U_i *in* (2.1)*, if* i *has a child* k*, there exists* $W_{ki} \in \mathbb{C}^{r\times r}$ *such that*

$$U_i(I_k, :) = U_k W_{ki}; \tag{2.2}$$

similarly, for every V_j *in* (2.1)*, if* j *has a child* k*, there exists* $Z_{kj} \in \mathbb{C}^{r\times r}$ *such that*

$$V_j(I_k, :) = V_k Z_{kj}. \tag{2.3}$$

Conceptually, r is small as in "low" rank; however, we impose no constraints on the magnitude of r. An example of the matrix corresponding to the partitioning tree in Figure 2.1(a) is shown in (b). Hidden in the labeling are A_{22}, which consists of the 3×3 blocks at the top left corner, and A_{44}, which consists of the 2×2 blocks at the lower right corner. Clearly, the whole matrix is A_{11}, since node 1 is the root.

It was found [6] that for some matrix operations, it is necessary to decompose the diagonal blocks A_{ii} in a manner that remedies the effect of ill conditioning. Hence, we augment Definition 2.2 with one additional requirement. Note that Definition 2.2 has already completely defined the matrix; the augmentation only serves computational purposes.

DEFINITION 2.3. *The structure of* recursively low-rank compressed matrix *is augmented such that for every node* i*, there exists* B_{ii}*, which has the same size as* A_{ii}*, and* $\Sigma_{ii} \in \mathbb{C}^{r\times r}$ *such that*

$$A_{ii} = B_{ii} + U_i \Sigma_{ii} V_i^*. \tag{2.4}$$

One can naively set $\Sigma_{ii} = 0$ and hence $B_{ii} = A_{ii}$. Using a Σ_{ii} such that B_{ii} is well conditioned, however, significantly boosts the numerical stability of several matrix operations.

The matrix components in Definitions 2.2 and 2.3 are stored with the partitioning tree in Definition 2.1.

1. For every leaf node i, A_{ii} is stored with the node. For nonleaf nodes i, A_{ii} needs not be stored.
2. The B_{ii} matrices need not be stored.
3. For every leaf node i, U_i and V_i are stored with the node. For nonleaf nodes i, U_i and V_i need not be stored.
4. For every pair of sibling nodes i and j, Σ_{ij} is stored with their parent node.
5. For every node i, Σ_{ii} is stored with the node itself. It is not stored with the parent node of i because the root does not have a parent.
6. For evert pair of parent node i and child node k, W_{ki} and Z_{ki} are stored with the child node.

Figure 2.2 illustrates the storage for the example matrix in Figure 2.1. In the balanced case (when the tree is a full and complete s-ary tree and each leaf node contains n_0 indices), the number of tree nodes is $O(n \cdot s/n_0)$ and thus storing the matrix takes space

$$O(n \cdot (r + n_0 + s^2 r^2/n_0)).$$

The matrix operations considered in [6], including matrix-vector multiplication, matrix inversion, and determinant computation, all have an $O(n)$ time cost.

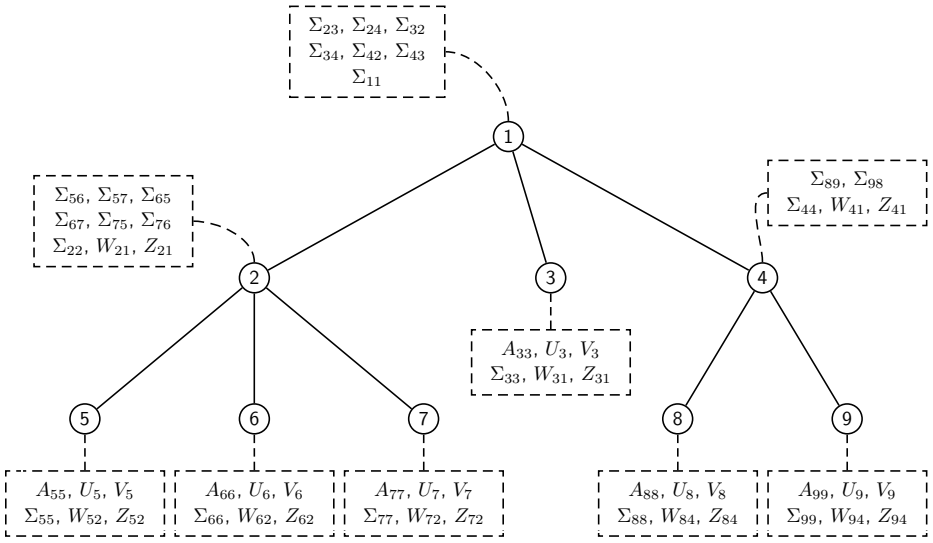


FIG. 2.2. *Data stored in the tree of Figure 2.1.*

3. Computing factorization (1.1)**.** We are to describe an algorithm for computing the square-root factorization (1.1) also in $O(n)$ time cost. We repeat the essential identities in Definitions 2.2 and 2.3 for a Hermitian matrix A:

$$A_{ij} = U_i \Sigma_{ij} U_j^* \quad \text{with} \quad \Sigma_{ij} = \Sigma_{ji}^* \quad \text{for sibling pair } i \text{ and } j, \tag{3.1}$$

$$A_{ii} = B_{ii} + U_i \Sigma_{ii} U_i^* \quad \text{with } B_{ii} \text{ and } \Sigma_{ii} \text{ Hermitian}, \tag{3.2}$$

$$U_i(I_k, :) = U_k W_{ki} \quad \text{for parent } i \text{ and child } k. \tag{3.3}$$

The objective is to construct G such that

$$G_{ij} = U_i \Omega_{ij} V_j^* \quad \text{for sibling pair } i \text{ and } j, \tag{3.4}$$

$$G_{ii} = C_{ii} + U_i \Omega_{ii} V_i^*, \tag{3.5}$$

$$U_i(I_k, :) = U_k W_{ki} \quad \text{and} \quad V_i(I_k, :) = V_k Z_{ki} \quad \text{for parent } i \text{ and child } k. \tag{3.6}$$

Note that the components U_i and W_{ki} of A are reused in G.

The strategy for constructing such a G is inductive. In the base step, we construct the factorization $B_{kk} = G_{kk} G_{kk}^*$ for all leaf nodes k. Then, suppose the factorization $B_{jj} = G_{jj} G_{jj}^*$ has been constructed for all children j of some node i, we construct the factorization $B_{ii} = G_{ii} G_{ii}^*$. Such an inductive step necessarily modifies the G_{jj} blocks and fills the $G_{jj'}$ blocks for all sibling pairs j and j'. Finally in the concluding step, we reach the root node p and construct $B_{pp} = G_{pp} G_{pp}^*$. Then, we update G_{pp} such that $A_{pp} = G_{pp} G_{pp}^*$, hence completing the construction.

The above algorithmic sketch immediately opens a number of technical developments for each step. The base step is simply a Cholesky factorization, but it requires that B_{kk} is positive definite. If this requirement is not met, we can modify Σ_{kk} to ensure so. See (3.2) and change the subscript notation from i to k. We can always subtract a positive diagonal from Σ_{kk} to increase the eigenvalues of B_{kk}. The detail of such a modification is presented in Section 5.

For the inductive step, we establish recurrence relations concerning two consecutive levels of the tree. We write A_{ii} in the following two forms, which naturally equate:

$$B_{ii} + U_i \Sigma_{ii} U_i^* \quad \text{and} \quad \begin{bmatrix} B_{jj} & & \\ & \ddots & \\ & & B_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \underline{\Lambda} \begin{bmatrix} U_j^* & & \\ & \ddots & \\ & & U_{j'}^* \end{bmatrix},$$

where $\underline{\Lambda}$ is a block matrix with the (j, j') block being $\Sigma_{jj'}$. Then, clearly,

$$B_{ii} = \begin{bmatrix} B_{jj} & & \\ & \ddots & \\ & & B_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \Lambda \begin{bmatrix} U_j^* & & \\ & \ddots & \\ & & U_{j'}^* \end{bmatrix}, \tag{3.7}$$

where the block matrix Λ is defined as

$$(j, j') \text{ block of } \Lambda = \Sigma_{jj'} - W_{ji} \Sigma_{ii} W_{j'i}^*. \tag{3.8}$$

We also write G_{ii} as

$$G_{ii} = \begin{bmatrix} G_{jj} & & \\ & \ddots & \\ & & G_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} D \begin{bmatrix} V_j^* & & \\ & \ddots & \\ & & V_{j'}^* \end{bmatrix}, \tag{3.9}$$

where D is a block matrix with the (j, j') block being $\Omega_{jj'}$. The computed G_{jj} in the lower level assume the role of C_{jj} in (3.5).

Based on (3.7) and (3.9), one set of sufficient conditions for $B_{ii} = G_{ii} G_{ii}^*$ to hold is that

$$G_{jj} V_j = U_j \tag{3.10}$$

and

$$\Lambda = D + D^* + D \Xi D^*, \tag{3.11}$$

where

$$\Xi = \begin{bmatrix} \Theta_j & & \\ & \ddots & \\ & & \Theta_{j'} \end{bmatrix} \quad \text{and} \quad \Theta_j = V_j^* V_j \text{ for all } j \in Ch(i). \tag{3.12}$$

Changing the index j in (3.10) to i and substituting (3.9) into the changed (3.10), we obtain

$$\begin{bmatrix} G_{jj} V_j Z_{ji} & & \\ & \ddots & \\ & & G_{j'j'} V_{j'} Z_{j'i} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} D \begin{bmatrix} V_j^* V_j Z_{ji} & & \\ & \ddots & \\ & & V_{j'}^* V_{j'} Z_{j'i} \end{bmatrix} = \begin{bmatrix} U_j W_{ji} \\ \vdots \\ U_{j'} W_{j'i} \end{bmatrix}.$$

A sufficient condition for this equality to hold is

$$(I + D\Xi) \begin{bmatrix} Z_{ji} \\ \vdots \\ Z_{j'i} \end{bmatrix} = \begin{bmatrix} W_{ji} \\ \vdots \\ W_{j'i} \end{bmatrix}. \tag{3.13}$$

Hence, we solve (3.11) for D and then use D to solve (3.13) for the stack of blocks Z_{ji}. Then, we compute Θ_i through the recurrence

$$\Theta_i = V_i^* V_i = \sum_{j \in Ch(i)} Z_{ji}^* V_j^* V_j Z_{ji} = \sum_{j \in Ch(i)} Z_{ji}^* \Theta_j Z_{ji},$$

based on the computed Z_{ji} factors and the Θ_j's. In light of (3.9), the (j, j') blocks of D serve as $\Omega_{jj'}$, including the case $j = j'$. In addition, the (j, j) blocks of D are used to update the corresponding G_{jj} blocks, such that they become the correct (j, j) blocks of G_{ii}. The method for solving (3.11) is discussed in Section 4.

In the concluding step, we have already constructed $B_{pp} = G_{pp} G_{pp}^*$. We want to update G_{pp} such that $A_{pp} = G_{pp} G_{pp}^*$. For this, we write

$$B_{pp} + U_p \Sigma_{pp} U_p^* = A_{pp} = (G_{pp} + U_p D V_p^*)(G_{pp}^* + V_p D^* U_p^*). \tag{3.14}$$

Because B_{pp} is equal to $G_{pp} G_{pp}^*$ before the update, then in order for (3.14) to hold, it suffices to find D that solves

$$\Sigma_{pp} = D + D^* + D \Theta_p D^*.$$

This equation has exactly the same form as (3.11). Hence, by using the same technique (to be presented in Section 4), we obtain D. Such D is used to update G_{pp}.

We have seen that all the (j, j) blocks of D are used to update the (j, j) blocks of the intermediate G_{ii}'s, and in the concluding step, D is used to update G_{pp}. For any pair of sibling nodes k and l that are descendants of j, the update reads

$$(k, l) \text{ block of } G_{ii} = [(k, l) \text{ block of } G_{jj}] + U_j(I_k, :) D_{jj} V_j(I_l, :)^*.$$

Let $(j, j_1, \ldots, j_s)$ be the path connecting j and the parent j_s of k and l. We expand the above formula as

$$(k, l) \text{ block of } G_{ii} = [(k, l) \text{ block of } G_{jj}] + U_k W_{kj_s} \cdots W_{j_1 j} D_{jj} Z_{j_1 j}^* \cdots Z_{lj_s}^* V_l^*.$$

Hence, to update the block, all we need to do is to update Ω_{kl}:

$$\Omega_{kl} \leftarrow \Omega_{kl} + W_{kj_s} \cdots W_{j_1 j} D_{jj} Z_{j_1 j}^* \cdots Z_{lj_s}^*.$$

Performing updates in this manner is, however, expensive. For a fixed pair k, l, this correction must be applied every time we construct G_{ii}, for all nodes i that are at least two levels above k, l. The repeating updates on the same block can be consolidated to only once. To this end, we define a correction term E_{kl} that is initialized as

$$E_{kl} \leftarrow W_{kj} D_{jj} Z_{lj}^*,$$

where j denotes the parent of k and l. The initialization is performed immediately after D is computed. Then, moving up the tree levels until reaching $j =$ root p, we perform a one-pass cascading correction top-down. The correction term is updated as

$$E_{kl} \leftarrow E_{kl} + W_{kj} E_{jj} Z_{lj}^*.$$

Assuming that E_{jj} has accumulated all the required corrections to Ω_{jj}, the term E_{kl} updated in this way will accumulate all the required corrections to Ω_{kl}. Thus, we correct Ω_{kl} by using

$$\Omega_{kl} \leftarrow \Omega_{kl} + E_{kl}.$$

At the bottom level (a leaf node k with $k = l$), Ω_{kk} is updated by using the same formula. In addition,

$$G_{kk} \leftarrow G_{kk} + U_k \Omega_{kk} V_k^*,$$

which concludes all the updates.

4. Solving (3.11)**.** In the real case, (3.11) is in the form of a continuous-time algebraic Riccati equation (CARE). The Schur method [13, 2] is the most robust method for solving a CARE to date. We adopt the same rationale of the method and argue that it applies to the complex case, too. The following theorem suggests a Hermitian solution to (3.11).

THEOREM 4.1. *If all the eigenvalues of $I + \Xi\Lambda$ are positive, then there admits a Schur decomposition*

$$\begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix}, \tag{4.1}$$

where the matrix containing the Q_{ij} blocks is unitary, S_{11} and S_{22} are strictly upper triangular, the diagonal of S_{11} is positive, and the diagonal of S_{22} is negative. In this case, (3.11) *has a solution $D = Q_{21}Q_{11}^{-1}$ and moreover, D is Hermitian.*

Proof. Let

$$M = \begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix}.$$

Note that for any λ, $\det(\lambda I - M) = \det(\lambda^2 I - (I + \Xi\Lambda))$. Hence, if all the eigenvalues of $I + \Xi\Lambda$ are positive, the eigenvalues of M are real. Moreover, the positive and negative eigenvalues come in pairs. Thus, M admits a Schur decomposition $MQ = QS$, where Q is unitary, S is upper triangular, and all the diagonal elements of S are real. The diagonal can always be reordered by using unitary transformations while maintaining the upper triangularity of S. Then, we reorder S such that the positive eigenvalues appear in the first part of the diagonal. Such a reordering result is (4.1).

We consider the part of (4.1) that corresponds to the invariant subspace associated with S_{11}:

$$\begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} S_{11}.$$

We multiply a common factor to the left and obtain

$$\begin{bmatrix} Q_{11}^* & Q_{21}^* \end{bmatrix} \begin{bmatrix} & -I \\ I & \end{bmatrix} \begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = \begin{bmatrix} Q_{11}^* & Q_{21}^* \end{bmatrix} \begin{bmatrix} & -I \\ I & \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} S_{11}.$$

Since the left-hand side of the above equation is Hermitian, so must be the right-hand side, which is simplified to $(Q_{11}^* Q_{21} - Q_{21}^* Q_{11}) S_{11}$. Because S_{11} is upper triangular

with diagonal elements having the same sign, and because $Q_{11}^* Q_{21} - Q_{21}^* Q_{11}$ is skew Hermitian, we can verify entry-by-entry that

$$Q_{11}^* Q_{21} - Q_{21}^* Q_{11} = 0. \tag{4.2}$$

Then, by the unitarity of the matrix consisting of the Q_{ij} blocks (i.e., $Q_{11}^* Q_{12} + Q_{21}^* Q_{22} = 0$), we obtain

$$-Q_{12} Q_{22}^{-1} = Q_{21} Q_{11}^{-1}. \tag{4.3}$$

Combining (4.2) and (4.3), we have

$$Q_{21} Q_{11}^{-1} = (Q_{21} Q_{11}^{-1})^*. \tag{4.4}$$

On the other hand, multiplying common factors to both sides of (4.1), we obtain

$$\begin{bmatrix} 0 & -Q_{22}^{-*} \end{bmatrix} \begin{bmatrix} Q_{11}^* & Q_{21}^* \\ Q_{12}^* & Q_{22}^* \end{bmatrix} \begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} Q_{11}^{-1} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & -Q_{22}^{-*} \end{bmatrix} \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix} \begin{bmatrix} Q_{11}^{-1} \\ 0 \end{bmatrix}.$$

Simplifying the equation yields

$$\begin{bmatrix} -Q_{22}^{-*} Q_{12}^* & -I \end{bmatrix} \begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} I \\ Q_{21} Q_{11}^{-1} \end{bmatrix} = 0.$$

Thus, by letting

$$D = -Q_{22}^{-*} Q_{12}^* = (Q_{21} Q_{11}^{-1})^*,$$

we have

$$\begin{bmatrix} D & -I \end{bmatrix} \begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} I \\ D^* \end{bmatrix} = 0,$$

which is exactly the equation (3.11). Clearly, D is Hermitian in light of (4.4). ∎

One must pay special attention to the Schur decomposition when implementing different cases. In the real case, we use the real Schur decomposition (LAPACK routine `DGEES`) so that the unitary matrix is real. In the complex case, we use the usual Schur decomposition (LAPACK routine `ZGEES`). Both resulting Schur forms are upper triangular with a real diagonal, as guaranteed by Theorem 4.1.

The remaining question is how to ensure that the condition of Theorem 4.1 holds. Recall that at the beginning of Section 3, we mention that for all leaf nodes k, B_{kk} may need to be modified to attain positive definiteness. In fact, the modification must be applied on all tree nodes.

THEOREM 4.2. *The eigenvalues of $I + \Xi\Lambda$ are positive if and only if B_{ii} is positive definite.*

Proof. Based on (3.8) and (3.12), we expand $I + \Xi\Lambda$ as

$$I + \begin{bmatrix} V_j^* V_j & & \\ & \ddots & \\ & & V_{j'}^* V_{j'} \end{bmatrix} \underline{\Lambda} - \begin{bmatrix} V_j^* V_j & & \\ & \ddots & \\ & & V_{j'}^* V_{j'} \end{bmatrix} \begin{bmatrix} W_{ji} \\ \vdots \\ W_{j'i} \end{bmatrix} \Sigma_{ii} \begin{bmatrix} W_{ji}^* & \cdots & W_{j'i}^* \end{bmatrix}.$$

Let $EIG(\cdot)$ denote the set of eigenvalues of a matrix. We will invoke a well known result, which states that $EIG(I+XY)$ is either a superset or a subset of $EIG(I+YX)$,

and the set difference between the two consists of element(s) 1. Hence, the eigenvalues of $I + \Xi\Lambda$ are positive if and only if the Hermitian matrix

$$I + \begin{bmatrix} V_j & & \\ & \ddots & \\ & & V_{j'} \end{bmatrix} \underline{\Lambda} \begin{bmatrix} V_j^* & & \\ & \ddots & \\ & & V_{j'}^* \end{bmatrix} - \begin{bmatrix} V_j W_{ji} \\ \vdots \\ V_j W_{j'i} \end{bmatrix} \Sigma_{ii} \begin{bmatrix} W_{ji}^* V_j^* & \cdots & W_{j'i}^* V_{j'}^* \end{bmatrix} \quad (4.5)$$

is positive definite. Because the G_{jj}'s are square and nonsingular, we can multiply the block diagonal matrix $\mathrm{diag}[G_{jj}]$ to the left of (4.5) and multiply $\mathrm{diag}[G_{jj}^*]$ to the right; then, the result will not change the definiteness. By noting that $G_{jj}V_j = U_j$, the multiplication result is

$$\begin{bmatrix} G_{jj}G_{jj}^* & & \\ & \ddots & \\ & & G_{j'j'}G_{j'j'}^* \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \underline{\Lambda} \begin{bmatrix} U_j^* & & \\ & \ddots & \\ & & U_{j'}^* \end{bmatrix} - U_i \Sigma_{ii} U_i^*.$$

This result is simply B_{ii}, hence concluding the proof. □

As a result, it suffices to modify B_{ii} to ensure that the condition of Theorem 4.1 holds. Because both B_{ii} and Λ are determined by Σ_{ii}, equivalently, we can modify Λ to achieve the same effect.

The Schur method suggested by Theorem 4.1 for solving (3.11) is generally not the most economic—a price paid for robustness. Other methods exist based on factorizations of smaller matrices. For example, because Ξ is positive semidefinite, we may write $\Xi = RR^*$ for some R. Then, (3.11) leads to

$$I + R^*\Lambda R = (I + R^*DR)(I + R^*DR)^*.$$

Hence, writing $I + R^*\Lambda R = SS^*$, we obtain a solution $D = R^{-*}(S - I)R^{-1}$. The factorizations of Ξ and $I + R^*\Lambda R$ can be derived from Cholesky, eigen-decomposition, or even QR factorization, each of which results in a slightly different cost for finally obtaining D. When Ξ is singular, it suffices to replace the inverse in D by pseudoinverse with a careful examination of the null space. The disadvantage of these methods, however, is that they are generally not as robust as the Schur method. A similar discussion is held in [2].

5. Modifying Σ_{ii}. We have seen in both Sections 3 and 4 that B_{ii} must be modified to ensure positive definiteness for all nodes i, if it was not so originally. The modification is achieved by subtracting a positive constant diagonal from the corresponding Σ_{ii}; that is, new $\Sigma_{ii} \leftarrow \Sigma_{ii} - tI$, $t > 0$. This section presents methods for computing a reasonable t. The cases for Sections 3 and 4 are separate.

Section 3 concerns leaf nodes i. The goal is to find a positive value t such that

$$\text{new } B_{ii} \leftarrow A_{ii} - U_i(\Sigma_{ii} - tI)U_i^*$$

is positive definite. Let λ be the smallest eigenvalue of the matrix pencil

$$(A_{ii} - U_i\Sigma_{ii}U_i^*,\ U_iU_i^*),$$

and assume that λ is nonpositive (otherwise, the original B_{ii} is already positive definite). Clearly, all t greater than $-\lambda$ will make the new B_{ii} positive definite. Hence, a straightforward approach is to let $t = -c\lambda$ for some $c > 1$. Numerical experiments suggest that setting c close 1, e.g., 1.5, is often sufficient.

A more sophisticated approach that concerns numerical stability is to recall that the Cholesky factor G_{ii} of B_{ii} is used to compute $V_i = G_{ii}^{-1}U_i$. Thus, we want the new B_{ii} to be as well conditioned as possible. This requirement leads to

$$t = \operatorname*{argmin}_{t>-\lambda} f(t) \quad \text{where} \quad f(t) = \operatorname{cond}(A_{ii} - U_i(\Sigma_{ii} - tI)U_i^*). \tag{5.1}$$

When $t > -\lambda$, the matrix $A_{ii} - U_i(\Sigma_{ii} - tI)U_i^*$ is always positive definite. Hence, the singular values coincide with the eigenvalues. Then, by the continuity of the extreme eigenvalues, the condition number of the matrix is continuous. In this case, several optimization algorithms are applicable for solving (5.1), including the interior point algorithm and the sequential quadratic programming algorithm [4]. A drawback of this approach is that solving (5.1) requires repeatedly computing condition numbers, which may be very expensive.

Section 4 concerns nonleaf nodes i. Because Σ_{ii} directly affects Λ (cf. (3.8)) and because Λ is the only modifiable component in $I + \Xi\Lambda$, we modify Λ to attain the positive definiteness of B_{ii} (cf. Theorem 4.2). To be specific, we find a positive value t and renew Λ as $\Lambda + tWW^*$ such that the eigenvalues of

$$I + \Xi(\Lambda + tWW^*)$$

are all positive, where

$$W = \begin{bmatrix} W_{ji} \\ \vdots \\ W_{j'i} \end{bmatrix}.$$

Let λ be the smallest eigenvalue of the matrix pencil

$$(I + \Xi\Lambda,\ \Xi WW^*), \tag{5.2}$$

and assume that λ is nonpositive. Clearly, any $t > -\lambda$ will ensure that the eigenvalues of $I + \Xi(\Lambda + tWW^*)$ are positive.

As in the preceding discussion, we may simply set $t = -c\lambda$ for some $c > 1$. Alternatively, we may impose a requirement that improves numerical behavior. For example, we would like a t that minimizes the conditioning of the Riccati equation (3.11). Unfortunately, such a requirement does not seem practically addressable. It was pointed out that "several proposed condition numbers ... are compared and all are shown to have deficiencies for some classes of problems" [2]. In light of this difficulty, we elect an artificial requirement similar to (5.1):

$$t = \operatorname*{argmin}_{t>-\lambda} f(t), \quad f(t) = \operatorname{cond}(I + \Xi(\Lambda + tWW^*)). \tag{5.3}$$

Such a requirement has the same drawback as (5.1) does: The optimization of the condition number is costly.

A numerical issue for computing the smallest eigenvalue λ of the matrix pencil (5.2) is that both matrices in the pencil are nonHermitian. Hence, reliably computing the eigenvalues is sometimes difficult. A more robust approach is to preprocess the pencil by symmetrization. For this, we let $\Xi = YY^*$. Then, the eigenvalues of (5.2) are the same as those of the pencil

$$(I + Y^*\Lambda Y,\ Y^*WW^*Y). \tag{5.4}$$

The computed eigenvalues of (5.4) are more reliable.

It is worth noting that if the Σ_{ii}'s are all zero, then the B_{ii}'s are positive definite (because $B_{ii} = A_{ii}$) and thus we do not need to modify any of them. In other words, the computations of the smallest eigenvalues are waived. Experimental results, however, generally favor the approach of maintaining the nonzero definitions of Σ_{ii} and modifying Σ_{ii} to ensure positive definiteness.

6. Formal algorithm. We summarize the overall calculation (Sections 3 to 5) in Algorithm 1. For implementation, we first need to augment the tree data structure discussed in Section 2 for storing intermediate results:

1. $\Theta_i \in \mathbb{C}^{r\times r}$, for all nodes i, and
2. $E_{jj'} \in \mathbb{C}^{r\times r}$, for all sibling pairs j, j', including $j = j'$.

Naturally, Θ_i is stored with the node i. For the $E_{jj'}$'s, in order to be consistent with the storage of the $\Omega_{jj'}$'s, $E_{jj'}$ is stored with the parent node of j and j' when $j \neq j'$, but is otherwise stored with the node j itself when $j = j'$. Because the storage added to each node is constant, and because the number of tree nodes is $O(n)$, clearly the augmented data structure maintains the $O(n)$ complexity.

Algorithm 1 comprises two parts, an upward pass and a downward pass. The upward pass is a postorder tree traversal, where components $\Omega_{jj'}$ and Z_{ji} are computed at the child level and the calculation moves up to the parent level. At the beginning of the pass, the components V_k for all leaf nodes k are computed and the components G_{kk} are initialized. During the pass, the correction terms E_{kl} are also initialized.

The downward pass is a preorder tree traversal, where the correction terms $E_{jj'}$ and the components $\Omega_{jj'}$ are updated at the parent level and the calculation moves down to the child level. At the end of the pass, the components G_{kk} for all leaf nodes k are also updated. This concludes the calculation.

The resulting matrix G carries over the components U_i and W_{ki} from A.

In the two passes, the subroutine SolveRiccati solves the Riccati equation (Section 4) and the subroutines Modify1 and Modify2 modify the Σ_{ii}'s (Section 5). Modify1 is responsible for the case when i is a leaf, and Modify2 is responsible for the other case.

7. Cost analysis. We base the analysis on the context of a balanced partitioning tree, where the tree is a full and complete s-ary tree and each leaf node contains n_0 indices. The time cost of the overall algorithm is thus $O(n)$, because the work at each tree node is constant and there are $O(n)$ nodes.

We perform a deeper analysis and calculate the prefactor hidden in the $O(n)$ notation. First, we consider the cost of three subroutines. Modify1 works on a matrix A_{ii} that has a size $n_0 \times n_0$; thus, the cost is $O(n_0^3)$. Similarly, Modify2 works on a matrix Λ that has a size $sr \times sr$ and thus the cost is $O((sr)^3)$. The subroutine SolveRiccati clearly has a cost $O((2sr)^3)$.

Then, we break the work of the overall algorithm in two parts—that for the leaf nodes and that for the nonleaf nodes. The work for leaf nodes contains the first "if" clause in both subroutines Upward and Downward. Not counting the work for Modify1, the cost per leaf node is $O(n_0^3 + n_0^2 r + n_0 r^2)$. Then, including Modify1, the total work for all leaf nodes is

$$O(n/n_0 \cdot (n_0^3 + n_0^2 r + n_0 r^2)), \tag{7.1}$$

because there are n/n_0 leaf nodes.

The work for nonleaf nodes include the rest of Upward below line 12 and the rest of Downward below line 33. The work is dominated by the calculations with

Algorithm 1 Factorizing $A = GG^*$ for a Hermitian positive definite A

1: Copy all components U_i and W_{ki} to G
2: UPWARD(`root`)
3: DOWNWARD(`root`)

4: **subroutine** UPWARD(i)
5: **if** i is leaf **then**
6: Modify Σ_{ii} by using subroutine MODIFY1(A_{ii}, U_i, Σ_{ii})
7: Factorize $A_{ii} - U_i\Sigma_{ii}U_i^* = G_{ii}G_{ii}^*$; $V_i \leftarrow G_{ii}^{-1}U_i$; $\Theta_i \leftarrow V_i^*V_i$; return
8: **end if**
9: **for all** children j of i **do**
10: UPWARD(j)
11: **end for**
12: Build the block matrix Λ and the diagonal-block matrix Ξ where

$$(j, j') \text{ block of } \Lambda = \Sigma_{jj'} - W_{ji}\Sigma_{ii}W_{j'i}^*,$$
$$(j, j) \text{ block of } \Xi = \Theta_j.$$

13: Modify Σ_{ii} by using subroutine MODIFY2(Λ, Ξ, W_{ji}, $\forall j \in Ch(i)$)
14: Recompute Λ by using the modified Σ_{ii}
15: Solve the equation $\Lambda = D + D^* + D\Xi D^*$ for D by SOLVERICCATI(Λ, Ξ)
16: **for all** children j, j' of i (including $j = j'$) **do** $\Omega_{jj'} \leftarrow D_{jj'}$ **end for**
17: **for all** children j of i **do**
18: $E_{kl} \leftarrow W_{kj}\Omega_{jj}Z_{lj}^*$ for all children k, l of j (including $k = l$)
19: **end for**
20: Compute

$$\begin{bmatrix} Z_{ji} \\ \vdots \\ Z_{j'i} \end{bmatrix} \leftarrow (I + D\Xi)^{-1} \begin{bmatrix} W_{ji} \\ \vdots \\ W_{j'i} \end{bmatrix}.$$

21: Compute $\Theta_i \leftarrow \sum_{j \in Ch(i)} Z_{ji}^*\Theta_j Z_{ji}$
22: **if** i is root **then**
23: Perform the calculation in line 15, where $\Lambda = \Sigma_{ii}$ and $\Xi = \Theta_i$
24: $\Omega_{ii} \leftarrow D$
25: $E_{jj'} \leftarrow W_{ji}\Omega_{ii}Z_{j'i}^*$ for all children j, j' of i (including $j = j'$)
26: $E_{ii} \leftarrow 0$
27: **end if**
28: **end subroutine**
Continued in Algorithm 2...

matrices of size $sr \times sr$. Then, the cost per nonleaf node is $O((sr)^3)$. Considering that the number of nonleaf nodes is $O(n/n_0)$ and that subroutines MODIFY2 and

Algorithm 2 Factorizing $A = GG^*$, continued from Algorithm 1

subroutine DOWNWARD(i)
  if i is leaf **then**
    $G_{ii} \leftarrow G_{ii} + U_i \Omega_{ii} V_i^*$
  else
    for all children j, j' of i (including $j = j'$) **do**
      $E_{jj'} \leftarrow E_{jj'} + W_{ji} E_{ii} Z_{j'i}^*$
      $\Omega_{jj'} \leftarrow \Omega_{jj'} + E_{jj'}$
    end for
    for all children j of i **do** DOWNWARD(j) **end for**
  end if
end subroutine

subroutine MODIFY1(A_{ii}, U_i, Σ_{ii})
  Compute the smallest eigenvalue λ of $(A_{ii} - U_i \Sigma_{ii} U_i^*,\ U_i U_i^*)$
  if $\lambda > 0$ **then**
    set $t \leftarrow 0$
  else
    Set $t \leftarrow -c\lambda$ for some $c > 1$ ▷ more efficient
    or set $t \leftarrow \operatorname{argmin}_{t > -\lambda} \operatorname{cond}(A_{ii} - U_i(\Sigma_{ii} - tI)U_i^*)$ ▷ better conditioned
  end if
  return $\Sigma_{ii} \leftarrow \Sigma_{ii} - tI$
end subroutine

Continued in Algorithm 3...

SOLVERICCATI are called, we have the total work for all nonleaf nodes

$$O(n/n_0 \cdot s^3 r^3). \tag{7.2}$$

Summing (7.1) and (7.2), the time cost of the overall algorithm is

$$O(n \cdot (n_0^2 + n_0 r + r^2 + s^3 r^3 / n_0)).$$

8. Generation of compressed matrix. In this section, we consider two settings for generating a recursively low-rank compressed matrix A that is Hermitian positive definite.

A random matrix is useful for numerically verifying the correctness of the proposed algorithm. Following the method in Appendix A of [6], we first construct a random tree. With certain probability, a node can span a number of children, where the number randomly falls in a prescribed range. Then, we instantiate the random components of the matrix in the tree. In order for the matrix to be Hermitian, the U_k components must be the same as V_k, W_{ki} be the same as Z_{ki}, and Σ_{ij} be the same as Σ_{ji}^*, including the case $i = j$.

Positive definiteness requires additional efforts. For a nonleaf node i, recall

$$A_{ii} = \begin{bmatrix} B_{jj} & & \\ & \ddots & \\ & & B_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \underline{\Lambda} \begin{bmatrix} U_j^* & & \\ & \ddots & \\ & & U_{j'}^* \end{bmatrix}, \tag{8.1}$$

Algorithm 3 Factorizing $A = GG^*$, continued from Algorithm 2

subroutine MODIFY2(Λ, Ξ, W_{ji}, $\forall j \in Ch(i)$)
  Let the vertical stacking of W_{ji} be W
  Perform factorization $\Xi = YY^*$
  Compute the smallest eigenvalue λ of $(I + Y^*\Lambda Y,\ Y^*WW^*Y)$
  if $\lambda > 0$ **then**
    set $t \leftarrow 0$
  else
    Set $t \leftarrow -c\lambda$ for some $c > 1$ ▷ more efficient
    or set $t \leftarrow \operatorname{argmin}_{t>-\lambda} \operatorname{cond}(I + \Xi(\Lambda + tWW^*))$ ▷ better conditioned
  end if
  return $\Sigma_{ii} \leftarrow \Sigma_{ii} - tI$
end subroutine

subroutine SOLVERICCATI(Λ, Ξ)
  Perform Schur decomposition with reordering of eigenvalues

$$\begin{bmatrix} I & \Xi \\ \Lambda & -I \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix},$$

  such that S_{11} is strictly upper triangular and its diagonal is positive.
  return $D \leftarrow Q_{21}Q_{11}^{-1}$
end subroutine

where $\underline{\Lambda}$ is a block matrix with the (j, j') block being $\Sigma_{jj'}$. Hence, for A to be positive definite, it suffices to let all such $\underline{\Lambda}$ be positive definite, and in addition, let B_{kk} be positive definite for all leaf nodes k. View (8.1) in a recursive manner and assume that j is a leaf. In the leaf level, B_{jj} is initially instantiated as a random positive definite matrix. Then, in the parent level i, a Hermitian positive definite term $U_j\Sigma_{jj'}U_j^*$ is added to B_{jj}. Moving up along the tree, in every level a Hermitian positive definite term will be added to B_{jj}. After the accumulation of the terms in all levels, B_{jj} becomes the final A_{jj}. The recursive accumulation can be done in a manner similar to the calculation of the correction terms in the factorization algorithm, which makes the matrix generation process $O(n)$.

Another setting for generating A is the compression of a kernel matrix Φ. We follow Appendix B of [6] to perform such a compression. The basic idea is a k-d tree partitioning of the point set with a Chebyshev interpolation of the kernel [5, 10]. As explained in Section 1, a regularization term δ is always present in the kernel. Such a term ensures the positive definiteness of A.

9. Numerical experiments. In this section, we show a comprehensive set of experiments to demonstrate the numerical behavior and the computational cost of the proposed algorithm. We also display results of a sampling application. The program is written in C++, where the basic matrix operations are called from BLAS and LAPACK provided by Intel MKL. The modifications of Σ_{ii} are all done by setting $t = -1.5\lambda$ without performing optimization. The results for using optimizations are similar, but the time cost is substantially higher. We add a subscript to G to distinguish the factorization results from different methods: "Chol" means the stan-

dard Cholesky factor, and "Alg1" means the square-root factor computed by using Algorithm 1. The machine precision `eps = 2.2e-16`.

9.1. Random complex matrix. We first experiment with a random complex matrix of size approximately $2,000$. The parameters for generating this matrix are given in the caption of Table 9.1. The matrix is ill-conditioned, with a 2-norm condition number `1.4e+09`.

TABLE 9.1
Computation results for a random matrix. $n = 1,996$. *Parameters: budgeted number of leaves* $= 50$, *probability* $p = 0.5$, *range of number of children* $[3, 5]$, *range of leaf size* $[30, 50]$, *rank* $r = 10$.

$\mathrm{cond}_2(A)$	`1.4e+09`	$\|A - G_{\mathrm{Chol}} G_{\mathrm{Chol}}^*\|_F/\sqrt{n}$	`3.2e-10`
		$\|A - G_{\mathrm{Alg1}} G_{\mathrm{Alg1}}^*\|_F/\sqrt{n}$	`9.4e-08`
		abs$[\ b^*Ab - \|G_{\mathrm{Chol}}^* b\|_2^2\ ]$	`2.8e-10`
		abs$[\ b^*Ab - \|G_{\mathrm{Alg1}}^* b\|_2^2\ ]$	`1.3e-10`

By construction, the B_{ii}'s are all positive definite, hence we see that no Σ_{ii} is modified. The solutions to the Riccati equation (3.11) all attain a 2-norm residual on the order of `1.0e-13`.

We use two error metrics to evaluate the accuracy of the factorization. One is the Frobenius norm of $A - GG^*$ normalized by $\sqrt{n}$; the other is the absolute difference between b^*Ab and $\|G^*b\|_2^2$, where b is a normalized random vector from the standard Gaussian. Both values are bounded by the 2-norm of $A - GG^*$; they are cheaper to compute than the 2-norm. The second metric is ad hoc but it can be computed in $O(n)$ time. It is the only metric affordable to compute when n is large.

Table 9.1 shows the errors. For the first metric, the error produced by the proposed algorithm is moderately close to that produced by the Cholesky factorization. For the second metric, the errors are on the same order, with the one of the proposed algorithm slightly better.

9.2. Kernel matrices. We experiment with three positive definite kernels. Recall that δ denotes the variance of measurement error in the context of Gaussian processes. The Matérn kernel is defined as

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{M}_\nu(\hat{\boldsymbol{x}} - \hat{\boldsymbol{y}}) + \delta(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}), \tag{9.1}$$

where

$$\mathrm{M}_\nu(\boldsymbol{r}) = \frac{\|\boldsymbol{r}\|_2^\nu K_\nu(\|\boldsymbol{r}\|_2)}{2^{\nu-1}\Gamma(\nu)}, \quad \hat{\boldsymbol{x}} = \left[\frac{x_1}{\ell_1}, \ldots, \frac{x_d}{\ell_d}\right], \quad \delta(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} \delta, & \boldsymbol{x} = \boldsymbol{y} \\ 0, & \boldsymbol{x} \neq \boldsymbol{y}, \end{cases}$$

and K_ν is the modified Bessel function of second kind of order ν. The hat notation on top of a d-dimensional point $\boldsymbol{x}$ means scaling by the range parameters ℓ_1, ℓ_2, $\ldots$, ℓ_d along each coordinate. Hence, the kernel is anisotropic if the ℓ_i's are different. Using the same scaling notation, the Gaussian kernel is

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{G}(\hat{\boldsymbol{x}} - \hat{\boldsymbol{y}}) + \delta(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}), \quad \text{where} \quad \mathrm{G}(\boldsymbol{r}) = \exp(-\|\boldsymbol{r}\|_2^2/2). \tag{9.2}$$

The Gaussian kernel is a Matérn kernel with order $\nu = \infty$. The periodic Gaussian kernel is defined as

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{P}(\boldsymbol{x} - \boldsymbol{y}) + \delta(\boldsymbol{x}, \boldsymbol{y}), \quad \text{where} \quad \mathrm{P}(\boldsymbol{r}) = \ell_1 \exp\left(-\frac{1}{\ell_2}\sum_{i=1}^{d} (\sin \pi r_i)^2\right). \tag{9.3}$$

This kernel is isotropic and it has a periodicity 1 along each dimension.

The computation results are summarized in Tables 9.2 to 9.4. Recall that Φ is the kernel matrix and A is its recursively low-rank compression. The detailed settings are given in the captions of the tables. The parameter k is the Chebyshev order in the low-rank compression. Hence, the "rank" r is equal to $(k+1)^d$ for a d-dimensional kernel. The kernels are instantiated in one or two dimensions, the size of the matrices ranges from 1,000 to 10,000, the regularization δ ranges from 10^{-4} to 10^{-2}, and the order of condition numbers ranges from `1.0e+05` to `1.0e+07`. One of the compressions (two-dimensional Gaussian) is highly accurate, whereas the other two are moderate. The factorization errors under both metrics are comparable with those of the Cholesky factorization.

TABLE 9.2
Computation results for a kernel matrix. 1D Matérn kernel (9.1). *Points uniform on* $[0,1]$. *Parameters:* $\nu = 1$, $\boldsymbol{\ell} = 1$, $\delta = 10^{-4}$, $n = 1,000$, $n_0 = 60$, $k = 15$.

$\dfrac{\|A-\Phi\|_F}{\|\Phi\|_F}$	`5.0e-08`	$\|A - G_{\mathrm{Chol}} G_{\mathrm{Chol}}^*\|_F/\sqrt{n}$	`7.3e-15`
		$\|A - G_{\mathrm{Alg1}} G_{\mathrm{Alg1}}^*\|_F/\sqrt{n}$	`1.0e-11`
$\mathrm{cond}_2(\Phi)$	`8.4e+06`	abs[$b^*Ab - \|G_{\mathrm{Chol}}^* b\|_2^2$]	`4.9e-15`
$\mathrm{cond}_2(A)$	`9.4e+06`	abs[$b^*Ab - \|G_{\mathrm{Alg1}}^* b\|_2^2$]	`3.7e-13`

TABLE 9.3
Computation results for a kernel matrix. 2D Gaussian kernel (9.2). *Points uniform on* $[0,1]^2$. *Parameters:* $\boldsymbol{\ell} = [1;2]$, $\delta = 10^{-4}$, $n = 4,000$, $n_0 = 200$, $k = 15$.

$\dfrac{\|A-\Phi\|_F}{\|\Phi\|_F}$	`1.1e-14`	$\|A - G_{\mathrm{Chol}} G_{\mathrm{Chol}}^*\|_F/\sqrt{n}$	`1.0e-14`
		$\|A - G_{\mathrm{Alg1}} G_{\mathrm{Alg1}}^*\|_F/\sqrt{n}$	`6.3e-11`
$\mathrm{cond}_2(\Phi)$	`3.6e+07`	abs[$b^*Ab - \|G_{\mathrm{Chol}}^* b\|_2^2$]	`3.1e-15`
$\mathrm{cond}_2(A)$	`3.6e+07`	abs[$b^*Ab - \|G_{\mathrm{Alg1}}^* b\|_2^2$]	`1.8e-13`

TABLE 9.4
Computation results for a kernel matrix. 2D periodic Gaussian kernel (9.3). *Points uniform on* $[0,1]^2$. *Parameters:* $\boldsymbol{\ell} = [1;2]$, $\delta = 10^{-2}$, $n = 10,000$, $n_0 = 200$, $k = 15$.

$\dfrac{\|A-\Phi\|_F}{\|\Phi\|_F}$	`7.1e-07`	$\|A - G_{\mathrm{Chol}} G_{\mathrm{Chol}}^*\|_F/\sqrt{n}$	`1.4e-14`
		$\|A - G_{\mathrm{Alg1}} G_{\mathrm{Alg1}}^*\|_F/\sqrt{n}$	`9.7e-13`
$\mathrm{cond}_2(\Phi)$	`6.3e+05`	abs[$b^*Ab - \|G_{\mathrm{Chol}}^* b\|_2^2$]	`1.8e-14`
$\mathrm{cond}_2(A)$	`6.9e+05`	abs[$b^*Ab - \|G_{\mathrm{Alg1}}^* b\|_2^2$]	`1.5e-14`

9.3. Scaling. We use the two-dimensional periodic Gaussian kernel (9.3) to perform a scaling test by varying n from one thousand to one million. The loss in compression becomes more and more severe when n grows, hence we use a sufficiently large regularization ($\delta = 10^2$) to ensure that the compressed matrix A is positive definite for the largest n. The computations are serial (no multithreading). The timings are plotted in Figure 9.1, with the settings given in the figure caption. We include the matrix operations considered in [6] for comparison. Clearly, all matrix operations,

including the square root factorization proposed in this paper, nicely follow the $O(n)$ trend. We see that performing square root factorization is more time consuming than inverting the matrix and other operations. The factorization errors corresponding to the six diamond dots from left to right, are `7.1e-14`, `9.1e-12`, `6.7e-12`, `1.3e-11`, `5.3e-12`, and `2.5e-12`, respectively. The error corresponds to the second metric abs[$b^*Ab - \|G^*b\|_2^2$].

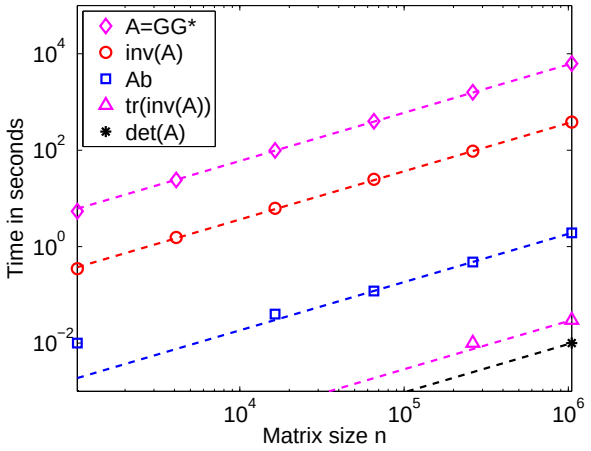
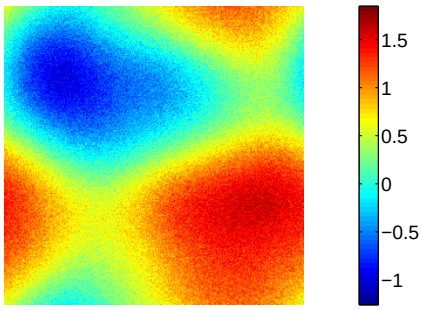


FIG. 9.1. *Running time versus matrix dimension n. 2D periodic Gaussian kernel* (9.3). *Points uniform on* $[0,1]^2$. *Parameters:* $\boldsymbol{\ell} = [1;2]$, $\delta = 10^2$, $n_0 = 128$, $k = 10$. *Dashed lines indicate linear scaling.*

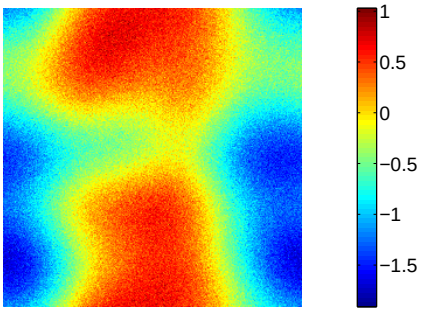
9.4. Application: Gaussian sampling. We sample a zero-mean Gaussian process on $[0,1]^2$ with the two-dimensional periodic kernel (9.3). Although algorithms for compressed matrices are generally intended for scattered points, for visualization purpose, we sample an $m \times m$ regular grid. The coordinates of the grid points along each dimension are 0, $1/m$, $2/m$, $\ldots$, $(m-1)/m$. The two plots on the top row of Figure 9.2 are two examples, where $m = 256$ (hence $n = 65,536$). Here, the measurement error has a variance $\delta = 10^{-2}$.

How do we know that the computed results are what supposed to look like? We examine three aspects. First, the factorization error (by using the second metric) is `1.2e-14`, an indication of high accuracy. Second, the parallel sides of a square sample match each other, indicating that periodicity is obeyed. Third, we use a different method to generate samples and compare. Because of the periodicity of the kernel and the regular grid structure, the uncompressed covariance matrix Φ is multilevel circulant. Hence, we can use multidimensional FFT to diagonalize Φ and thus obtain $\Phi^{1/2}$. Using this method, a Gaussian sample is generated, as shown in plot (c). Because this method essentially computes a different factorization, it is practically impossible to generate samples that match (a) or (b). Hence, the comparison focuses on the pattern of randomness but not the numerical values. We see that plot (c) shares many characteristics with (a) and (b): all plots have bulging peaks and valleys, and they appear to be filled with the same level of salt-and-pepper noise. The "noise" is caused by δ, which governs the variance of error. We make $\delta = 0$ and use the FFT method to regenerate the sample (see plot (d)). The contrast between (c) and (d) clearly confirm the noise effect of δ. These three aspects together show that the sampling results (a) and (b) computed by the proposed algorithm are reliable.

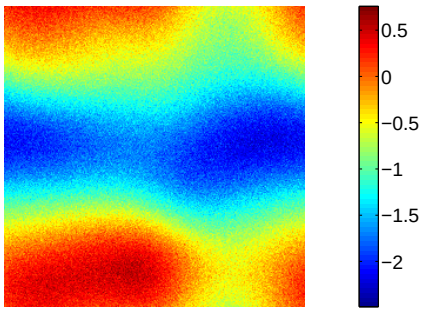
Moreover, the presence of δ causes noise in the sample, which would have been a smooth surface were $\delta = 0$.



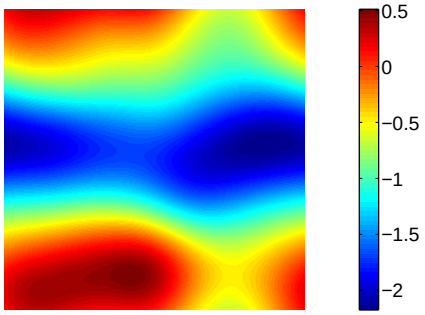
(a) Sample generated by Algorithm 1



(b) Another sample generated by Algorithm 1



(c) Sample generated by FFT



(d) Sample generated by FFT, $\delta = 0$

FIG. 9.2. *Gaussian samples of the 2D periodic Gaussian kernel* (9.3). *Resolution* 256×256. *Grid on* $[0,1]^2$. *Parameters:* $\boldsymbol{\ell} = [1;2]$. *If* $\delta \neq 0$, *then* $\delta = 10^{-2}$.

10. Concluding remarks. We have developed an $O(n)$ algorithm for factorizing a Hermitian positive definite matrix A into a square root form GG^*. The algorithm is based on the recursively low-rank compressed data structure defined in [6] and it returns a factor G with the same structure. The proposed algorithm carries the same signature of all the algorithms in [6]: two tree traversals (first postorder, then preorder) and local data dependence. Experimental results with both random matrices and kernel matrices show that the accuracy of the factorization is comparable with that of the standard Cholesky factorization, where the matrix is treated fully dense. The proposed factorization can be used for generating samples from a multivariate normal distribution or a Gaussian process, resulting in an overall $O(n)$ cost for sampling.

Challenges remain to ensure the positive definiteness of A in the compression of a kernel matrix Φ. In this paper, the compression is done by using Chebyshev interpolation [5, 10]. When the matrix size increases, the off-diagonal blocks gradually lose accuracy when moving up along the tree levels. Because in the Fourier domain, the tail of a positive definite kernel approaches zero, this means that Φ tends to have tiny eigenvalues when the size increases. As a result, the compression may cause loss of positive definiteness. In some applications, e.g., Gaussian processes, a regularization term naturally appears in the model, hence positive definiteness may

still be maintained. However, if the regularization is small (e.g., when data is close to noise-free), the loss of positive definiteness becomes a roadblock. An immediate future work is to design compression methods that better preserve the spectrum of kernel matrices.

REFERENCES

[1] S. Ambikasaran and M. O'Neil, *Fast symmetric factorization of hierarchical matrices with applications*. arXiv preprint arXiv:1405.0223, 2014.

[2] W. Arnold and A. J. Laub, *Generalized eigenproblem algorithms and software for algebraic Riccati equations*, Proceedings of the IEEE, 72 (1984), pp. 1746–1754.

[3] M. Bebendorf and W. Hackbusch, *Stabilized rounded addition of hierarchical matrices*, Numer. Lin. Alg. Appl., 4 (2007), pp. 407–423.

[4] D. P. Bertsekas, *Nonlinear Programming*, Athena Scientific, 2nd ed., 1999.

[5] S. Chandrasekaran, M. Gu, and T. Pals, *A fast ULV decomposition solver for hierarchically semiseparable representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 603–622.

[6] J. Chen, *Data structure and algorithms for recursively low-rank compressed matrices*, Tech. Rep. ANL/MCS-P5112-0314, Argonne National Laboratory, 2014.

[7] J. Chen, M. Anitescu, and Y. Saad, *Computing $f(A)b$ via least squares polynomial approximations*, SIAM J. Sci. Comput., 33 (2011), pp. 195–222.

[8] E. Chow and Y. Saad, *Preconditioned Krylov subspace methods for sampling multivariate Gaussian distributions*, SIAM J. Sci. Comput., 2 (2014), pp. A588–A608.

[9] T. A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, 2006.

[10] W. Fong and E. Darve, *The black-box fast multipole method*, J. Comput. Phys., 228 (2009), pp. 8712–8725.

[11] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 1996.

[12] N. Hale, N. J. Higham, and L. N. Trefethen, *Computing A^α, $\log(A)$, and related matrix functions by contour integrals*, SIAM J. Numer. Anal., 46 (2008), pp. 2505–2523.

[13] A. J. Laub, *A Schur method for solving algebraic Riccati equations*, IEEE Transcation on Automatic Control, AC-24 (1979), pp. 913–921.

[14] S. Li, M. Gu, C. Wu, and J. Xia, *New efficient and robust HSS Cholesky factorization of SPD matrices*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 886–904.

[15] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006.

[16] M. L. Stein, J. Chen, and M. Anitescu, *Stochastic approximation of score functions for Gaussian processes*, Annals of Applied Statistics, 7 (2013), pp. 1162–1191.

[17] J. Xia and M. Gu, *Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2899–2920.