SIAM J. SCI. COMPUT.
Vol. 42, No. 2, pp. C43–C68

hm-toolbox: MATLAB SOFTWARE FOR HODLR AND HSS MATRICES*

STEFANO MASSEI†, LEONARDO ROBOL‡, AND DANIEL KRESSNER§

Abstract. Matrices with hierarchical low-rank structure, including HODLR and HSS matrices, constitute a versatile tool to develop fast algorithms for addressing large-scale problems. While existing software packages for such matrices often focus on linear systems, their scope of applications is in fact much wider and includes, for example, matrix functions and eigenvalue problems. In this work, we present a new MATLAB toolbox called hm-toolbox, which encompasses this versatility with a broad set of tools for HODLR and HSS matrices, unmatched by existing software. While mostly based on algorithms that can be found in the literature, our toolbox also contains a few new algorithms as well as novel auxiliary functions. Being entirely based on MATLAB, our implementation does not strive for optimal performance. Nevertheless, it maintains the favorable complexity of hierarchical low-rank matrices and offers, at the same time, a convenient way of prototyping and experimenting with algorithms. A number of applications illustrate the use of the hm-toolbox.

Key words. HODLR matrices, HSS matrices, hierarchical matrices, MATLAB, low-rank approximation

AMS subject classification. 15B99

DOI. 10.1137/19M1288048

1. Introduction.

This work presents hm-toolbox, a new MATLAB software available from https://github.com/numpi/hm-toolbox for working with HODLR (hierarchically off-diagonal low-rank) and HSS (hierarchically semiseparable) matrices. Both formats are defined via a recursive block partition of the matrix. More specifically, for

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \tag{1}$$

it is assumed that the off-diagonal blocks A_{12}, A_{21} have low rank. This partition is repeated recursively for the diagonal blocks until a minimal block size is reached. In the HSS format, the low-rank factors representing the off-diagonal blocks on the different levels of the recursions are nested, while the HODLR format treats all off-diagonal blocks independently. During the last decade, both formats have shown their usefulness in a wide variety of applications. Recent examples include the acceleration of sparse direct linear system solvers [25, 51, 53], large-scale Gaussian process modeling [1, 24], stationary distribution of quasi-birth-death Markov chains [8], as well as fast solvers for (banded) eigenvalue problems [37, 49, 50] and matrix equations [34, 35].

*Submitted to the journal's Software and High-Performance Computing section September 18, 2019; accepted for publication (in revised form) January 28, 2020; published electronically April 8, 2020.

https://doi.org/10.1137/19M1288048

Funding: The work of the first author was supported by the SNSF research project Fast Algorithms from Low-Rank Updates under grant 200020_178806. The work of the second author was partially supported by the GNCS/INdAM project "Metodi di proiezione per equazioni di matrici e sistemi lineari con operatori deniti tramite somme di prodotti di Kronecker, e soluzioni con struttura di rango."

†Applied Mathematics, EPFL Lausanne, Lausanne, CH-1015, Vaud, Switzerland (massei.stef@gmail.com).

‡Department of Mathematics, University of Pisa, Pisa, 56127, Italy (leonardo.robol@unipi.it).

§SB-MATHICSE-ANCHP, EPFL Lausanne, Lausanne, CH-1015, Vaud, Switzerland (daniel.kressner@epfl.ch).

Both the HODLR and HSS formats allow us to design fast algorithms for various linear algebra tasks. Our toolbox offers basic operations (addition, multiplication, inversion), matrix decompositions (Cholesky, LU, QR, ULV), as well as more advanced functionality (matrix functions, solution of matrix equations). It also offers multiple ways of constructing and recompressing these representations as well as converting between HODLR, HSS, and sparse matrices. While most of the toolbox is based on known algorithms from the literature, we also make novel algorithmic contributions. This includes the fast computation of Hadamard products, the matrix product $A^{-1}B$ for HSS matrices A, B, and numerous auxiliary functionalities.

The primary goal of the hm-toolbox is to provide a comprehensive and convenient framework for prototyping algorithms and ensuring reproducibility. Having this goal in mind, our implementation is entirely based on MATLAB and thus does not strive for optimal performance. Still, the favorable complexity of the fast algorithms is preserved.

The HODLR and HSS formats are special cases of hierarchical and $\mathcal{H}^2$ matrices, respectively. The latter two formats allow for significantly more general block partitions, described via cluster trees, which in turn gives the ability to treat a wider range of problems effectively, including two-dimensional (2D) and 3D partial differential equations; see [28] and the references therein. On the other hand, the restriction to partitions of the form (1) comes with a major advantage: it simplifies the design, analysis, and implementation of fast algorithms. Another advantage of (1) is that a low-rank perturbation makes A block diagonal, which opens the door for divide-and-conquer methods; see [35] for an example.

Existing software. In the following, we provide a brief overview of existing software for various flavors of hierarchical low-rank formats. An $n \times n$ matrix S is called *semiseparable* if every submatrix residing entirely in the upper or lower triangular part of S has rank at most one. The class of *quasiseparable* matrices is more general by only considering submatrices in the *strictly* lower and upper triangular parts. The class of sequentially semiseparable matrices is another generalization, which has been defined in [16].

While a fairly complete MATLAB library for semiseparable matrices is available,[1] the public availability of software for quasiseparable matrices seems to be limited to a set of MATLAB functions targeting specific tasks.[2]

Fortran and MATLAB packages for solving linear systems with HSS and sequentially semiseparable matrices are available.[3,4] The Structured Matrix Market[5] provides benchmark examples and supporting functionality for HSS matrices. STRUMPACK [43] is a parallel C++ library for HSS matrices with a focus on randomized compression and the solution of linear systems. HODLRlib [2] is a C++ library for HODLR matrices, which provides shared-memory parallelism through OpenMP and again puts a focus on linear systems. HLib [14] and H2Lib [13] are C libraries which provide a wide range of functionality for hierarchical and $\mathcal{H}^2$ matrices, respectively. HLIBpro [10] and AHMED [4] are C++ libraries implementing optimized algorithms for $\mathcal{H}$-matrices. Pointers to other software packages, related to hierarchical low-rank formats, can be found at https://github.com/gchavez2/awesome_hierarchical_matrices.

[1] https://people.cs.kuleuven.be/~raf.vandebril/homepage/software/sspack.php.
[2] http://people.cs.dm.unipi.it/boito/software.html.
[3] http://scg.ece.ucsb.edu/software.html.
[4] http://www.math.purdue.edu/~xiaj/packages.html.
[5] http://smart.math.purdue.edu/.

Outline. The rest of this work is organized as follows. In section 2, we recall the definitions of HODLR and HSS matrices. Section 3 is concerned with the construction of such matrices in our toolbox and the conversion between different formats. In section 4, we give a brief overview of those arithmetic operations implemented in the hm-toolbox that are based on existing algorithms. More details are provided on two new algorithms and the important recompression operation. Finally, in section 5, we illustrate the use of our toolbox with various examples and applications.

2. Preliminaries and MATLAB classes `hodlr`, `hss`.

2.1. HODLR matrices.

As discussed in the introduction, HODLR matrices are defined via a recursive block partition (1), assuming that the off-diagonal blocks have low rank on every level of the recursion.

The concept of a *cluster tree* allows us to formalize the definition of such a partitioning.

DEFINITION 2.1. *Given $n \in \mathbb{N}$, let $\mathcal{T}_p$ be a completely balanced binary tree of depth p whose nodes are subsets of $\{1, \dots, n\}$. We say that $\mathcal{T}_p$ is a* cluster tree *if it satisfies the following:*

- *The root is $I_1^0 := I = \{1, \dots, n\}$.*
- *The nodes at level ℓ, denoted by $I_1^\ell, \dots, I_{2^\ell}^\ell$, form a partitioning of $\{1, \dots, n\}$ into consecutive indices:*

$$I_i^\ell = \left\{ n_{i-1}^{(\ell)} + 1 \dots, n_i^{(\ell)} - 1, n_i^{(\ell)} \right\}$$

  for some integers $0 = n_0^{(\ell)} \le n_1^{(\ell)} \le \cdots \le n_{2^\ell}^{(\ell)} = n$, $\ell = 0, \dots p$. In particular, if $n_{i-1}^{(\ell)} = n_i^{(\ell)}$, then $I_i^\ell = \emptyset$.
- *The node I_i^ℓ has children $I_{2i-1}^{\ell+1}$ and $I_{2i}^{\ell+1}$ for any $1 \le \ell \le p-1$. The children form a partitioning of their parent.*

In practice, the cluster tree $\mathcal{T}_p$ is often chosen in a balanced fashion, that is, the cardinalities of the index sets on the same level are nearly equal and the depth of the tree is determined by a minimal diagonal block size $n_{\min}$ for stopping the recursion.

In particular, if $n = 2^p n_{\min}$, such a construction yields a perfectly balanced binary tree of depth p; see Figure 1 for $n = 8$ and $n_{\min} = 1$.

The nodes at a level ℓ induce a partitioning of A into a $2^\ell \times 2^\ell$ block matrix, with the blocks given by $A(I_i^\ell, I_j^\ell)$ for $i, j = 1, \dots, 2^\ell$, where we use MATLAB notation for submatrices.

The definition of a HODLR matrix requires that some of the off-diagonal blocks (marked with stripes in Figure 1) have (low) bounded rank.

DEFINITION 2.2. *Let $A \in \mathbb{C}^{n \times n}$ and consider a cluster tree $\mathcal{T}_p$.*

1. *Given $k \in \mathbb{N}$, A is said to be a $(\mathcal{T}_p, k)$-HODLR matrix if every off-diagonal block*

$$A(I_i^\ell, I_j^\ell) \quad \text{such that } I_i^\ell \text{ and } I_j^\ell \text{ are siblings in } \mathcal{T}_p, \quad \ell = 1, \dots, p, \tag{2}$$

   has rank at most k.
2. *The HODLR rank of A (with respect to $\mathcal{T}_p$) is the smallest integer k such that A is a $(\mathcal{T}_p, k)$-HODLR matrix.*

MATLAB class. The hm-toolbox provides the MATLAB class `hodlr` for working with HODLR matrices. The properties of `hodlr` store a matrix recursively in accordance with the partitioning (1) (or, equivalently, the cluster tree) as follows:

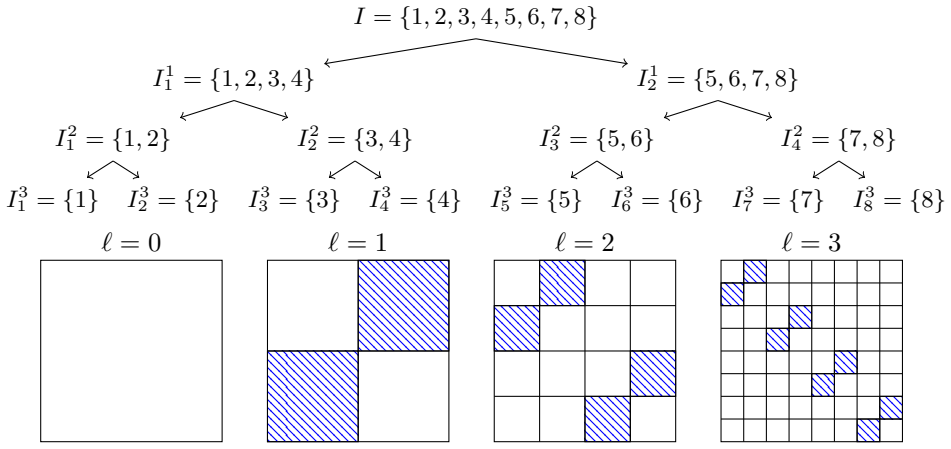

FIG. 1. *Pictures taken from* [35]: *Example of a cluster tree of depth* 3 *and the block partitions induced on each level. The blocks marked with blue stripes are stored as low-rank matrices in the HODLR format.*

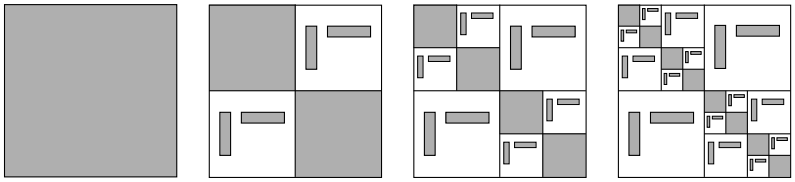

FIG. 2. *Image taken from* [35]: *Illustration of the HODLR format for cluster trees of varying depth. The gray blocks are the (dense) matrices that need to be stored to represent a HODLR matrix.*

- `A11` and `A22` are `hodlr` instances representing the diagonal blocks (for a non-leaf node);
- `U12` and `V12` are the low-rank factors of the off-diagonal block A_{12};
- `U21` and `V21` are the low-rank factors of the off-diagonal block A_{21};
- `F` is either a dense matrix representing the whole matrix (for a leaf node) or empty.

Figure 2 illustrates the storage format. For a matrix of HODLR rank k, the memory consumption reduces from $\mathcal{O}(n^2)$ to $\mathcal{O}(pnk) = \mathcal{O}(kn \log n)$ when using `hodlr`.

2.2. HSS matrices. The $\log(n)$ factor in the memory complexity of HODLR matrices arises from the fact that the low-rank factors take $\mathcal{O}(kn)$ memory on each of the $\mathcal{O}(\log(n))$ levels of the recursion. However, in many—if not most—applications these factors share similarities across different levels, which can be exploited by nested hierarchical low-rank formats, such as the HSS format, to potentially remove the $\log(n)$ factor.

An HSS matrix is associated with a cluster tree $\mathcal{T}_p$; see Definition 2.1. In analogy to HODLR matrices, it is assumed that the off-diagonal blocks can be factorized as

$$A\left(I_i^\ell, I_j^\ell\right) = U_i^{(\ell)} S_{i,j}^{(\ell)} \left(V_j^{(\ell)}\right)^*, \quad S_{i,j}^{(\ell)} \in \mathbb{C}^{k\times k}, \quad U_i^{(\ell)} \in \mathbb{C}^{n_i^{(\ell)}\times k}, \quad V_j^{(\ell)} \in \mathbb{C}^{n_j^{(\ell)}\times k},$$

for all siblings I_i^ℓ, I_j^ℓ in $\mathcal{T}_p$. The matrices $S_{i,j}^{(\ell)}$ are called *core blocks*. Additionally, and in contrast to HODLR matrices, for HSS matrices we require the factors $U_i^{(\ell)}, V_j^{(\ell)}$ to

be nested across different levels of $\mathcal{T}_p$. More specifically, it is assumed that there exist so-called translation operators $R_{U,i}^{(\ell)}, R_{V,j}^{(\ell)} \in \mathbb{C}^{2k\times k}$ such that

$$U_i^{(\ell)} = \begin{bmatrix} U_{i_1}^{(\ell+1)} & 0 \\ 0 & U_{i_2}^{(\ell+1)} \end{bmatrix} R_{U,i}^{(\ell)}, \qquad V_j^{(\ell)} = \begin{bmatrix} V_{j_1}^{(\ell+1)} & 0 \\ 0 & V_{j_2}^{(\ell+1)} \end{bmatrix} R_{V,j}^{(\ell)}, \tag{3}$$

where $I_{i_1}^{\ell+1}, I_{i_2}^{\ell+1}$ and $I_{j_1}^{\ell+1}, I_{j_2}^{\ell+1}$ denote the children of I_i^ℓ and I_j^ℓ, respectively. These relations allow us to retrieve the low-rank factors $U_i^{(\ell)}$ and $V_i^{(\ell)}$ for the higher levels $\ell = 1,\dots,p-1$ recursively from the bases $U_i^{(p)}$ and $V_i^{(p)}$ at the deepest level p. Therefore, in order to represent A, one only needs to store the diagonal blocks $D_i := A(I_i^p, I_i^p)$, the bases $U_i^{(p)}$, $V_i^{(p)}$, the core factors $S_{i,j}^{(\ell)}$, $S_{j,i}^{(\ell)}$, and the translation operators $R_{U,i}^{(\ell)}$, $R_{V,i}^{(\ell)}$. In particular, note that only the bases on the lowest level, $U_i^{(p)}$ and $V_i^{(p)}$, are stored. We remark that, for simplifying the exposition, we have considered translation operators and bases $U_i^{(p)}, V_j^{(p)}$ with k columns for every level and node. This is not necessary, as long as the dimensions are compatible, and this more general framework is handled in hm-toolbox.

As explained in [29], a matrix A admits the decomposition explained above if and only if it is an HSS matrix in the sense of the following definition, which imposes rank conditions on certain block rows and columns without their diagonal blocks; see Figure 3 for an illustration.

DEFINITION 2.3. *Let $A \in \mathbb{C}^{n\times n}$, $I = \{1,\dots,n\}$, and consider a cluster tree $\mathcal{T}_p$.*

(a) *$A(I_i^\ell, I \setminus I_i^\ell)$ is called an HSS block row and $A(I \setminus I_i^\ell, I_i^\ell)$ is called an HSS block column for $i = 1,\dots,2^\ell$, $\ell = 1,\dots,p$.*

(b) *For $k \in \mathbb{N}$, A is called a $(\mathcal{T}_p, k)$-HSS matrix if every HSS block row and column of A has rank at most k.*

(c) *The HSS rank of A (with respect to $\mathcal{T}_p$) is the smallest integer k such that A is a $(\mathcal{T}_p, k)$-HSS matrix.*

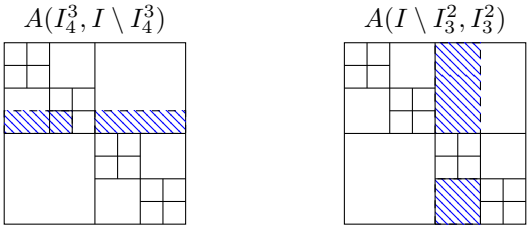

FIG. 3. *Image taken from* [35]: *Illustration of an HSS block row and an HSS block column for a cluster tree of depth* 3.

MATLAB class. The `hss` class provided by the hm-toolbox uses the following properties to represent an HSS matrix recursively:

- `A11` and `A22` are `hss` instances representing the diagonal blocks (for a nonleaf node);
- `U` and `V` contain the basis matrices $U_i^{(p)}$ and $V_i^{(p)}$ for a leaf node and are empty otherwise;
- `Rl` and `Rr` are such that `[Rl; Rr]` is the translation operator $R_{U,i}^{(\ell)}$, and `Wl` and `Wr` are such that `[Wl; Wr]` is the translation operator $W_{U,i}^{(\ell)}$ (note that `Rl`, `Rr`, `Wl`, `Wr` are empty for the top node or a leaf node);
- `B12`, `B21` contain the matrices $S_{i,j}^{(\ell)}$, $S_{j,i}^{(\ell)}$ for a nonleaf node;

- `D` is either a dense matrix representing the whole matrix (for a leaf node) or empty.

Using the `hss` class, $\mathcal{O}(nk)$ memory is needed to represent a matrix of HSS rank k.

2.3. Appearance of HODLR and HSS matrices. Our toolbox is most effective for matrices of small HODLR or HSS rank. In some cases, this property is evident, e.g., for matrices with particular sparsity patterns such as banded matrices. However, there are numerous situations of interest in which the matrix is dense but still admits a highly accurate approximation by a matrix of small HODLR or HSS rank. In particular, this is the case for the discretization of (kernel) functions and integral operators under certain regularity conditions; see [12, 28, 29, 40] for examples.

When manipulating HODLR and HSS matrices, using the functionality of the toolbox, it would be desirable that the off-diagonal low-rank structure is (approximately) preserved. For more restrictive formats, such as semi- and quasiseparable matrices, the low-rank structure is preserved exactly by certain matrix factorizations and inversion; see the monographs [19, 20, 47, 48]. While the HSS rank is also preserved by inversion, the same does not hold for the HODLR rank. Often, additional properties are needed in order to show that the HODLR and HSS formats are (approximately) preserved under arithmetic operations; see [5, 21, 22, 27, 35, 41].

3. Construction of HODLR/HSS representation. Even when it is known that a given matrix can be represented or accurately approximated in the HODLR or HSS formats, it is by no means a trival task to construct such structured representations efficiently. Often, the construction needs to be tailored to the problem at hand, especially if one aims at handling large-scale matrices and thus needs to bypass the $\mathcal{O}(n^2)$ memory needed for the explicit dense representation of the matrix. The hm-toolbox provides several constructors (summarized in Table 1 below) trying to capture the most typical situations for which the HODLR and HSS formats are utilized. The constructors and, more generally, the hm-toolbox support both real and complex valued matrices.

3.1. Parameter settings for constructors. The output of the constructors depends on a number of parameters. In particular, the truncation tolerance ϵ, which guides the error in the spectral norm when approximating a given matrix by a HODLR/HSS matrix, can be set with the following commands:

TABLE 1

Complexities of constructors. The symbol $\mathcal{C}_1$ denotes the complexity of computing a single entry of the matrix through the handle function. The symbol $\mathcal{C}_2$ indicates the cost of the matrix-vector multiplication by A or A^*.

Constructor	HODLR complexity	HSS complexity
Dense	$\mathcal{O}(kn^2)$	$\mathcal{O}(kn^2)$
Sparse	$\mathcal{O}(k^2 n \log(n) + kn_z)$	$\mathcal{O}(k^2 n + kn_z)$
`'banded'`	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn)$
`'cauchy'`	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn \log n)$
`'diagonal'`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`'eye'`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`'handle'`	$\mathcal{O}(\mathcal{C}_1 kn \log n)$	$\mathcal{O}(\mathcal{C}_1 n + \mathcal{C}_2 k)$
`'low-rank'`	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn)$
`'ones'`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
`'toeplitz'`	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn \log n)$
`'zeros'`	$\mathcal{O}(n)$	$\mathcal{O}(n)$

```
hodlroption('threshold', ϵ)
hssoption('threshold', ϵ)
```

The default setting is $\epsilon = 10^{-12}$ for both formats.

When approximating with a HODLR matrix, the rank of each off-diagonal block $A(I_i^p, I_j^p)$ is chosen such that the spectral norm of the approximation error is bounded by ϵ times $\|A(I_i^p, I_j^p)\|_2$ or an estimate thereof. For example, when using the truncated singular value decomposition (SVD) for low-rank truncation, this means that k is determined by the number of singular values larger than $\epsilon\|A(I_i^p, I_j^p)\|_2$; see, e.g., [26]. Ensuring such a (local) truncation error guarantees that the overall approximation for the whole matrix A is bounded by $\mathcal{O}(\epsilon \log(n)\|A\|_2)$ in the spectral norm; see [28, Lemma 6.3.2] and [9, Theorem 2.2].

When approximating with an HSS matrix, the tolerance ϵ guides the approximation error when compressing HSS block rows and columns. The interplay between local and global approximation errors is more subtle and depends on the specific procedure. In general, the global approximation error stays proportional to ϵ. Specific results for the Frobenius and spectral norms can be found in [52, Corollary 4.3] and [35, Theorem 4.7], respectively. See also [12, Theorem 5.30, Corollary 5.31] for results on the more general class of $\mathcal{H}^2$-matrices.

By default, our constructors determine the cluster tree $\mathcal{T}_p$ by splitting the row and column index sets as equally as possible until a minimal block size $n_{\min}$ is reached. More specifically, an index set $\{1, \dots, n\}$ is split into $\{1, \dots, \lceil \frac{n}{2} \rceil\} \cup \{\lceil \frac{n}{2} \rceil + 1, \dots, n\}$. The default value for $n_{\min}$ is 256; this value can be adjusted by calling `hodlroption('block-size', nmin)` and `hssoption('block-size', nmin)`. In section 3.7 below, we explain how nonstandard cluster trees can be specified.

3.2. Construction from dense or sparse matrices. The HODLR/HSS approximation of a given dense or sparse matrix $A \in \mathbb{C}^{n\times n}$ is obtained via

```
hodlrA = hodlr(A);
hssA = hss(A);
```

In the following, we discuss the algorithms behind these two commands.

`hodlr` *for dense* A. To obtain a HODLR approximation, the Householder QR decomposition with column pivoting [15] is applied to each off-diagonal block. The algorithm is terminated when an upper bound for the spectral norm of the remainder is below ϵ times the maximum pivot element. Although there are examples for which such a procedure severely overestimates the (numerical) rank [26, section 5.4.3], this rarely happens in practice. If k denotes the HODLR rank of the output, this procedure has complexity $\mathcal{O}(kn^2)$. Optionally, the truncated SVD mentioned above instead of QR with pivoting can be used for compression. The following commands are used to switch between both methods:

```
hodlroption('compression', 'svd');
hodlroption('compression', 'qr');
```

`hodlr` *for sparse* A. The two-sided Lanczos method [45], which only requires matrix-vector multiplications with an off-diagonal block and its (Hermitian) transpose, combined with recompression [3] is applied to each off-diagonal block. The method uses the heuristic stopping criterion described in [3, p. 173] with threshold ϵ times an estimate of the spectral norm of the block under consideration. Letting k again denote the HODLR rank of the output and assuming that Lanczos converges

in $\mathcal{O}(k)$ steps, this procedure has complexity $\mathcal{O}(k^2 n \log(n) + k n_z)$, where n_z denotes the number of nonzero entries of A.

`hss` *for dense* A. The algorithm described in [54, Algorithm 1] is used, which essentially applies low-rank truncation to every HSS block row and column starting from the leaves to the root of the cluster tree and ensuring the nestedness of the factors (3). As for `hodlr`, one can choose between QR with column pivoting or the SVD (default) for low-rank truncation via `hssoption`.

Letting k denote the HSS rank of the output, the complexity of this procedure is $\mathcal{O}(kn^2)$.

`hss` *for sparse* A. The algorithm described in [39] is used, which is based on the randomized SVD [30] and involves matrix-vector products with the entire matrix A and its (Hermitian) transpose. We use 10 random vectors for deciding whether to terminate the randomized SVD, which ensures an accuracy of $\mathcal{O}(\epsilon)$ with probability at least $1 - 6 \cdot 10^{-10}$ [39, section 2.3]. Assuming that $\mathcal{O}(k)$ random vectors are needed in total, the complexity of this procedure is $\mathcal{O}(k^2 n + k n_z)$.

3.3. Construction from handle functions. We provide constructors that access A indirectly via handle functions.

For HODLR, given a handle function `@(I, J) Aeval(I,J)` that provides the submatrix $A(I, J)$ given row and column indices I, J, the command

```
hodlrA = hodlr('handle', Aeval, n, n);
```

returns a HODLR approximaton of A. We apply *adaptive cross approximation* (ACA) with partial pivoting [11, Algorithm 1] to approximate each off-diagonal block. The global tolerance ϵ is used as a threshold for the (heuristic) stopping criterion of ACA.

The HSS constructor uses two additional handle functions `@(v) Afun(v)` and `@(v) Afunt(v)` for matrix-vector produces with A and A^*, respectively. The command

```
hssA = hss('handle', Afun, Afunt, Aeval, n, n);
```

returns an HSS approximation using the algorithm for sparse matrices discussed in section 3.2.

3.4. Construction from structured matrices. When A is endowed with a structure that allows its description with a small number of parameters, it is sometimes possible to efficiently obtain a HODLR/HSS approximation. All such constructors provided in hm-toolbox have the syntax

```
hodlrA = hodlr(structure, ...);
hssA = hss(structure, ...);
```

where `structure` is a string describing the properties of A. The following options are provided:

`'banded'` Given a banded matrix (represented as a sparse matrix on input) with lower and upper bandwidth b_l and b_u, this constructor returns an exact $(\mathcal{T}_p, \max\{b_u, b_l\})$-HODLR or $(\mathcal{T}_p, b_l + b_u)$-HSS representation of the matrix. For instance,

```
h = 1/(n - 1);
A = spdiags(ones(n,1) * [1 -2 1], -1:1, n,
    n) / h^2;
hodlrA = hodlr('banded', A);
```

returns a representation of the 1D discrete Laplacian; see also (8) below.

`'cauchy'` Given two vectors x and y representing a Cauchy matrix A with entries $a_{ij} = \frac{1}{x_i + y_j}$, the commands

```
hodlrA = hodlr('cauchy', x, y);
hssA = hss('cauchy', x, y);
```

return a HODLR/HSS approximation of A. For the HODLR format, the construction relies on the `'handle'` constructor described above. The HSS representation is obtained by first performing a HODLR approximation and then converting to the HSS format; see section 3.5.

`'diagonal'` Given the diagonal v of a diagonal matrix, the commands

```
hodlrA = hodlr('diagonal', v);
hssA = hss('diagonal', v);
```

return an exact representation with HODLR/HSS ranks equal to 0.

`'eye'` Given n, this constructs a HODLR/HSS representations of the $n \times n$ identity matrix.

`'low-rank'` Given $A = UV^*$ in terms of its (low-rank) factors U, V with k columns, this returns an exact $(\mathcal{T}_p, k)$-HODLR or $(\mathcal{T}_p, k)$-HSS representation.

`'ones'` This constructs a HODLR/HSS representation of the matrix of all ones. As this is a rank-one matrix, this represents a special case of `'low-rank'`.

`'toeplitz'` Given the first column c and the first row r of a Toeplitz matrix A, the following lines construct HODLR and HSS approximations of A:

```
hodlrA = hodlr('toeplitz', c, r);
hssA = hss('toeplitz', c, r);
```

For a Toeplitz matrix, the off-diagonal blocks A_{12}, A_{21} on the first level in the cluster tree already contain most of the required information. Indeed, all off-diagonal blocks are submatrices of these two. To obtain a HODLR approximation we first construct low-rank approximations of A_{12}, A_{21} using the two-sided Lanczos algorithm discussed above, combined with FFT-based fast matrix-vector multiplication. For all deeper levels, low-rank factors of the off-diagonal blocks are simply obtained by restriction. This constructor is used in section 5.3 to discretize fractional differential operators. For obtaining an HSS approximation, we rely on the `'handle'` constructor.

`'zeros'` This constructs a HODLR/HSS representations of the zero matrix.

3.5. Conversion between formats. The hm-toolbox functions `hodlr2hss` and `hss2hodlr` convert between the HODLR and HSS formats. An HSS matrix is converted into a HODLR matrix by simply building explicit low-rank factorizations of the off-diagonal blocks from their implicit nested representation in the HSS format. This is done recursively by using the translation operators and the core blocks $S_{i,j}^{(\ell)}$ with a cost of $\mathcal{O}(kn \log n)$ operations. A HODLR matrix is converted into an HSS matrix by first incorporating the (dense) diagonal blocks and then performing a sequence of low-rank updates in order to add the off-diagonal blocks that appear on each level. In order to keep the HSS rank as low as possible, recompression is performed after each sum; see also section 4.5 below. The whole procedure has a cost of $\mathcal{O}(k^2 n \log n)$, where k is the HSS rank of the argument.

HODLR and HSS matrices are converted into dense matrices using the `full` function. In analogy to the sparse format in MATLAB, an arithmetic operation

TABLE 2
Format of the outcome of a matrix-matrix operation op $\in \{+, -, *, \backslash, /\}$ *depending on the structure of the two inputs.*

op	HSS	HODLR	Dense
HSS	HSS	HODLR	Dense
HODLR	HODLR	HODLR	Dense
Dense	Dense	Dense	Dense

between different types of structure always results in the "less structured" format. As we consider HSS to be the more structured format compared to HODLR, this induces the hierarchy reported in Table 2.

Some matrices, like inverses of banded matrices, are HODLR and approximately sparse at the same time. In such situations, it can be of interest to convert a HODLR matrix into a sparse matrix by neglecting entries below a certain tolerance. The overloaded function `sparse` effects this conversion efficiently by only considering those off-diagonal entries for which the corresponding rows of the low-rank factors are sufficiently large. In the following example, entries below 10^{-8} are neglected.

```
n = 2^(14);
A = spdiags( ones(n, 1) * [1 3 -1], -1:1, n, n);
hodlrA = hodlr(A); hodlrA = inv(hodlrA);
spA = sparse(hodlrA, 1e-8);
fprintf('Bandwidth: %d, Error = %e\n',...
bandwidth(spA), normest(spA * A - speye(n), 1e-4));
Bandwidth: 14, Error = 3.131282e-08
```

For an HSS matrix, the `sparse` function proceeds indirectly via first converting to the HODLR format by means of `hss2hodlr`.

In summary, the described functionality allows us to switch back and forth between HODLR, HSS, and sparse formats.

3.6. Auxiliary functionality. The hm-toolbox contains several functions that make it convenient to work with HODLR and HSS matrices. For example, the MATLAB functions `diag`, `imag`, `real`, `trace`, `tril`, `triu` have been overloaded to compute the corresponding quantities for HODLR/HSS matrices. We also provide the command `spy` to inspect the structure of an `hodlr` or `hss` instance by plotting the ranks of off-diagonal blocks in the given partitioning. Two examples for the output of `spy(A)` will be given in Figure 5.

3.7. Nonstandard cluster trees. A cluster tree $\mathcal{T}_p$ is determined by the partitioning of the index set on the deepest level and can thus be represented by the vector $c := [n_1^{(p)}, \ldots, n_{2^p}^{(p)}]$; see Definition 2.1. For example, the cluster tree in Figure 1 is represented by $c = [1, 2, \ldots, 8]^T$.

Note that it is possible to construct cluster trees for which the index sets are not equally partitioned on one level. In fact, some index sets can be empty. For instance, the cluster tree in Figure 4 is represented by the vector $c = [2, 4, 8, 8]^T$.

The vector c is used inside the hm-toolbox to specify a cluster tree. For all constructors discussed above, an optional argument can be provided to specify the cluster tree for the rows and columns. For those constructors that also allow for rectangular matrices (see below), different cluster trees can be specified for the rows and columns. For example, the partitioning of Figure 4 can be imposed on an 8×8 matrix A as follows:

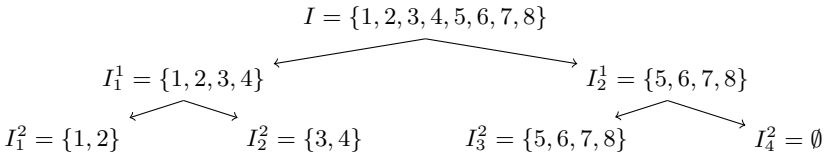

FIG. 4. *Example of a cluster tree with a leaf node containing an empty index set.*

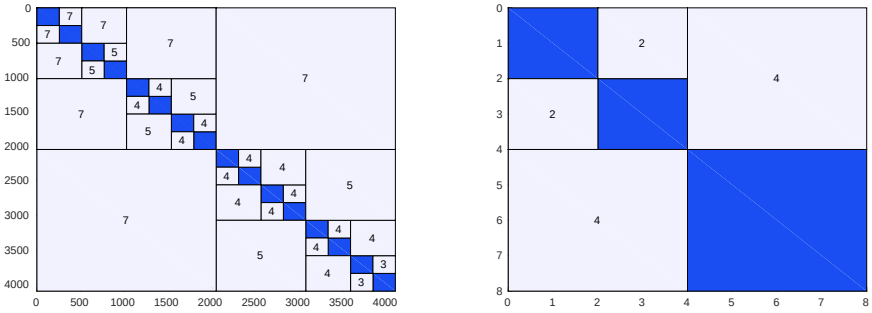

FIG. 5. *Output of the command* `spy` *for a Cauchy matrix (left) and for a matrix with a non-standard cluster tree (right).*

```
c = [2 4 8 8];
hodlrA = hodlr(A, 'cluster', c);
spy(hodlrA);
```

The output of `spy` for such a matrix is reported in Figure 5.

The vectors describing the row and column clusters of a given HODLR/HSS matrix can be retrieved using the `cluster` command.

3.8. Rectangular matrices. The hm-toolbox also allows us to create rectangular HODLR/HSS matrices by means of the dense/sparse constructors or one of the following arguments for the constructor: `'cauchy'`, `'handle'`, `'low-rank'`, `'ones'`, `'toeplitz'`,`'zeros'`. This requires building two cluster trees, one for the row indices and one for the column indices. If these clusters are not specified, they are built in the default way discussed in section 3.2, such that the children of each node have nearly equal cardinality. The procedure is carried out simultaneously for the row and column cluster trees and it stops when either both index sets are smaller than the minimal block size or one of the two reaches cardinality 1. In particular, this ensures that the returned row and column cluster trees have the same depth.

We remark that some operations for a HODLR/HSS matrix are available only when the row and column cluster trees are equal (which in particular implies that the matrix is square), such as the solution of linear systems, matrix powers, and the determinant.

4. Arithmetic operations. The HODLR and HSS formats allow us to carry out several arithmetic operations efficiently, a fact that greatly contributes to the versatility of these formats in applications. In this section, we first illustrate the design of fast operations for matrix-vector products and then give an overview of the operations provided in the hm-toolbox, many of which have already been described in the literature. However, we also provide a few operations that are new to the best of our knowledge. In particular, this holds for the algorithms for computing $A^{-1}B$ in

the HSS format and the Hadamard product in both formats described in sections 4.3 and 4.4, respectively. As arithmetic operations often increase the HODLR/HSS ranks, it is important to combine them with recompression, a matter discussed in section 4.5.

4.1. Matrix-vector products. The block partitioning (1) suggests the use of a recursive algorithm for computing the matrix-vector product Av. Partitioning v in accordance with the columns of A, one obtains

$$Av = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} A_{11}v_1 + A_{12}v_2 \\ A_{21}v_1 + A_{22}v_2 \end{bmatrix}.$$

In turn, this reduces Av to smaller matrix-vector products involving low-rank off-diagonal blocks and diagonal blocks. If A is HODLR, the diagonal and off-diagonal blocks are not coupled and $A_{11}v_1, A_{22}v_2$ are simply computed by recursion. The resulting procedure has complexity $\mathcal{O}(kn \log n)$; see Figure 6 (left).

If A is HSS, then the off-diagonal blocks A_{12}, A_{21} are not directly available, unless the recursion has reached a leaf. To address this issue, the following four-step procedure is used; see, e.g., [17, section 3]. In step 1, the (column) cluster tree is traversed from bottom to top in order to multiply the right-factor matrices $\left(V_j^{(\ell)}\right)^*$ with the corresponding portions of v via the recursive representation (3). More specifically, letting $v(I_i^p)$ denote the restriction of v to a leaf I_i^p, we first compute $v_i^p := (V_i^{(p)})^* v(I_i^p)$ on the deepest level and then retrieve all quantities $v_i^\ell := (V_i^{(\ell)})^* v(I_i^\ell)$ for $\ell = p-1, \dots, 1$ by applying the translation operators $R_{V,i}^{(\ell)}$ in a bottom-up fashion. In step 2, all core blocks $S_{i,j}^\ell$ are applied. In step 3—analogous to step 1—the (row) cluster tree is traversed from top to bottom in order to multiply the left-factor matrices $U_i^{(\ell)}$ with the corresponding portions of v via the recursive representation (3). In step 4, the contributions from the diagonal blocks are added to the vectors obtained at the end of step 2. The resulting procedure has complexity $\mathcal{O}(kn)$; see Figure 6 (right).

procedure HODLR_MATVEC(A, v)
  if A is dense **then**
    return Av
  end if
  $y_1 \leftarrow$ HODLR_MATVEC(A_{11}, v_1)
  $y_2 \leftarrow A_{12}v_2$
  $y_3 \leftarrow A_{21}v_1$
  $y_4 \leftarrow$ HODLR_MATVEC(A_{22}, v_2)
  return $\begin{bmatrix} y_1 + y_2 \\ y_3 + y_4 \end{bmatrix}$
end procedure

procedure HSS_MATVEC(A, v)
  On level $\ell = p$ compute
    $v_i^p \leftarrow (V_i^{(p)})^* v(I_i^p), \quad i = 1, \dots, 2^p$
    $d_i^p \leftarrow A(I_i^p, I_i^p) v(I_i^p)$
  for $\ell = p-1, \dots, 1$, $i = 1, \dots, 2^\ell$ **do**
    $v_i^\ell \leftarrow (R_{V,i}^{(\ell)})^* \begin{bmatrix} v_{2i-1}^{\ell+1} \\ v_{2i}^{\ell+1} \end{bmatrix}$
  end for
  for $\ell = 1, \dots, p-1$, $i = 1, \dots, 2^\ell$ **do**
    $\begin{bmatrix} v_{2i-1}^\ell \\ v_{2i}^\ell \end{bmatrix} \leftarrow \begin{bmatrix} 0 & S_{2i-1,2i}^{(\ell)} \\ S_{2i,2i-1}^{(\ell)} & 0 \end{bmatrix} \begin{bmatrix} v_{2i-1}^\ell \\ v_{2i}^\ell \end{bmatrix}$
  end for
  for $\ell = 1, \dots, p-1$, $i = 1, \dots, 2^\ell$ **do**
    $\begin{bmatrix} v_{2i-1}^{\ell+1} \\ v_{2i}^{\ell+1} \end{bmatrix} \leftarrow \begin{bmatrix} v_{2i-1}^{\ell+1} \\ v_{2i}^{\ell+1} \end{bmatrix} + R_{U,i}^{(\ell)} v_i^\ell$
  end for
  On level $\ell = p$ compute
    $y(I_i^p) \leftarrow U_i^{(p)} v_i^p + d_i^p, \quad i = 1, \dots, 2^p$
  return y
end procedure

FIG. 6. *Pseudocodes of HODLR matrix-vector product (on the left) and HSS matrix-vector product (on the right).*

TABLE 3
Complexity of arithmetic operations in the hm-toolbox; A, B are $n \times n$ matrices with HODLR/HSS rank k and v is a vector of length n.

Operation	HODLR complexity	HSS complexity
`A*v`	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn)$
`A\v`	$\mathcal{O}(k^2 n \log^2 n)$	$\mathcal{O}(k^2 n)$
`A+B`	$\mathcal{O}(k^2 n \log n)$	$\mathcal{O}(k^2 n)$
`A*B`	$\mathcal{O}(k^2 n \log^2 n)$	$\mathcal{O}(k^2 n)$
`A\B`	$\mathcal{O}(k^2 n \log^2 n)$	$\mathcal{O}(k^2 n)$
`inv(A)`	$\mathcal{O}(k^2 n \log^2 n)$	$\mathcal{O}(k^2 n)$
`A.*B` [6]	$\mathcal{O}(k^4 n \log n)$	$\mathcal{O}(k^4 n)$
`lu(A), chol(A)`	$\mathcal{O}(k^2 n \log^2 n)$	—
`ulv(A), chol(A)`	—	$\mathcal{O}(k^2 n)$
`qr(A)`	$\mathcal{O}(k^2 n \log^2 n)$	—
compression	$\mathcal{O}(k^2 n \log(n))$	$\mathcal{O}(k^2 n)$

4.2. Overview of fast arithmetic operations in the hm-toolbox. The fast algorithms for performing matrix-matrix operations and matrix factorizations and solving linear systems are based on extensions of the recursive paradigms discussed above for the matrix-vector product. In the HODLR format the original task is split into subproblems that are solved either recursively or relying on low-rank matrix arithmetic; see, e.g., [28, Chapter 3] for an overview and [38] for the QR decomposition. In the HSS format, the algorithms have a tree-based structure and a bottom-to-top-to-bottom data flow; see [44, 54]. The HSS solver for linear systems is based on an implicit ULV factorization of the coefficient matrix [17]. A list of the matrix operations available in the toolbox, with the corresponding complexities, is given in Table 3. In the latter, we assume the HODLR/HSS ranks of the matrix arguments to be bounded by k. Moreover, for the matrix-matrix multiplication and factorization of HODLR matrices, repeated recompression is needed to limit rank growth of intermediate quantities and we assume that these ranks stay $\mathcal{O}(k)$. We refer to [18] for an alternative approach for matrix-matrix multiplication based on the randomized SVD.

4.3. $A^{-1}B$ in the HSS format. Matrix iterations for solving matrix equations or computing matrix functions [32] sometimes involve the computation of $A^{-1}B$ for square matrices A, B. Being able to perform this operation in HODLR/HSS arithmetic in turn gives the ability to address large-scale structured matrix equations/functions; see [8] for an example.

For HODLR matrices A, B, the operation $A^{-1}B$ can be implemented in a relatively simple manner, by first computing an LU factorization of A and then applying the factors to B; see [28]. For HSS matrices A, B, this operation is more delicate and in the following we describe an algorithm based on the ideas behind the fast ULV solvers from [16, 17].

Our algorithm for computing $A^{-1}B$ performs the following four steps:

1. The HSS matrix A is sparsified as $\widetilde{A} = Q^* A Z$ by means of orthogonal transformation Q acting on the row generators at level p, and Z triangularizing the diagonal blocks. B is updated accordingly by left multiplying it with Q^*.
2. The sparsified matrix is decomposed as a product $\widetilde{A} = A_1 \cdot A_2$ such that A_1^{-1} is easy to apply to B; the matrix A_2 is, up to permutation, of the form $I \oplus \hat{A}_2$,

[6] The complexity of the Hadamard product is dominated by the recompression stage due to the k^2 HODLR/HSS rank of $A \circ B$. Without recompression the cost is $\mathcal{O}(k^2 n \log n)$ for HODLR and $\mathcal{O}(k^2 n)$ for HSS.

where $\hat{A}_2$ is again HSS with the same tree of A, but with smaller blocks.

3. The leaf nodes of $\hat{A}_2$ are merged, yielding an HSS matrix with $p-1$ levels. The procedure is recursively applied for applying $\hat{A}_2^{-1}$ to the corresponding rows and columns of $A_1^{-1}Q^*B$.
4. Finally, $A^{-1}B$ is recovered by applying the orthogonal transformation Z from the left to $A_2^{-1}A_1^{-1}Q^*B$.

We now discuss the four steps in detail. To simplify the description, we assume that all involved ranks are equal to k,

Step 1. For each left basis $U_i^{(p)}$ of the HSS matrix A, we compute a QL factorization $U_i^{(p)} = Q_i\hat{U}_i^{(p)}$ with a square unitary matrix Q_i such that

$$\hat{U}_i^{(p)} = Q_i^* U_i^{(p)} = \begin{bmatrix} 0 \\ \widetilde{U}_i^{(p)} \end{bmatrix}, \qquad \widetilde{U}_i^{(p)} \in \mathbb{C}^{k\times k}.$$

We define $Q = Q_1 \oplus \cdots \oplus Q_{2^p}$ and, in turn, the matrix Q^*A takes the shape displayed in the left plot of Figure 7, where $\widetilde{D}_i := Q_i^* D_i$, and D_i are the diagonal blocks of A. Similarly, we consider an orthogonal transformation $Z = Z_1 \oplus \cdots \oplus Z_{2^p}$ such that each $Q_i^* D_i Z_i$ has the form

$$Q_i^* D_i Z_i = \begin{bmatrix} \widetilde{D}_{i,11} & 0 \\ \widetilde{D}_{i,21} & \widetilde{D}_{i,22} \end{bmatrix}, \quad \widetilde{D}_{i,11} \text{ lower triangular and } \widetilde{D}_{i,22} \in \mathbb{C}^{k\times k}.$$

Then $\widetilde{A} := Q^*AZ$ has the sparsity pattern displayed in the right plot of Figure 7.

Step 2. The matrix $\widetilde{A}$ is decomposed into a product $\widetilde{A} = A_1 \cdot A_2$ as follows. For each block column of $\widetilde{A}$ on the lowest level of recursion we partition $\widetilde{A}(:, I_j^p) =: \big[C_1, C_2\big]$ such that C_2 has k columns. The corresponding block column of the identity matrix is partitioned analogously: $I(:, I_j^p) =: \big[E_1, E_2\big]$. Now, the matrices A_1, A_2 are built by setting

$$A_1(:, I_j^p) := \big[C_1, E_2\big], \quad A_2(:, I_j^p) := \big[E_1, C_2\big].$$

The resulting sparsity patterns of these factors are displayed in Figure 8.

Letting $\left[\begin{smallmatrix} \widetilde{D}_{i,11} & 0 \\ \widetilde{D}_{i,21} & I_k \end{smallmatrix}\right]$ denote a diagonal block of A_1, we construct the block diagonal matrix $A_{1,D}$ with the diagonal blocks $\widetilde{D}_{i,11} \oplus I_k$ for $i = 1, \ldots, 2^p$. We decompose

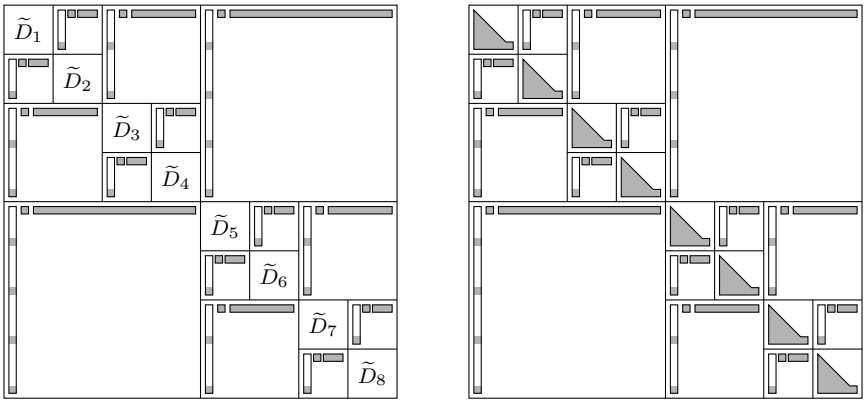

FIG. 7. *Sparsity patterns of the transformations of A during step* 1.

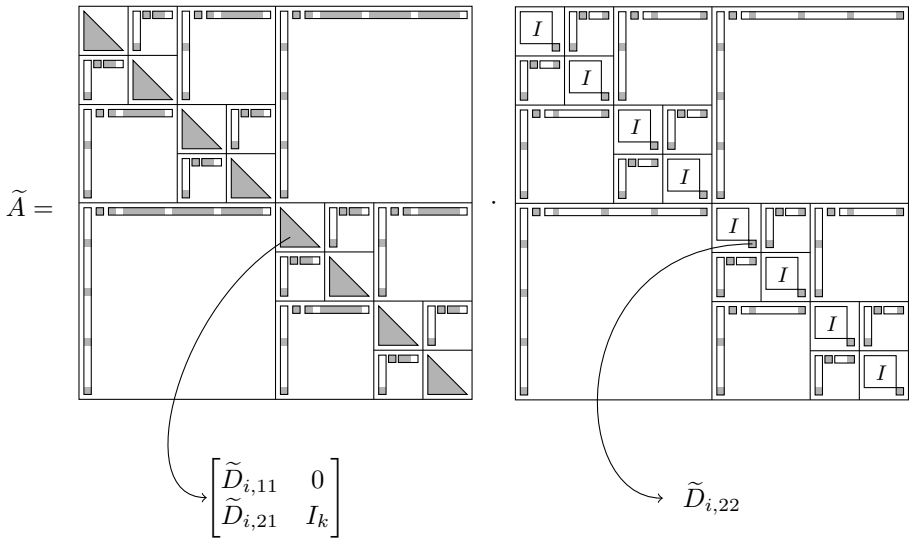

FIG. 8. *Sparsity patterns of the factors A_1, A_2 constructed in step* 2.

$A_1 = A_{1,D} + U_A V_A^T$, where $U_A V_A^T$ is a low-rank factorization of the off-diagonal part of A_1. In particular, the factors U_A, V_A have $2^p k$ columns and, thanks to their sparsity pattern (see the left matrix in Figure 8), they satisfy the relations $V_A^T U_A = 0$ and $A_{1,D} U_A = A_{1,D}^{-1} U_A = U_A$. In turn, by the Woodbury matrix identity, we obtain

$$A_1^{-1} = (A_{1,D} + U_A V_A^T)^{-1} = (I - U_A V_A^T) A_{1,D}^{-1}.$$

Therefore, computing $A_1^{-1} Q^* B$ comes down to applying the block diagonal matrix $A_{1,D}^{-1}$, followed by a correction which involves the multiplication with the matrix $U_A V_A^T$ which is $(\mathcal{T}_p, k)$-HSS.

Step 3. To apply A_2^{-1} to $A_1^{-1} Q^* B$, we follow the strategy of the fast implicit ULV solver for linear systems presented in [16, section 4.2.3]. After a suitable permutation, A_2 has the form $I \oplus \hat{A}_2$, where $\hat{A}_2$ is a $2^p k \times 2^p k$ HSS matrix (of level p) assembled by selecting the indices corresponding to the trailing $k \times k$ minors of the diagonal blocks. As a principal submatrix, the HSS structure of $\hat{A}_2$ is directly inherited from the one of $\widetilde{A}$ at no cost. Then we call the whole procedure recursively to apply $\hat{A}_2^{-1}$ to the corresponding rows in $A_1^{-1} Q^* B$, which are viewed as a (rectangular) HSS matrix of depth $p - 1$.

Step 4. To conclude, we apply the block diagonal orthogonal transformation Z, arising from step 1, to $A_2^{-1} A_1^{-1} Q^* B$.

4.4. Hadamard product in the HODLR and in the HSS format. To carry out the Hadamard (or elementwise) product $A \circ B$ of two HODLR/HSS matrices A, B with the same cluster trees, it is useful to recall the Hadamard product of two low-rank matrices. More specifically, given $U_1 B_1 V_1^*$ and $U_2 B_2 V_2^*$ we have that (see Lemma 3.1 in [36])

$$U_1 B_1 V_1^* \circ U_2 B_2 V_2^* = (U_1 \odot^T U_2)(B_1 \otimes B_2)(V_1 \odot^T V_2)^*, \tag{4}$$

where $\otimes$ denotes the Kronecker product and $\odot^T$ is the *transpose Khatri–Rao product* defined as

procedure HODLR_HADAM(A, B)
  if A, B are dense **then**
    return $A \circ B$
  end if
  $C_{11} \leftarrow$ HODLR_HADAM(A_{11}, B_{11})
  $C_{22} \leftarrow$ HODLR_HADAM(A_{22}, B_{22})
  $C.U_{12} \leftarrow A.U_{12} \odot^T B.U_{12}$
  $C.V_{12} \leftarrow A.V_{12} \odot^T B.V_{12}$
  $C.U_{21} \leftarrow A.U_{21} \odot^T B.U_{21}$
  $C.V_{21} \leftarrow A.V_{21} \odot^T B.V_{21}$
  $C := \begin{bmatrix} C_{11} & C.U_{12}\, C.V_{12}^* \\ C.U_{21}\, C.V_{21}^* & C_{22} \end{bmatrix}$
  return C
end procedure

procedure HSS_HADAM(A, B)
  On level $\ell = p$, for $i = 1, \dots, 2^p$
    $C(I_i^p, I_i^p) \leftarrow A(I_i^p, I_i^p) \circ B(I_i^p, I_i^p)$
    $C.U_i^{(p)} = A.U_i^{(p)} \odot^T B.U_i^{(p)}$
    $C.V_i^{(p)} = A.V_i^{(p)} \odot^T B.V_i^{(p)}$
  for $\ell = p-1, \dots, 1$ **do**
    $C.R_{U,i}^{(\ell)} \leftarrow A.R_{U,i}^{(\ell)} \otimes B.R_{U,i}^{(\ell)}$
    $C.R_{V,i}^{(\ell)} \leftarrow A.R_{V,i}^{(\ell)} \otimes B.R_{V,i}^{(\ell)}$
    $C.S_{i,j}^{(\ell)} \leftarrow A.S_{i,j}^{(\ell)} \otimes B.S_{i,j}^{(\ell)}$
  end for
  return C
end procedure

FIG. 9. *Pseudocodes of Hadamard product $C = A \circ B$ in the HODLR format (on the left) and the HSS format (on the right). We used the dot notation (e.g., $C.U_{12}$) to distinguish the parameters in the representation of the matrices A, B, C.*

$$C \in \mathbb{C}^{n \times q}, \quad D \in \mathbb{C}^{n \times m}, \qquad C \odot^T D := \begin{bmatrix} c_1^T \otimes d_1^T \\ c_2^T \otimes d_2^T \\ \vdots \\ c_n^T \otimes d_n^T \end{bmatrix} \in \mathbb{C}^{n \times qm},$$

with c_i^T and d_i^T denoting the ith rows of C and D, respectively.

Equation (4) applied to the off-diagonal blocks immediately provides a HODLR representation, where the HODLR ranks multiply; see Figure 9 (left).

For the HSS format we need to specify how to update the translation operators. To this end we remark that

$$\begin{aligned} \left(\begin{bmatrix} \widetilde{U}_1 & 0 \\ 0 & \widehat{U}_1 \end{bmatrix} R_{U,1} \right) \odot^T \left(\begin{bmatrix} \widetilde{U}_2 & 0 \\ 0 & \widehat{U}_2 \end{bmatrix} R_{U,2} \right) &= \begin{bmatrix} \widetilde{U}_1 & 0 \\ 0 & \widehat{U}_1 \end{bmatrix} \odot^T \begin{bmatrix} \widetilde{U}_2 & 0 \\ 0 & \widehat{U}_2 \end{bmatrix} (R_{U,1} \otimes R_{U,2}) \\ &= \begin{bmatrix} \widetilde{U}_1 \odot^T \widetilde{U}_2 & 0 \\ 0 & \widehat{U}_1 \odot^T \widehat{U}_2 \end{bmatrix} (R_{U,1} \otimes R_{U,2}), \end{aligned}$$

where we used [36, property (4) in section 2.1] to obtain the first identity. Putting all the pieces together yields the procedure for the Hadamard product of two HSS matrices; see Figure 9 (right).

4.5. Recompression. The term recompression refers to the following task: Given a $(\mathcal{T}_p, k)$-HODLR/HSS matrix A and a tolerance τ we aim at constructing a $(\mathcal{T}_p, \widetilde{k})$-HODLR/HSS matrix $\widetilde{A}$, with $\widetilde{k} \le k$ as small as possible, such that $\|A - \widetilde{A}\|_2 \le c \cdot \tau$ for some constant c depending on the format and the cluster tree $\mathcal{T}_p$.

The recompression of a HODLR matrix applies a well-known QR-based procedure [28, section 2.5] to efficiently recompress each (factorized) off-diagonal block. This procedure ensures that the error in each block is bounded by τ, yielding an overall accuracy $\|A - \widetilde{A}\|_2 \le p \cdot \tau$.

The recompression of an HSS matrix uses the algorithm from [54, section 5], which proceeds in two phases. In the first phase, the HSS representation is transformed to the so-called proper form such that all factors $U_i^{(\ell)}$ and $V_i^{(\ell)}$ have orthonormal columns on every level $\ell = 1, \dots, p$. This moves all (near) linear dependencies to the

core factors. In the second phase, these core factors are compressed by truncated SVD in a top-to-bottom fashion, while ensuring the nestedness and the proper form of the representation. The output $\widetilde{A}$ satisfies $\|A-\widetilde{A}\|_2 \leq 2\frac{\sqrt{2^p}-1}{\sqrt{2}-1}\cdot\tau \approx \sqrt{n}/n_{\min}\tau$; see Appendix A for a more detailed description of the algorithm and an error analysis.

The command `compress` carries out the recompression discussed above. Additionally to this explicit involvement, most of the algorithms in the toolbox involve recompression techniques implicitly. Performing arithmetic operations often leads to HODLR/HSS representations with ranks larger than necessary to attain the desired accuracy. For instance, if A and B are $(\mathcal{T}_p, k_A)$-HSS and $(\mathcal{T}_p, k_B)$-HSS matrices, respectively, then both $A+B$ and $A\cdot B$ are exactly represented as $(\mathcal{T}_p, k_A+k_B)$-HSS matrices. However, k_A+k_B is usually an overestimate of the required HSS rank and recompression can be used to limit this rank growth.

When applying recompression to the output A of an arithmetic operation, the toolbox proceeds by first estimating $\|A\|_2$ by means of the power method on AA^*. Then recompression is applied with the tolerance $\tau = \|A\|_2\cdot\epsilon$, where ϵ is the global tolerance discussed in section 3.

The matrix-matrix multiplication in the HODLR format requires some additional care due to the accumulation of low-rank updates from recursive calls [18]. Currently, our implementation performs intermediate recompression after each low-rank update with accuracy τ.

5. Examples and applications. In this section, we illustrate the use of the hm-toolbox for a range of applications.

All experiments have been performed on a server with a Xeon CPU E5-2650 v4 running at 2.20GHz; for each test the running process has been allocated 8 cores and 128 GB of RAM. The algorithms are implemented in MATLAB and tested under MATLAB2017a, with MKL BLAS version 11.3.1, using the 8 cores available.

If not stated otherwise, the parameters ϵ and $n_{\min}$ are set to their default values.

5.1. Fast Toeplitz solver. HSS matrices can be used to design a superfast solver for Toeplitz linear systems. We briefly review the approach in [55] and describe its implementation that is contained in the function `toeplitz_solve` of the toolbox.

Let T be an $n\times n$ Toeplitz matrix

$$T = \begin{bmatrix} t_0 & t_1 & \dots & t_{n-1} \\ t_{-1} & t_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 \\ t_{1-n} & \dots & t_{-1} & t_0 \end{bmatrix}.$$

In particular, the entries on every diagonal of T are constant and the matrix is completely described by the $2n-1$ real or complex scalars $t_{1-n},\dots,t_{n-1}$.

It is well known that a Toeplitz matrix T satisfies the so-called displacement equation

$$Z_1 T - T Z_{-1} = GH^T, \tag{5}$$

where

$$G = \begin{bmatrix} 1 & 2t_0 \\ 0 & t_{n-1}+t_{-1} \\ 0 & t_{n-2}+t_{-2} \\ \vdots & \vdots \\ 0 & t_1+t_{1-n} \end{bmatrix}, \qquad H = \begin{bmatrix} t_{1-n}-t_1 & 0 \\ t_{2-n}-t_2 & 0 \\ \vdots & 0 \\ t_1-t_{n-1} & \vdots \\ 0 & 1 \end{bmatrix}, \qquad Z_t := \begin{bmatrix} 0 & t \\ I_{n-1} & 0 \end{bmatrix}.$$

Here, Z_1 is a circulant matrix, which is diagonalized by the normalized inverse discrete Fourier transform

$$\Omega_n = \frac{1}{\sqrt{n}}(\omega_n^{(i-1)(j-1)})_{1\le i,j\le n}, \qquad \Omega_n Z_1 \Omega_n^* = \operatorname{diag}(1, \omega_n, \dots, \omega_n^{(n-1)}) =: D_1,$$

with $\omega_n = e^{\frac{2\pi \mathrm{i}}{n}}$. Let us call $D_0 := \operatorname{diag}(1, \omega_{2n}, \dots, \omega_{2n}^{(n-1)})$. Then, applying Ω_n from the left and $D_0^*\Omega_n^*$ from the right of (5) leads to another displacement equation [31],

$$D_1\mathcal{C} - \mathcal{C}D_{-1} = \widehat{G}\widehat{F}^T, \tag{6}$$

where

$$\mathcal{C} = \Omega_n T D_0^* \Omega_n^*, \qquad \widehat{G} = \Omega_n G, \qquad \widehat{F} = \Omega_n D_0 H$$

and $D_{-1} = \omega_{2n} D_1$. Since the linear coefficients of (6) are diagonal matrices, the matrix $\mathcal{C}$ is a Cauchy-like matrix of the following form:

$$\mathcal{C} = \left(\frac{\widehat{G}_i \widehat{H}_j^T}{\omega_{2n}^{2(i-1)} - \omega_{2n}^{2j-1}} \right)_{1\le i,j\le n}, \tag{7}$$

where $\widehat{G}_i, \widehat{H}_j$ indicate the ith and jth rows of G and H, respectively.

The fundamental idea of the superfast solver from [55] consists of representing the Cauchy matrix $\mathcal{C}$ in the HSS format. A linear system $Tx = b$ can be turned into $\mathcal{C}y = z$, with $y = \Omega D_0 x$ and $z = \Omega_n b$. Exploiting the HSS structure of $\mathcal{C}$ provides an efficient solution of $\mathcal{C}y = z$. The solution x of the original system is retrieved with an inverse FFT and a diagonal scaling, which can be performed with $\mathcal{O}(n \log n)$ flops.

The compression of $\mathcal{C}$ in the HSS format is performed using the `'handle'` constructor described in section 3. Indeed, given a vector $x \in \mathbb{C}^n$ we see that $\mathcal{C}x = \Omega_n T D_0^* \Omega_n^* x$. Therefore, we can evaluate the matrix vector product by means of FFTs and a diagonal scaling. We assume to have at our disposal an FFT-based matrix-vector multiplication for Toeplitz matrices. The latter is used to implement an efficient routine `C_matvec` that performs the matrix vector product with $\mathcal{C}$. Analogously, a routine `C_matvec_transp` for $\mathcal{C}^*$ is constructed.

The MATLAB code of `toeplitz_solve` (which is included in the toolbox) is sketched in the following:

```
function x = toeplitz_solve(c, r, b)
%n, Gh and Fh defined as above
d0  = exp(1i * pi / n .* (0 : n - 1));
d1  = d0 .^ 2;
dm1 = exp(1i * pi / n) * d1;
C = hss('handle',...
@(v)   C_matvec(c, r, d0, v), ...
@(v)   C_matvec_transp(c, r, d0, v), ...
@(i,j) (Gh(i,:) * Fh(j,:)') ./ (d1(i).' - dm1(j)), n, n)
    ;
z = ifft(b);
y = C \ z;
x = d0' .* fft(y);
end
```

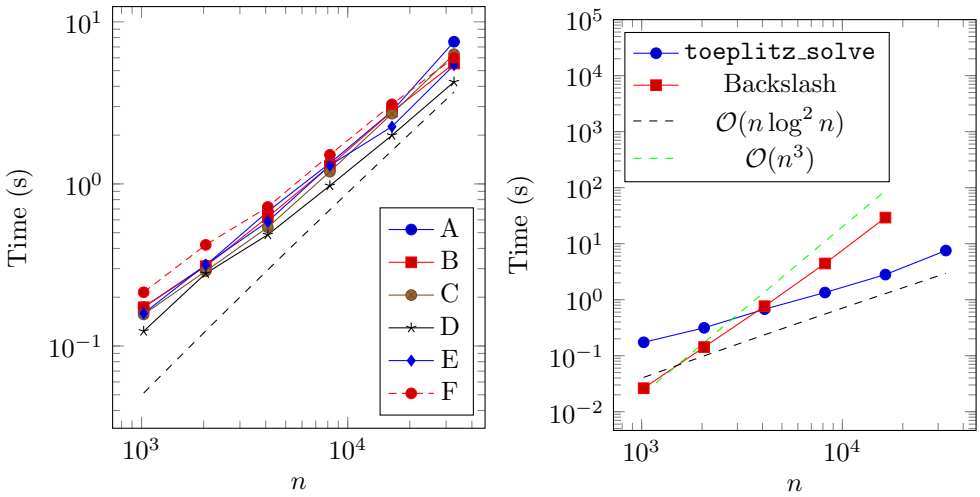

FIG. 10. *Left: Execution time (in seconds) for* `toeplitz_solve` *applied to the Toeplitz matrices A to F from* [55] *and a dashed line indicating an* $\mathcal{O}(n \log^2 n)$ *growth. Right: Execution times for* `toeplitz_solve` *versus MATLAB "backslash" applied to the Toeplitz matrix A.*

TABLE 4
Relative residuals for `toeplitz_solve` *with global tolerance* $\epsilon = 10^{-10}$ *applied to the Toeplitz matrices A to F from* [55].

Size	A	B	C	D	E	F
1,024	$6.24 \cdot 10^{-11}$	$1.49 \cdot 10^{-9}$	$1.07 \cdot 10^{-14}$	$5.02 \cdot 10^{-15}$	$3.52 \cdot 10^{-15}$	$1.19 \cdot 10^{-10}$
2,048	$1.14 \cdot 10^{-10}$	$1.22 \cdot 10^{-9}$	$1.31 \cdot 10^{-12}$	$2.95 \cdot 10^{-15}$	$9.47 \cdot 10^{-15}$	$1.7 \cdot 10^{-10}$
4,096	$9.04 \cdot 10^{-11}$	$1.58 \cdot 10^{-9}$	$5.23 \cdot 10^{-13}$	$9.44 \cdot 10^{-16}$	$3.91 \cdot 10^{-14}$	$1.3 \cdot 10^{-10}$
8,192	$1.44 \cdot 10^{-10}$	$2.81 \cdot 10^{-9}$	$1.11 \cdot 10^{-12}$	$1.7 \cdot 10^{-15}$	$1.26 \cdot 10^{-14}$	$1.28 \cdot 10^{-10}$
16,384	$9.08 \cdot 10^{-10}$	$5.92 \cdot 10^{-9}$	$3.17 \cdot 10^{-12}$	$6.08 \cdot 10^{-16}$	$2.58 \cdot 10^{-14}$	$1.5 \cdot 10^{-10}$
32,768	$2.17 \cdot 10^{-9}$	$7.4 \cdot 10^{-11}$	$2.64 \cdot 10^{-12}$	$5.99 \cdot 10^{-17}$	$2.72 \cdot 10^{-14}$	$1.9 \cdot 10^{-10}$

The whole procedure can be carried out in $\mathcal{O}(k^2 n + kn \log n)$ flops, where k is the HSS rank of the Cauchy-like matrix $\mathcal{C}$. Since k is $\mathcal{O}(\log n)$ [55], the solver has a complexity of $\mathcal{O}(n \log^2 n)$ (assuming that the HSS constructor for the Cauchy-like matrix needs $\mathcal{O}(k)$ matrix-vector products).

We have tested our implementation on the matrices—named from A to F— considered in [55]. The right-hand side is obtained by calling `randn(n,1)` and we set ϵ to 10^{-10}. The timings are reported in Figure 10 and the relative residuals $\frac{\|Tx-b\|_2}{\|T\|_2\|x\|_2+\|v\|_2}$ in Table 4.

5.2. Matrix functions for banded matrices. The computation of matrix functions arises in a variety of settings. When A is banded, the banded structure is sometimes numerically preserved by $f(A)$ [6, 7], in the sense that $f(A)$ can be well approximated by a banded matrix. For example, this is the case for an entire function f of a symmetric matrix A, provided that the width of the spectrum of A remains modest. In other cases, such as for matrices arising from the discretization of unbounded operators, $f(A)$ may lose approximate sparsity. Nevertheless, as discussed in [23], and demonstrated in the following, $f(A)$ can be highly structured and admit an accurate HSS or HODLR approximation.

TABLE 5
Relative errors for the approximation of the matrix exponential in the HODLR and HSS formats.

n	Error (HSS)	Error (HODLR)	Error (`expm`)	$\|A\|_2$
512	$4.29 \cdot 10^{-9}$	$4.12 \cdot 10^{-9}$	$6.56 \cdot 10^{-11}$	$1.04 \cdot 10^{6}$
1,024	$1.74 \cdot 10^{-8}$	$1.79 \cdot 10^{-8}$	$2.86 \cdot 10^{-10}$	$4.19 \cdot 10^{6}$
2,048	$7.37 \cdot 10^{-8}$	$7.24 \cdot 10^{-8}$	$1.47 \cdot 10^{-9}$	$1.68 \cdot 10^{7}$
4,096	$3.08 \cdot 10^{-7}$	$2.97 \cdot 10^{-7}$	$4.74 \cdot 10^{-9}$	$6.71 \cdot 10^{7}$
8,192	$1.15 \cdot 10^{-6}$	$1.14 \cdot 10^{-6}$	$1.88 \cdot 10^{-8}$	$2.68 \cdot 10^{8}$
16,384	$4.81 \cdot 10^{-6}$	$4.68 \cdot 10^{-6}$	$6.53 \cdot 10^{-8}$	$1.07 \cdot 10^{9}$

As an example, we consider the function $f(z) = e^z$ and the 1D discrete Laplacian

$$A = -\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix} \in \mathbb{C}^{n\times n}, \qquad h = \frac{1}{n-1}. \tag{8}$$

The `expm` function included in the toolbox computes the exponential of A in the HSS and HODLR formats via a Padé expansion of degree [13/13] combined with scaling and squaring [33]. For relatively small sizes (up to 16384), we compare the execution time with the one of the `expm` function included in MATLAB. We compute a reference solution from the spectral decomposition of A, which is known in closed form, and use it to check the relative accuracy (in the spectral norm) of MATLAB `expm` and the corresponding HODLR and HSS functions; see Table 5. The left plot of Figure 11 shows that the break-even point for the matrix size, where exploiting structure becomes beneficial in terms of execution time, is around 8192. For nonsymmetric matrices, this threshold reduces to around 1000. For example, computing the exponential of the stiffness matrix of the convection diffusion problem in [35, section 5.3], for $n = 1024$, requires a computational time of about 1 second with all three versions of `expm`. For $n = 4096$ we measure a computational time of about 110 seconds for MATLAB's `expm` and of about 4.5 seconds for the corresponding HODLR/HSS functions. One can also observe, in Figure 11, the slightly better asymptotic complexity of HSS with respect to HODLR.

As the norm of A grows as $\mathcal{O}(n^2)$, the decay of off-diagonal entries can be expected to stay moderate. To verify this, we have computed a sparse approximant to e^A by discarding all entries smaller than $10^{-5} \cdot \max_{i,j} |(e^A)_{ij}|$ in the result obtained with MATLAB `expm`. The threshold has been chosen a posteriori to ensure an accuracy similar to the one obtained with HODLR/HSS arithmetic. The right plot of Figure 11 shows that approximate sparsity is not very effective in this setting; the memory consumption still grows quadratically with n. In contrast, the growth is much slower for the HODLR and HSS formats.

5.3. Matrix equations and 2D fractional PDEs. It has been recently noticed that discretizations of 1D fractional differential operators $\frac{\partial^\alpha}{\partial x^\alpha}$, $\alpha \in (1, 2)$, can be efficiently represented by HODLR matrices [40]. We consider 2D separable operators arising from a fractional PDE of the form

$$\begin{cases} \frac{\partial^\alpha u(x,y)}{\partial x^\alpha} + \frac{\partial^\alpha u(x,y)}{\partial y^\alpha} = f(x,y), & (x,y) \in \Omega := (0,1)^2, \\ u(x,y) \equiv 0, & (x,y) \in \mathbb{R}^2 \setminus \Omega. \end{cases} \tag{9}$$

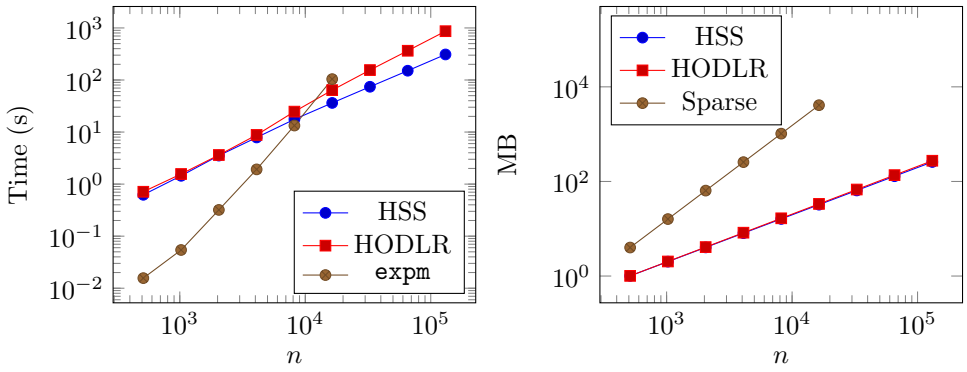

FIG. 11. *Left: Execution times for computing of e^A, with A being the discrete 1D Laplacian. Right: Memory consumption (in MBytes) in the HODLR and HSS formats, compared to the sparse approximant obtained by thresholding entries.*

Discretizing (9) on a tensorized $(n+2)\times(n+2)$ grid provides an $n^2\times n^2$ matrix of the form $\mathcal{M} := A\otimes I + I\otimes A$ and a vector $b\in\mathbb{R}^{n^2}$ containing the representation of the right-hand side $f(x,y)$. Thanks to the Kronecker structure, the linear system $\mathcal{M}x = b$ can be recast into the matrix equation

$$AX + XA^T = C, \qquad \mathrm{vec}(C) = b, \qquad \mathrm{vec}(X) = x. \tag{10}$$

If C is a low-rank matrix—a condition sometimes satisfied in the applications—the solution X is numerically low-rank and it is efficiently approximated via *rational Krylov subspace methods* [46]. The latter require fast procedures for the matrix-vector product and the solution of shifted linear systems with the matrix A. If A is represented in the HODLR or HSS format this requirement is satisfied. In particular, the Lyapunov solver `ek_lyap` included in the hm-toolbox is based on the *extended Krylov subspace method*, described in [46].

We consider a simple example where we choose $\alpha = 1.7$ and the finite difference discretization described in [42]. In this setting, the matrix A is given by

$$A = T_{\alpha,n} + T_{\alpha,n}^T, \qquad T_{\alpha,n} = -\frac{1}{\Delta x^\alpha}\begin{bmatrix} g_1^{(\alpha)} & g_0^{(\alpha)} & 0 & \cdots & 0 & 0 \\ g_2^{(\alpha)} & g_1^{(\alpha)} & g_0^{(\alpha)} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ g_{n-1}^{(\alpha)} & \ddots & \ddots & \ddots & g_1^{(\alpha)} & g_0^{(\alpha)} \\ g_n^{(\alpha)} & g_{n-1}^{(\alpha)} & \cdots & \cdots & g_2^{(\alpha)} & g_1^{(\alpha)} \end{bmatrix},$$

where $g_j^{(\alpha)} = (-1)^j\binom{\alpha}{k}$. The matrix A has Toeplitz structure and it has been proven to have off-diagonal blocks of (approximate) low rank in [40]. The source term is $f(x,y) = \sin(2\pi x)\sin(2\pi y)$ and the matrix C containing its samplings has rank 1.

To retrieve the HODLR representation of A we rely on the Toeplitz constructor:

```
Dx = 1/(n + 2);
[c, r] = fractional_symbol(alpha, n);
T = hodlr('toeplitz', c, r, n) / Dx^alpha;
A = T + T';
```

TABLE 6
Performances of `ek_lyap` *with HODLR and HSS matrices.*

Size	hodlr			hss			rank(X)
	T_{build}	T_{tot}	Res	T_{build}	T_{tot}	Res	
1,024	0.06	0.14	$1.21 \cdot 10^{-8}$	0.14	0.33	$1.21 \cdot 10^{-8}$	23
2,048	0.06	0.2	$1.19 \cdot 10^{-8}$	0.24	0.66	$1.19 \cdot 10^{-8}$	27
4,096	0.14	0.54	$9.03 \cdot 10^{-9}$	0.47	1.45	$9.03 \cdot 10^{-9}$	31
8,192	0.3	1.28	$9.95 \cdot 10^{-9}$	1.03	3.2	$9.95 \cdot 10^{-9}$	35
16,384	0.65	2.99	$8.17 \cdot 10^{-9}$	1.98	6.42	$8.4 \cdot 10^{-9}$	39
32,768	1.32	6.68	$1.15 \cdot 10^{-8}$	4.13	12.98	$8.82 \cdot 10^{-9}$	42
65,536	2.83	14.91	$1.08 \cdot 10^{-8}$	8.12	27.06	$9.83 \cdot 10^{-9}$	46
$1.31 \cdot 10^5$	5.71	32.7	$5.5 \cdot 10^{-8}$	16.89	60.32	$2.74 \cdot 10^{-8}$	50

We combine this with `ek_lyap` in order to solve (10):

```
u = sin(2 * pi * (1:n) / (n + 2))';
Xu = ek_lyap(A, u, inf, 1e-6);
X = hodlr('low-rank', Xu, Xu);
U = hodlr('low-rank', u, u);
Res = norm(A * X + X * A + U) / norm(U);
```

The obtained results are reported in Table 6, where

- T_{build} indicates the time for constructing the HODLR or HSS representation,
- T_{tot} indicates the total time of the procedure,
- Res denotes the residual associated with the approximate solution X: $\|AX + XA + C\|_2/\|C\|_2$.

The results demonstrate the linear polylogarithmic asymptotic complexity of the proposed scheme.

6. Conclusions. We have presented the hm-toolbox, a MATLAB software for working with HODLR and HSS matrices. Based on state-of-the-art and newly developed algorithms, its functionality matches much of the functionality available in MATLAB for dense matrices, while most existing software packages for matrices with hierarchical low-rank structures focus on specific tasks, most notably linear systems. Nevertheless, there is room for further improvement and future work. In particular, the range of constructors could be extended further by advanced techniques based on function expansions and randomized sampling. Also, the full range of matrix functions and other nonstandard linear algebra tasks is not fully exhausted by our toolbox.

Appendix A. HSS recompression: Algorithm and error analysis. Here, we provide a description and an analysis of the algorithm from [54, section 5], which performs the recompression of an HSS matrix A with respect to a certain tolerance τ. As discussed in section 4.5, we suppose that A is already in proper form, i.e., its factors $U_i^{(\ell)}, V_i^{(\ell)}$ have orthonormal columns for all i, ℓ.

The recompression procedures handle HSS block rows and HSS block columns in an analogous manner; to simplify the exposition we only describe the compression of HSS block rows. For this purpose, we consider the following partition of the translation operators:

$$R_{U,i}^{(\ell)} = \begin{bmatrix} R_{U,i,1}^{(\ell)} \\ R_{U,i,2}^{(\ell)} \end{bmatrix} \in \mathbb{C}^{2k \times k}, \qquad R_{U,i,h}^{(\ell)} \in \mathbb{C}^{k \times k} \quad h = 1, 2.$$

For each level $\ell = 1, \dots, p$ and every $i = 1, \dots, 2^\ell$ the algorithm has access to a matrix W_i—having k rows—such that the ith HSS block row can be written as

$$\begin{bmatrix} U_{2i-1}^{(\ell+1)} R_{U,i,1}^{(\ell)} W_i \widetilde{V}_i^* \\ U_{2i}^{(\ell+1)} R_{U,i,2}^{(\ell)} W_i \widetilde{V}_i^* \end{bmatrix} \tag{11}$$

for some matrix $\widetilde{V}_i$ having orthonormal columns. At level $\ell = 1$ the algorithm chooses $W_1 = S_{1,2}^{(1)}$, $W_2 = S_{2,1}^{(1)}$, $\widetilde{V}_1 = V_2^{(1)}$, and $\widetilde{V}_2 = V_1^{(1)}$. Note that the relation (11) allows us to write the HSS block rows at level $\ell + 1$ as

$$U_{2i-1}^{(\ell+1)} \begin{bmatrix} S_{i,i+1}^{(\ell+1)} & R_{U,i,1}^{(\ell)} W_i \end{bmatrix} \breve{V}_1^*, \qquad U_{2i}^{(\ell+1)} \begin{bmatrix} S_{i+1,i}^{(\ell+1)} & R_{U,i,2}^{(\ell)} W_i \end{bmatrix} \breve{V}_2^*,$$

where $\breve{V}_1, \breve{V}_2$ are suitable row permutations of $\widetilde{V}_i \oplus V_{2i}^{(\ell)}$ and $\widetilde{V}_i \oplus V_{2i-1}^{(\ell)}$, respectively.

The algorithm proceeds with the following steps:

- Compute the truncated SVDs (neglecting singular values below the tolerance τ)

$$\widehat{U}_1 \widehat{S}_1 \begin{bmatrix} \widehat{V}_{11}^* & \widehat{V}_{12}^* \end{bmatrix} \approx \begin{bmatrix} S_{i,i+1}^{(\ell)} & R_{U,i,1}^{(\ell)} W_i \end{bmatrix},$$
$$\widehat{U}_2 \widehat{S}_2 \begin{bmatrix} \widehat{V}_{21}^* & \widehat{V}_{22}^* \end{bmatrix} \approx \begin{bmatrix} S_{i+1,i}^{(\ell)} & R_{U,i,2}^{(\ell)} W_i \end{bmatrix}.$$

  In particular, we have the approximate factorizations

$$U_{2i-1}^{(\ell+1)} \begin{bmatrix} S_{i,i+1}^{(\ell+1)} & R_{U,i,1}^{(\ell)} W_i \end{bmatrix} \breve{V}_1^* \approx U_{2i-1}^{(\ell+1)} \widehat{U}_1 \begin{bmatrix} \widehat{S}_1 \widehat{V}_{11}^* & \widehat{U}_1^* R_{U,i,1}^{(\ell)} W_i \end{bmatrix} \breve{V}_1^*,$$
$$U_{2i}^{(\ell+1)} \begin{bmatrix} S_{i+1,i}^{(\ell+1)} & R_{U,i,2}^{(\ell)} W_i \end{bmatrix} \breve{V}_2^* \approx U_{2i}^{(\ell+1)} \widehat{U}_2 \begin{bmatrix} \widehat{S}_2 \widehat{V}_{21}^* & \widehat{U}_2^* R_{U,i,2}^{(\ell)} W_i \end{bmatrix} \breve{V}_2^*.$$

- The above factorizations are equivalent to performing the following updates:

$$S_{i,i+1}^{(\ell)} = \widehat{S} \widehat{V}_{11}^*, \qquad R_{U,2i-1}^{(\ell+1)} = R_{U,2i-1}^{(\ell+1)} \widehat{U}_1, \qquad R_{U,i}^{(\ell)} = \begin{bmatrix} \widehat{U}_1 & \\ & \widehat{U}_2 \end{bmatrix} R_{U,i}^{(\ell)},$$
$$S_{i+1,i}^{(\ell)} = \widehat{S} \widehat{V}_{21}^*, \qquad R_{U,2i}^{(\ell+1)} = R_{U,2i}^{(\ell+1)} \widehat{U}_2.$$

The analogous operations are performed on the HSS block columns. We notice that the truncated SVDs introduced an error with norm bounded by τ in every HSS block row and column on every level. This leads to the following.

PROPOSITION A.1. *Let A be a $(\mathcal{T}_p, k)$-HSS matrix for some $p, k \in \mathbb{N}$ and $\widetilde{A}$ the output of the recompression algorithm described above, using the truncation tolerance $\tau > 0$. Then, $\|A - \widetilde{A}\|_2 \leq 2 \frac{\sqrt{2^p} - 1}{\sqrt{2} - 1} \tau$.*

Proof. We remark that at each level ℓ the algorithm introduces row and column perturbations of the form

$$E^{(\ell)} + \left(F^{(\ell)} \right)^T = \begin{bmatrix} E_1^{(\ell)} & \dots & E_{2^\ell}^{(\ell)} \end{bmatrix} + \begin{bmatrix} F_1^{(\ell)} & \dots & F_{2^\ell}^{(\ell)} \end{bmatrix}^T,$$

where $E_j^{(\ell)}, F_j^{(\ell)}$ have norm bounded by τ for every j. Since $\|E^{(\ell)}\|_2, \|F^{(\ell)}\|_2 \leq \sqrt{2^\ell} \tau$, the claim follows by summing for $\ell = 1, \dots, p$. ∎

REFERENCES

[1] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, *Fast direct methods for Gaussian processes*, IEEE Trans. Pattern Anal. Mach. Intell., 38 (2016), pp. 252–265, https://doi.org/10.1109/TPAMI.2015.2448083.

[2] S. Ambikasaran, K. Singh, and S. Sankaran, *HODLRlib: A Library for Hierarchical Matrices*, J. Open Source Software, 4 (2019), 1167, https://doi.org/10.21105/joss.01167.

[3] J. Ballani and D. Kressner, *Matrices with hierarchical low-rank structures*, in Exploiting Hidden Structure in Matrix Computations: Algorithms and Applications, Lecture Notes in Math. 2173, Springer, Cham, 2016, pp. 161–209.

[4] M. Bebendorf, *AHMED*, https://github.com/xantares/ahmed, (2019).

[5] M. Bebendorf and W. Hackbusch, *Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients*, Numer. Math., 95 (2003), pp. 1–28, https://doi.org/10.1007/s00211-002-0445-6.

[6] M. Benzi, P. Boito, and N. Razouk, *Decay properties of spectral projectors with applications to electronic structure*, SIAM Rev., 55 (2013), pp. 3–64, https://doi.org/10.1137/100814019.

[7] M. Benzi and N. Razouk, *Decay bounds and $O(n)$ algorithms for approximating functions of sparse matrices*, Electron. Trans. Numer. Anal., 28 (2007), pp. 16–39.

[8] D. A. Bini, S. Massei, and L. Robol, *Efficient cyclic reduction for quasi-birth-death problems with rank structured blocks*, Appl. Numer. Math., 116 (2017), pp. 37–46, https://doi.org/10.1016/j.apnum.2016.06.014.

[9] D. A. Bini, S. Massei, and L. Robol, *On the decay of the off-diagonal singular values in cyclic reduction*, Linear Algebra Appl., 519 (2017), pp. 27–53, https://doi.org/10.1016/j.laa.2016.12.027.

[10] S. Börm, *HLIBPro*, https://www.hlibpro.com/.

[11] S. Börm, *$\mathcal{H}_2$-Matrices—An Efficient Tool for the Treatment of Dense Matrices*, Habilitationsschrift, Christian-Albrechts-Universität zu Kiel, 2006.

[12] S. Börm, *Efficient Numerical Methods for Non-Local Operators: $\mathcal{H}^2$-Matrix Compression, Algorithms and Analysis*, EMS Tracts Math. 14, European Mathematical Society, Zürich, 2010, https://doi.org/10.4171/091.

[13] S. Börm, *H2Lib*, https://github.com/H2Lib/H2Lib, (2019).

[14] S. Börm, L. Grasedyck, and W. Hackbusch, *Hierarchical Matrices*, Lecture Note 21/2003, MPI-MIS Leipzig, 2006, http://www.mis.mpg.de/preprints/ln/lecturenote-2103.pdf.

[15] P. Businger and G. H. Golub, *Linear least squares solutions by Householder transformations*, Numer. Math., 7 (1965), pp. 269–276, https://doi.org/10.1007/BF01436084.

[16] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Some fast algorithms for sequentially semiseparable representations*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 341–364, https://doi.org/10.1137/S0895479802405884.

[17] 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, https://doi.org/10.1137/S0895479803436652.

[18] J. Dölz, H. Harbrecht, and M. D. Multerer, *On the best approximation of the hierarchical matrix product*, SIAM J. Matrix Anal. Appl., 40 (2019), pp. 147–174, https://doi.org/10.1137/18M1189373.

[19] Y. Eidelman, I. Gohberg, and I. Haimovici, *Separable Type Representations of Matrices and Fast Algorithms, Volume 1: Basics. Completion Problems. Multiplication and Inversion Algorithms*, Oper. Theory Adv. Appl. 234, Birkhäuser/Springer, Basel, 2014.

[20] Y. Eidelman, I. Gohberg, and I. Haimovici, *Separable Type Representations of Matrices and Fast Algorithms, Volume 2: Eigenvalue Method*, Oper. Theory Adv. Appl. 235, Birkhäuser/Springer, Basel, 2014.

[21] M. Faustmann, J. M. Melenk, and D. Praetorius, *Existence of $\mathcal{H}$-matrix approximants to the inverse of BEM matrices: The hyper-singular integral operator*, IMA J. Numer. Anal., 37 (2017), pp. 1211–1244, https://doi.org/10.1093/imanum/drw024.

[22] I. P. Gavrilyuk, W. Hackbusch, and B. N. Khoromskij, *$\mathcal{H}$-matrix approximation for the operator exponential with applications*, Numer. Math., 92 (2002), pp. 83–111, https://doi.org/10.1007/s002110100360.

[23] I. P. Gavrilyuk, W. Hackbusch, and B. N. Khoromskij, *Data-sparse approximation to the operator-valued functions of elliptic operator*, Math. Comp., 73 (2004), pp. 1297–1324, https://doi.org/10.1090/S0025-5718-03-01590-4.

[24] C. J. Geoga, M. Anitescu, and M. L. Stein, *Scalable Gaussian process computations using hierarchical matrices*, J. Comput. Graph. Statist., https://doi.org/10.1080/10618600.2019.1652616.

[25] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, *An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. S358–S384, https://doi.org/10.1137/15M1010117.

[26] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins Stud. Math. Sci., Johns Hopkins University Press, Baltimore, MD, 2013.

[27] L. Grasedyck, *Existence of a low rank or $\mathcal{H}$-matrix approximant to the solution of a Sylvester equation*, Numer. Linear Algebra Appl., 11 (2004), pp. 371–389, https://doi.org/10.1002/nla.366.

[28] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, Springer Ser. Comput. Math. 49, Springer, Heidelberg, 2015, https://doi.org/10.1007/978-3-662-47324-5.

[29] W. Hackbusch, B. N. Khoromskij, and R. Kriemann, *Hierarchical matrices based on a weak admissibility criterion*, Computing, 73 (2004), pp. 207–243, https://doi.org/10.1007/s00607-004-0080-4.

[30] N. Halko, P. G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288, https://doi.org/10.1137/090771806.

[31] G. Heinig, *Inversion of generalized Cauchy matrices and other classes of structured matrices*, in Linear Algebra for Signal Processing (Minneapolis, MN, 1992), IMA Vol. Math. Appl. 69, Springer, New York, 1995, pp. 63–81, https://doi.org/10.1007/978-1-4612-4228-4_5.

[32] N. J. Higham, *Functions of Matrices*, SIAM, Philadelphia, 2008.

[33] N. J. Higham, *The scaling and squaring method for the matrix exponential revisited*, SIAM Rev., 51 (2009), pp. 747–764, https://doi.org/10.1137/090768539.

[34] D. Kressner, P. Kürschner, and S. Massei, *Low-rank updates and divide-and-conquer methods for quadratic matrix equations*, Numer. Algorithms, (2019), https://doi.org/10.1007/s11075-019-00776-w.

[35] D. Kressner, S. Massei, and L. Robol, *Low-rank updates and a divide-and-conquer method for linear matrix equations*, SIAM J. Sci. Comput., 41 (2019), pp. A848–A876, https://doi.org/10.1137/17M1161038.

[36] D. Kressner and L. Periša, *Recompression of Hadamard products of tensors in Tucker format*, SIAM J. Sci. Comput., 39 (2017), pp. A1879–A1902, https://doi.org/10.1137/16M1093896.

[37] D. Kressner and A. Šušnjara, *Fast computation of spectral projectors of banded matrices*, SIAM J. Matrix Anal. Appl., 38 (2017), pp. 984–1009, https://doi.org/10.1137/16M1087278.

[38] D. Kressner and A. Šušnjara, *Fast QR Decomposition of HODLR Matrices*, preprint, arXiv:1809.10585, 2018.

[39] P. G. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 1251–1274, https://doi.org/10.1137/100786617.

[40] S. Massei, M. Mazza, and L. Robol, *Fast solvers for two-dimensional fractional diffusion equations using rank structured matrices*, SIAM J. Sci. Comput., 41 (2019), pp. A2627–A2656, https://doi.org/10.1137/18M1180803.

[41] S. Massei and L. Robol, *Decay bounds for the numerical quasiseparable preservation in matrix functions*, Linear Algebra Appl., 516 (2017), pp. 212–242, https://doi.org/10.1016/j.laa.2016.11.041.

[42] M. M. Meerschaert and C. Tadjeran, *Finite difference approximations for fractional advection-dispersion flow equations*, J. Comput. Appl. Math., 172 (2004), pp. 65–77, https://doi.org/10.1016/j.cam.2004.01.033.

[43] F. Rouet, X. S. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, ACM Trans. Math. Software, 42 (2016), 27, https://doi.org/10.1145/2930660.

[44] Z. Sheng, P. Dewilde, and S. Chandrasekaran, *Algorithms to solve hierarchically semiseparable systems*, in System Theory, the Schur Algorithm and Multidimensional Analysis, Oper. Theory Adv. Appl. 176, Birkhäuser, Basel, 2007, pp. 255–294, https://doi.org/10.1007/978-3-7643-8137-0_5.

[45] H. D. Simon and H. Zha, *Low-rank matrix approximation using the Lanczos bidiagonalization process with applications*, SIAM J. Sci. Comput., 21 (2000), pp. 2257–2274, https://doi.org/10.1137/S1064827597327309.

[46] V. Simoncini, *Computational methods for linear matrix equations*, SIAM Rev., 58 (2016), pp. 377–441, https://doi.org/10.1137/130912839.

[47] R. Vandebril, M. Van Barel, and N. Mastronardi, *Matrix Computations and Semiseparable Matrices, Volume 1: Linear Systems*, Johns Hopkins University Press, Baltimore, MD, 2008.

[48] R. Vandebril, M. Van Barel, and N. Mastronardi, *Matrix Computations and Semiseparable Matrices, Volume 2: Eigenvalue and Singular Value Methods*, Johns Hopkins University Press, Baltimore, MD, 2008.

[49] J. Vogel, J. Xia, S. Cauley, and V. Balakrishnan, *Superfast divide-and-conquer method and perturbation analysis for structured eigenvalue solutions*, SIAM J. Sci. Comput., 38 (2016), pp. A1358–A1382, https://doi.org/10.1137/15M1018812.

[50] A. Šušnjara and D. Kressner, *A Fast Spectral Divide-and-Conquer Method for Banded Matrices*, arXiv:1801.04175, 2018.

[51] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, and M. V. de Hoop, *A parallel geometric multifrontal solver using hierarchically semiseparable structure*, ACM Trans. Math. Software, 42 (2016), 21, https://doi.org/10.1145/2830569.

[52] Y. Xi, J. Xia, S. Cauley, and V. Balakrishnan, *Superfast and stable structured solvers for Toeplitz least squares via randomized sampling*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 44–72, https://doi.org/10.1137/120895755.

[53] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Superfast multifrontal method for large structured linear systems of equations*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 1382–1411, https://doi.org/10.1137/09074543X.

[54] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Linear Algebra Appl., 17 (2010), pp. 953–976, https://doi.org/10.1002/nla.691.

[55] J. Xia, Y. Xi, and M. Gu, *A superfast structured solver for Toeplitz linear systems via randomized sampling*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 837–858, https://doi.org/10.1137/110831982.